Calhoun
Institutional Archive of the Naval Postgraduate School

**Calhoun: The NPS Institutional Archive**

**DSpace Repository**

Faculty and Researchers | Faculty and Researchers' Publications

1988

# Artificial Intelligence through Prolog by Neil C. Rowe

Rowe, Neil C.

Prentice-Hall

http://hdl.handle.net/10945/36984

*Downloaded from NPS Archive: Calhoun*

DUDLEY KNOX LIBRARY

Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

# ADVICE TO THE INSTRUCTOR

If artificial intelligence is a hard subject to learn, it's an even harder subject to teach. Intellectually, it's one of the most difficult areas of computer science, with many complicated ideas to understand. It requires a good deal of background. It's very rigorous in some subareas, and sloppy in others. It's not like any other subject, so students have few analogies on which to draw. Instead, they see and hear distorted media reports about it that encourage unreasonable expectations. So instructors of artificial intelligence courses have their hands full.

But on the other hand, if you can successfully teach an artificial intelligence course in the style and at the level of this book, you've really accomplished something. The students will almost certainly learn a lot. And much of what they'll learn is quite practical, and much of it will tie in with important concepts they've learned in other courses. So don't be easily discouraged.

## Evaluating Student Progress

One major difficulty in teaching artificial intelligence is in planning the test questions and homework exercises. Many instructors have told me how conventional tests just don't seem to work for artificial intelligence courses. The interesting questions that really test if students know anything require more than an hour to do, and multi-hour tests are hard to arrange and unfair to students who just can't show creativity under the pressure of a test. Substantial homework assignments seem the only alternative, and in fact, most of the questions in this book are intended as homework questions. So don't be afraid to give lots of homework if you have the capability to grade it.

Homework has a potentially big disadvantage: the possibility of collaboration or copying of answers between students. You'll just have to judge how worrisome this is for your institution. At MIT in 1973, in my first course in this subject, Patrick Winston deliberately encouraged students to work together on the all-day take-home exams, claiming that grades in the course weren't that important. At Stanford in 1981, when I assisted with the first course, we didn't issue any policy on the homework, and trusted the student's adherence to the Honor Code, which a few students flagrantly violated. At the Naval Postgraduate School, where I grade all papers myself, I make a special effort to compare papers on the first homework assignment, and warn students whose papers seem to be significantly more similar than could be due to pure chance. But I can't say that my experience has been typical.

Still, you have to give some tests. Currently, in my 11-week quarter course (introductory artificial intelligence) I give a 50-minute midterm (with about four questions) and a 110-minute final exam (without about seven questions). The easier exercises in the book, especially those involving following of algorithms, are generally my test questions. I also give three or four homework assignments, evenly spaced through the quarter. Currently, I get an average of about 70% correct on the homework and tests. I recommend open-book tests because there are many terms used in the field, and the material is so hard anyway for students.

## About Good Exercises and Test Questions

I suppose some of you are thinking that it surely must be possible to construct some easier test questions--maybe multiple choice?--than those in this book. I'll argue that it's very difficult to do this and remain fair to the content of AI, because the subject concerns thinking, and you need students to think on a test to verify that they see how computers could do it. Let's examine some question types that are not in my book.

Q1. Define resolution.

That just requires memory regurgitation; there's no assurance that correct answer means the student understands what resolution requires or what it's good for. This sort of bad question is the too-frequent resort of lazy professors.

Q2. Which of these is a control structure: Bayes' Rule, caching, unification, or forward chaining?

Again, mostly memory regurgitation is needed. And it's too easy a question; if students are really so confused that they haven't realized that forward chaining is a control structure, then either standards were too low, the course was really awful, or the students shouldn't be in college. Why not instead kill several birds with one stone, and ask them a more complicated question about forward chaining that they won't make any progress on unless they know it's a control structure, like "Can there be a breadth-first forward chaining?"

Q3. Implement in a computer program the idea expressed on page 76, paragraph 3, line 5.

Learning computer science means doing some programming, but not necessarily a lot, and not necessarily in every course. Though my book uses Prolog, there aren't many programming exercises for the reason that such exercises can require unexpected amounts of student time and effort. I think there are better and faster ways for students to learn the material, ways that do not involve programming, just in the same way there are better ways to learn physics than trying to duplicate every famous experiment in the lab. And also, programming exercises easy for the professor to write like the one above are rarely the most valuable programming experiences for students.

Q4. Discuss the difference between breadth-first search and best-first search.

Broad essay questions can be trouble to grade because they give the student a lot of room to pontificate, and it can be hard to compare student performances when different students cover different points. And it's not clear what level of analysis is an adequate answer to the question, so you'll frequently have students in your office grumbling if you give questions like this.

Q5. Prove that if A implies B or C, and B implies A, then C is true.

Leave proofs for math classes; mathematics instructors know better how to teach proof rigor, and have more reason to teach it. And there's more than enough material to teach in artificial intelligence that other courses don't teach.

Q6. Suzy Seashell is preparing a Honorary Dinner and needs to select a wine. Her choices are Chateau Aulde-Tenni-Chue, Rotgutter Dranow, and Cheap Yellow. Her expert system has rules as follows: ...

OK, I confess I have a few silly questions in my book, to get back at certain peeves. But if there are too many such questions, students lose track of what the course is for, and eventually lose their motivation. A diet of desserts is not good for the health.

Demonstrations

If at all possible, try to use an actual computer during class to show off the programs of the book and other similar programs. It will make some things much clearer for the students. This is particularly valuable at the beginning of the course, but it works well for some of the later programs in the book too; I'd recommend

doing a demonstration once every eight classroom hours. To make this easier, Prentice-Hall is selling tapes and diskettes containing the major programs in the book; see your representative. Devices for projecting computer-terminal screens on a lecture-hall screen are an invaluable for computer science courses, if you haven't tried them before.

A Sample Syllabus

Following is my syllabus for when I gave a quarter course "Artificial Intelligence" based on this book, as the only text, in Summer Quarter 1987. Our quarter was 11 weeks long, and the course met four times a week for 50-minute lecture/recitations (with zero lab hours allocated). Thirty students per quarter has been my average enrollment, and I grade all homework and tests myself.

I have also used other parts of the book in an advanced course, "Expert Systems". I usually use Chapters 7 and 10, plus an expert-systems book. Careful coverage of those two chapters could almost fill a quarter by itself, since many issues in Prolog programming as well as expert systems are raised by those chapters.

##########################################

CS3310, Artificial Intelligence

Summer 1987

Profs. Rowe (Spanagel 528B) and McGhee (Spanagel 512)

Official office hours for Prof. Rowe: Mondays, Wednesdays, and Thursdays 9-10 and 11-12. No official hours for Prof. McGhee; see him anytime.

Grading: 75 points on each of four Homework assignments; 100 points Midterm Exam (open-book); 175 points Final Exam (open-book).

Both sections of this course will cover the same material, and they will have the same homework and tests.

The listed prerequisite for this course is a course in logic, such as Mathematics 2025 or 0125 at this school, but this isn't too important. But it is important to have taken at least one college-level programming course, hopefully two. A data structures course helps considerably, as do software engineering courses.

This course will require at least one hour of independent study outside of class every day (six days a week). The homework and tests will be difficult, with expected class averages around 70%. Questions will emphasize thinking more than memorization, and will often involve applying what you know to new applications. This is definitely a graduate-level course.

The textbook is a set of class notes. For additional reading (not required) see Winston, Artificial Intelligence, second edition (Addison-Wesley, 1984), or the other books listed in the bibliography in the notes (Appendix F).

For the last six terms of this course, 36 A grades were given, 38 A-, 65 B+, 53 B, 22 B-, 3 C+, and 4 C grades.

Schedule (do not hand in the practice problems):

By 7/9: read chapters 1 and 2 of the notes; do for practice problems 2-2 and 2-6

By 7/15: read chapter 3 (look over Appendix A if you need to review logic); do for practice problem 3-9

By 7/22: read chapter 4; do for practice problem 4-11

Homework #1 due Thursday 7/23

By 7/29: read chapter 5 (look over Appendices B and C to review recursion and data structures); do for practice problems 5-2 and 5-14

By 8/5: read chapter 6, sections 6.1-6.6 and 6.8 only; do for practice problem 6-5

Homework #2 due Monday 8/10

By 8/11: read chapter 7, sections 7.1-7.9 only; do for practice problem 7-2

Midterm exam Thursday 8/13 (covering chapters 1-7)

By 8/19: read chapter 8, sections 8.1-8.10 only; do for practice problem 8-3

By 8/25: read chapter 9 except for sections 9.14-9.16; do for practice problem 9-11

By 8/31: read chapter 10, sections 10.1-10.4, 10.6, and 10.7 only; do for practice problem 10-3

Homework #3 due Tuesday 9/1

By 9/7: read chapter 11, sections 11.1-11.7 only; do for practice problem 11-6

By 9/10: read chapter 12; do for practice problem 12-5

By 9/15: read chapter 15

Homework #4 due Wednesday 9/16

Final exam Monday, 9/21, 0800, Sp-421

################################################## Alternative Syllabi

Despite the statements of the Preface (which are designed to encourage people not to skip chapters), flexibility is possible in which chapters are taught. But please try not to skip Chapter 15, which is important.

More and more Prolog courses are being offered at schools. If a Prolog course is required or common for your students, there is still more than enough material in this book for an AI course. Certainly, you can go more quickly through the first five chapters of this book. However, definitely do not skip those chapters entirely, because they contain many important AI ideas not usually covered in Prolog texts.

Some of you may prefer a more formal approach based on logic, perhaps because you teach in a mathematics department or come from a more theoretical background. Then you can take Chapter 14 early, perhaps right

after Chapter 3; Chapter 14 has been designed so it makes minimal references to the rest of the book. You may want to skip the more pragmatic parts of the book, like Chapters 7 and 10. However, if you want to take a formal approach, consider the following argument. Students can't be motivated to learn the syntax and manipulations of logic unless they can see what it is used for in practical applications. My approach of emphasizing the semantics of predicate expression from the very beginning, bringing up formal details slowly and only as absolutely necessary, tends I believe to motivate students more.

Some of you would prefer to spend some time on the specifics of some subarea of AI; natural language seems to be the most commonly mentioned subarea. My book does try to stick with subarea-independent concepts, but actually, there is a lot of natural language material in my book, mixed in with others things as per my integrated approach; note especially sections 2.8, 4.4, 5.9, 6.12, and 12.13. You may want to supplement this with additional material. But before you do, consider the following. Linguistics is an important field of scientific study, but claims of important practical benefits for it have never panned out. There seems little need to have computers communicate with us in English any more, what with major recent advances in graphics, positional input devices, and menu-driven systems. Words take a long time to type, and many people can't type including many who are illiterate, so it's unfair to emphasize words in using computers; an emphasis on words may just be a relic of the overly verbal education most computer researchers receive. There is still some need to process written text by computer, but the increasing computerization of every aspect of human life is making it progressively unimportant. Emphasizing language in an AI course also means teaching science instead of engineering, and I don't think this is what students need in a university: there are too many science courses already.

Some reviewers of the manuscript complained about various omissions and placements of Prolog features in the book. Please understand I'm not trying to teach Prolog; many books available can do that, and it really requires a full quarter of instruction. Instead, I'm trying to use Prolog to illustrate AI concepts, and I'm very concerned about introducing too much too fast. Some specific issues:

--The anonymous variable ("_") is omitted because I believe it encourages poor programming. Intelligent naming of variables is an important part of programming (see section 2.4), an students shouldn't be allowed to avoid or "cop out" on it.

--It's a lot easier to explain the "not" construct by pretending it's a predicate that takes a predicate expression as input, though this isn't the way it's implemented in many Prologs. Explaining it any other way would needlessly complicate the early chapters of the book.

--I ignore ways of entering facts directly into the Prolog interpreter's database (like "data-entry mode"), for a philosophical reason: I think students need to have emphasized the distinction between knowledge (facts and rules) and control structures. If you muddy this distinction, students can get very confused. It's also a distinction important historically in AI.

--The cut predicate is terribly confusing for students, and it's much abused in programming (I was amazed to see a recent Prolog text demonstrate expert systems with a cut symbol at the end of every rule, horrendous programming style), so I postpone it to a place where its complexities are an appropriate match to the material, in Chapter 10. I fervently hope you'll resist the temptation to introduce it earlier to your students, because there's plenty of more important things in AI to confuse them. However, some of the software accompanying this book contains cuts for efficiency, when the versions in the book don't have cuts.

--I do not use complex structures very much in this book (that is, embedded predicate expressions). I hope

you won't either, because it is important for an AI course to maintain some compatibility with Lisp. Embedded lists can do everything that complex structures can, and in a way more analogous to Lisp, so I've used them instead in this book (for instance, in the search path lists of Chapter 10). Nearly all students who continue on to study AI will learn Lisp.

--I haven't been afraid to give some things that don't work in Turbo Prolog. Turbo has generated a lot of initial interest based on its price, but it's unclear how popular it will be in the long run. Its compiler-based approach means quite different processing from interpreted Prologs, and it has a number of important incompatibilities with Clocksin and Mellish Prolog, despite claims of its source, most notably in the "funarg" problem and other issues in the persistence of variable bindings. Several important and interesting programs like forall and doall in Chapter 7 won't work in Turbo, but I feel they're important enough (and relate directly to counterparts in Lisp) that students should see them. Cheaper and simpler dialects of Prolog are desirable, but they shouldn't leave out key features.

.PA

ADDITIONAL PROBLEMS

Note: answers to these questions are at the back of this Manual.

I-1. (for Chapter 3) Consider the Prolog database:

a(X) :- b(X,Y), b(Y,Z). b(1,3). b(2,2). b(4,2).

(a) What will a Prolog interpreter answer to the query:

?- a(2).

(b) List all the possible answers that a Prolog interpreter could find to the following query to this database (not including "no"), if you kept typing semicolons:

?- a(Q), not(b(Q,Q)).

(c) How many times total does the Prolog interpreter backtrack from b(Y,Z) to b(X,Y) in the rule, when you keep typing semicolons, before it answers "no" to the query of part (b)?

I-2 (for Chapter 3). Consider the query:

?- link(A,B), link(B,C).

Suppose there are four "link" facts in the Prolog database. Assume no duplicate facts. Assume the two arguments are never identical within a fact.

(a) What is the maximum number of answers a Prolog interpreter will find for the this query?

(b) What is the maximum number of times a Prolog interpreter will backtrack from the second predicate expression to the first predicate expression before answering?

I-3. (for Chapter 3) Why do these two Prolog queries always give the same set of answers?

?- (a;not(c)), b, c. ?- a, b, c.

I-4 (for Chapter 3). (a) Sometimes unbound variables inside "not"s work OK. Consider this query:

?- bosses(R,S), bosses(T,R), not(bosses(S,U)).

with the database

bosses(poindexter,north). bosses(reagan,poindexter). bosses(north,hall).

What is the first answer the Prolog interpreter will find to the query? (Hint: draw a semantic network.)

(b) For the query and database of part (b), how many times will Prolog backtrack from the second predicate expression to the first in getting the first answer?

I-5. (for Chapter 3) Suppose a Prolog query has 26 answers total (obtained by typing semicolons). Suppose this query is the "and" of some predicate expressions, with no "or"s (semicolons) or "not"s.

(a) Suppose you give the Prolog interpreter a new query just like the old except with another predicate expression "and"ed on the right end. Suppose this new expression doesn't contain any "new" variables, variables not occurring in the old query. Suppose this augmented query has 26 answers too. Must they be the same answers? Why?

(b) For the same situation as (a), suppose we measure the CPU (processing) time to get all possible answers, measuring up until the interpreter types "no". Will the new query take more time than the original query, less time, or sometimes more and sometimes less depending on the query? Why?

(c) Repeat question (b) but assume the new query has 7 answers total (with the original query still having 26). Why?

(d) Repeat question (c) but assume the new predicate expression was "and"ed on the left end of the query, not the right. (Still assume 7 answers total.)

I-6. (for Chapter 4) (a) Which of the following (pick only one) is a Prolog rule for downward inheritance of "owns" with respect to the part-of relationship? ("owns(P,T)" means P owns thing T.)

part_of(P,T) :- owns(P,Q), part_of(Q,T). part_of(P,T) :- owns(U,T), part_of(U,P). part_of(P,T) :- part_of(T,Q), owns(P,T). owns(P,T) :- part_of(Q,P), owns(Q,T). owns(P,T) :- part_of(U,T), owns(U,P). owns(P,T) :- owns(P,Q), owns(Q,T).

(b) Suppose the U.S. Navy owns the Enterprise. Hence the U.S. Government owns the Enterprise; does that follow from the inheritance rule you selected above? If so, what do the variables match? If not, why not?

I-7. (for Chapter 4) Suppose predicate "necessary(X,Y)" means that a specific single action X must occur before specific single action Y can occur.

(a) Is predicate "necessary" transitive?

(b) Consider the predicate "before9(X)" which is true if specific single action X occurred before 9AM on

February 10, 1986. Does "before" inherit with respect to "necessary" in any way? If so, describe how. (Hint: "necessary" can be used backwards.)

I-8. (for Chapter 5) This "delete" predicate takes three arguments: an item, a list, and a list. It says that the result of deleting all occurrences of the item from the first list gives the second list.

delete(X,[],[]). delete(X,[X|L],L2) :- delete(X,L,L2). delete(X,[Y|L],[Y|L2]) :- delete(X,L,L2), not(X=Y).

Consider the query:

?- delete(X,[a,b,c],[a,c]).

(a) What is the first recursive call of "delete"? Identify the arguments of the call.

(b) What answer does the original query (the one printed above) give? (Hint: think what the definition means.)

I-9. (for Chapter 5). Define a predicate "pair(L,PL)" which takes a list of even length as first argument, pairs the adjacent items in the list, and binds that result to its second argument. So for instance:

?- pair([a,b,c,d,e,f],P). P=[[a,b],[c,d],[e,f]]

Show your program working.

I-10. (for Chapter 10) Define a predicate "intersection(S1,S2,I)" which says that set I is the intersection of sets S1 and S2, where all three sets are expressed as lists. (That is, items in I are all the items that occur within both S1 and S2.) You can use any of the definitions of chapter 5 to make the definition simpler, but don't use the "var" predicate. Show your program works correctly for two cases: (1) for S1 and S2 bound, I unbound; (2) for S1 and S2 unbound, I bound. (Remember, things Prolog prints out that start with an underscore symbol ("_") are Prolog-invented variable names.)

I-11. (for Chapter 5) In 20 words or less, give the declarative meaning of program "f".

f(L,M) :- g(L,[],L2). g([],L,L). g([X|L],L2,L3) :- g(L,[X|L2],L3).

I-12. (for Chapter 6) Consider the rule:

bottom(S) :- bosses(R,S), not(bosses(S,U)).

with the facts:

bosses(poindexter,north). bosses(reagan,poindexter). bosses(north,hall).

Do pure forward chaining. Give every new rule the algorithm creates, and give every new fact it proves, until it runs out of things to do. Use focus-of-attention placement of new facts.

I-13. (for Chapter 6) Consider this database of rules and facts:

a(X) :- b(X), c(X). c(Longvariable) :- f(Longvariable). c(2) :- not(e(Z)). b(Y) :- c(Y), not(d(Y)). e(5). f(3).

(a) What rule rewritings are done and what facts are proved, in order, by pure forward chaining with the focus-of-attention approach? Assume you want all possible facts. Remember, save "not"s for last.

(b) What facts are proved, in order, by rule-cycle hybrid chaining with the focus-of-attention approach? Also give the cycle number on which the fact was proved. Assume you want all possible facts. Remember, save "not"s for last.

I-14 (for Chapter 6). Consider the following rules for diagnosis of childhood illnesses:

R1: If there's a rash and but not a high fever, suspect chicken pox. R2: If there's a rash, a cold symptom, and a mild fever, suspect German measles. R3: If there's a rash, a cold symptom, and a high fever, suspect measles. R4: If there's a mild fever and sore throat, suspect mumps. R5: If there's a medium fever and sore throat, suspect scarlet fever. R6: If there's a cold symptom and any fever, suspect a cold. R7: A high fever is over 102. R8: A medium fever is over 100.5 to 102. R9: A mild fever is 99 to 100.5. R10: Sneezing is a cold symptom. R11: Headache is a cold symptom. R12: Sore throat is a cold symptom.

(a) Write the above as twelve Prolog rules. Use variables where appropriate. (Hint: they are appropriate somewhere.)

(b) Suppose we do backwards chaining on these rules with the goals taken in order to be measles, scarlet fever, chicken pox, mumps, German measles, cold. Suppose the facts in order are a sore throat, a temperature of 101, a desire to go to the circus, and a rash. Suppose we continue until everything provable is proved. What conclusions are reached (including intermediate ones), in order?

(c) Suppose we do pure forward chaining with the same specifications as part (b), using focus-of-attention placement of new facts. Suppose we continue until everything provable is proved. What conclusions (including intermediate) are reached, in order?

(d) Suppose we do rule-cycle hybrid chaining with the same specifications as part (b). Suppose we continue until everything provable is proved. What conclusions (including intermediate) are reached, in order?

(e) What is the advantage of taking the disease conclusions in the order given in part (b)? That is, why that order and not some other?

(f) Would a decision lattice be a good way to compile a set of medical-diagnosis rules like these, a set say of about 1000 rules for common problems? Why?

(g) Would "and" parallelism implemented with multiple processors be a good idea with a set of medical-diagnosis rules like these, assuming the same 1000 common-problem rules as part (f)? Why? Think of how the rule-based would actually be used.

I-15. (for Chapter 6) Suppose you have two processors that access the same Prolog database, both of which can run Prolog independently. Describe all the useful possible concurrencies (including full details of what should be run in parallel with what) in the following code:

p(X) :- q(X), r(X,Y), s(X), t(Y). p(X) :- u(X).

when used with the following query:

?- p(Q).

In particular, explain carefully how maximal and-parallelism could be done.

I-16. (for Chapter 6) Suppose a grammar is written in the manner of section 5.9, as substitutions of lists of words for other lists of words. Substitutions are not "coupled", so two substitutions might make sense individually, but not when used together, at two different places in the same sentence. How could a Prolog implementation apply "semantic" criteria to rule out such nonmeaningful combinations? Give implementation details.

I-17. (for Chapter 6) (a) Which control structure is a decision lattice most similar to, backward chaining or forward chaining? Explain.

(b) Which control structure is an and-or-not lattice most similar to, backward chaining or forward chaining? Explain.

I-18. (for Chapter 7) Study Exercise 7-5 and its answer.

(a) Now suppose the original code we have for testing whether a(X) is true is in the form of M rules, each requiring computation cost of 2H/M. Suppose the rules are tried sequentially until one succeeds, just like cached facts. Suppose each has a mutually exclusive probability Q of success. Redo part (a) of the original problem to get a new formula for when using a cache will be preferable.

(b) Now suppose the rules are not equally costly, but are arranged from least costly to most costly. Suppose further the costs increase linearly so that the Ith rule costs about 4HI/M*M. Redo part (a) of problem 7-5.

I-19. (for Chapter 7) The appliance program (via the "ask" code) caches the answers to all questions it asks the user. Would any additional caching (besides that of final answers, which could prevent repeated printing of the same diagnosis) significantly improve the program?

I-20. (for Chapter 7) Explain how forward chaining must be modified to handle rules with arithmetic computations, where the computations are used only to compute numbers for the left sides of the rules.

I-21. (for Chapter 7) (for interpreted Prologs only) The language Lisp has a "prog" construct, whereby a list of statements are executed in order. Define the equivalent for Prolog. That is, define a predicate "prog" of one argument, a list of predicate expressions, that calls every one of the expressions in order, ignoring whether any expression succeeds or fails. It also should not permit backtracking from expression to expression if something after the "prog" fails. Hint: you can't backtrack into a "not".

I-22. (for Chapter 7) Construct a rule-based system that models an elevator that you have access to. Conduct experiments to find out what the elevator does in particular situations. Then test your program in those same situations to confirm the same behavior. Use as program input information about which buttons were pushed when, where the other elevator is, whether someone is standing in the door, etc. Let the program output be a statement about elevator behavior, either (1) it goes to a particular floor, or (2) its doors open or close, or (3) it stops, or (4) its alarm bell rings, etc.

I-23. (for Chapter 7) Find a misleading advertisement for an expert-system software product. Point out what is misleading. Beyond the hype, what do you each is really selling, in the terminology of this chapter?

I-24. (for Chapter 8) Suppose you're writing an expert system to detect Russian submarines.

--Suppose an expert tells you that if there are Russian manuevers in the area, the probability is 0.5 that any submarine waveform you detect will be a Russian submarine.

--Suppose the expert also says that if you see a type Z37 waveform, it's a Russian submarine with probability 0.8.

(a) Now suppose an expert says that if (i) there are Russian manuevers in the area, and (ii) you think with probability 0.5 that you see a Z37 waveform (that is, you're only half sure that the waveform you see go by on the screen was a Z37), then the probability it's a Russian submarine is 0.7. What evidence combination method is the expert using: independence-assumption, conservative, liberal, or neither? Show your math.

(b) Suppose instead for the same situation as (a) the expert says the probability it is a Russian submarine is 1.0. What evidence combination method is the expert using: independence-assumption, conservative, liberal, or neither? Show your math.

I-25 (for Chapter 8). Consider:

c :- a, b. c :- not(a), not(b). d :- a, not(b). e :- not(a), b.

(a) Suppose now all the above predicate expressions are given probabilities. Suppose further that the second rule has a rule strength of 0.8 and the third rule has a rule strength of 0.5. Write the modified rules containing probabilities.

(b) Suppose b has probability 0.4. Suppose we combine probabilities (including those for rule strengths) with the independence assumption. Conduct computer experiments to graph c, d, and e as a function of the probability of a. Use the representative a-probability values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.

(c) Suppose b has probability 0.9. Suppose we combine probabilities (including those for rule strengths) with the independence assumption. Conduct computer experiments to graph c, d, and e as a function of the probability of a. Use the representative a-probability values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.

(d) Suppose b has probability 0.4. Suppose we combine probabilities (including those for rule strengths) with the conservative assumption. Conduct computer experiments to graph c, d, and e as a function of the probability of a. Use the representative a-probability values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.

(e) Suppose b has probability 0.9. Suppose we combine probabilities (including those for rule strengths) with the conservative assumption. Conduct computer experiments to graph c, d, and e as a function of the probability of a. Use the representative a-probability values 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0.

(f) What major disadvantage of the conservative formulas is shown by the graphs for parts (d) and (e)? Note that being equal to zero a lot is not a disadvantage per se, because maybe the combination value really is zero.

I-26. (for Chapter 9) Suppose in a search you are in a state "a" and you can go to states "b", "c", and "d". Suppose "a" has evaluation 9, "b" 14, "c" 11, and "d" 15. Suppose the cost from "a" to "b" is 10, from "a" to "c" 20, from "a" to "d" 12, from "b" to "c" 15, and from "b" to "d" 8.

(a) What state do you go to next from "a" in a best-first search?

(b) What state do you go to next from "a" in an A* search?

(c) What state do you go to next from "a" in a depth-first search with the heuristic that states are preferred in alphabetical order?

I-27 (for Chapter 9). Consider as search the problem of using inheritance to find a value of a property for some object (Problem 9-2 and its answer may help).

(a) Would bidirectional search (in general) be a good idea?

(b) In which direction(s) would you expect the branching factors to be less?

(c) Give a domain-independent heuristic--that is, one that potentially applies to any inheritance situation, and doesn't reference features of the objects over which the inheritance occurs.

(d) Give a domain-dependent heuristic, one that depends on features of the objects over which inheritance occurs.

(e) Is the search (in general) decomposable? Why?

(f) Is the search (in general) monotonic for the operators "follow a_kind_of link", "follow part_of link", etc.? Why?

I-28. (for Chapter 10) Implement forward chaining as a search problem. Assume there are "rule" and "fact" facts defining the initial rules and facts. Give starting, "successor", and "eval" definitions. Assume no "not"s or other built-in predicates are used in the rules.

I-29. (for Chapter 10) Implement bidirectional best-first search by modifying the best-first program of the chapter. Alternate between forward and backward steps, where each step finds all the successors of a state. Assume you have also a backwards evaluation function, "backeval", and well as the usual "eval".

I-30. (for Chapter 11) Means-ends analysis can also involve numbers. To do this, the means-ends program can be modified to call on a user-defined predicate "transform" after the postconditions have been applied, but before the second recursive call. This "transform" has three arguments: an input operator name, an input state, and an output state representing the result state after necessary arithmetic calculations and adjustments have been done. For instance:

transform(smash_case,L,[value(0)|L2]) :- singlemember(value(X),L), delete(value(X),L,L2).

says that the "smash_case" operator changes the value of the flashlight from X cents to 0 cents; this would be in addition to all the preconditions, deletepostconditions, and addpostconditions shown in chapter 11. (The "singlemember" is a version of "member" that doesn't backtrack, which is already included in the means-ends program file.) To call on "transform", substitute for the "union" line in the means_ends definition the following:

union(Addpostlist,Prestate2,Postlist2), transform(Operator,Postlist2,Postlist),

Suppose you want to fight a fire in a compartment on board a ship, using means-ends analysis. Suppose you have three operators:

"hose": apply water from a hose to the fire for 10 seconds; "drain": open the drain; "wait": wait for the water to drain.

Suppose states can be described by a lists of the facts:

--fire(X): the fire is of degree X, where X is an integer, and larger X means a worse fire --open: the drain is open --closed: the drain is closed --flooded: the compartment is flooded --dry: the compartment is not flooded

And suppose these have as preconditions and postconditions (and nothing else is a precondition or postcondition):

--hose: you can't hose if the compartment is flooded, and you can't hose if the drain isn't open; each "hose" operation reduces the degree of the fire by 2, and floods the compartment.

--drain: you can do this anytime, and the drain stays open once opened.

--wait: you can do this anytime; if the compartment is flooded, it becomes unflooded; and the fire degree increases by 1 if it's not zero.

Assume in your starting state that the fire is degree 3, the drain is closed, and the compartment is dry. Your goal is a dry compartment with the fire out.

I-31. (for Chapter 11) Study problem 10-9. You are to program a computer in Prolog to solve the problem described, but using means-ends analysis (from Chapter 11) rather than A* search. Try to order the operator recommendations, preconditions, and postconditions in a reasonable way. Show your program working.

To simplify the problem:

--change the goal state to be one in which the robot must be in office 1.

--assume the robot can carry more than one basket at a time.

--assume it doesn't make sense to pick up a basket that isn't full.

--assume it doesn't make sense to put down a basket that is full.

--assume that if the robot is holding something, it carries it wherever it goes;

--don't reason about the last action taken in the recommended facts or the precondition facts.

--don't worry about the costs; don't worry if your answer seems a little stupid in doing some unnecessary things.

The choice of operators is important. Probably you'll need separate operators for picking up a trash basket, putting down the basket, carrying a basket somewhere, and disposing of the basket down the chute. Also distinguish carrying the basket from just going between rooms without carrying anything, because the two have different side effects. However, it would be poor programming to have two separate operators (and facts) for going from office 1 to the chute, and from the chute to office 1--except in Turbo Prolog, where you won't be able to use variables with operators (with some exceptions), and you'll have to write out each

possible action as a separate fact.

You don't need any rules for recommendations, preconditions, and postconditions: you should be able to define them with just facts. The order of these facts makes a lot of difference. Also, the order of terms within the lists makes some difference too. Here's an example of an operator definition: the preconditions for carrying the basket that belongs in room X to place Y are that the robot is holding that basket, the robot is in room Z, and the basket is also in room Z. Then the deletepostconditions are that the basket is in room Z and the robot is in room Z. And the addpostconditions are that the basket is in room Y and the robot is in room Y. (The requirement that Y be different from Z can be enforced by the "recommended" facts.)

Since you can't write "negative preconditions" for the means-ends program, you'll need to define some "negative predicates" like "vacuumed" and "emptybasket" that can be used in preconditions.

Warning: be very careful you don't write your own definitions of the predicates difference, append, union, subset, deleteitems, and delete. Otherwise, your program will take forever to run, because of the unnecessary extra backtracking choices.

You'll need some debugging facilities, because the program you'll write will do some complicated things. It's particularly helpful to watch the calls to the "means_ends" predicate, because that tells you the identity of the overall goal the program is working on. One good way to debug is to check whether the arguments to means_ends make sense. Check for duplicate facts; these suggest mistakes in the deletepostconditions. Check for contradictory facts; these suggest mistakes in either the addpostconditions or deletepostconditions. Check for missing facts (for instance, if there are no facts at all about the dustiness of room 1); these suggest mistakes in the addpostconditions. When you've found a problem in the trace, try to find where it first occurs; the action then applied must have faulty definitions.

Another bug you may see is running out of stack space because of an infinite loop. Often this is due to the operator-defining facts being out of order. Other times it may be due to a recommendation or precondition failure of an operator.

I-32. (for Chapter 11) Modify the means-ends program to allow for "negative preconditions", things that must be false before an operator can be applied. These can mean simpler state descriptions. Implement them by facts in the preconditions with "not" in front of them, like saying "not(open(case))" instead of "closed(case)".

I-33. (for Chapter 13) Solve the following Prolog query by relaxation:

?- a(X,Y), a(Y,Z), a(X,Z), a(Z,Y).

given the database:

a(1,4). a(2,4). a(1,5). a(3,3). a(3,6). a(5,4).

I-34. (for Chapter 13) The scheduling program in section 13.1 is still very slow because duplicates aren't caught early enough. Modify the program to fix this.

I-35. (for Miscellaneous problems) (a) Which of the following is true (that is, most false) about writing artificial intelligence applications?

(i) Programs consist of many small pieces. (ii) Programmers implement algorithms. (iii) Programs use

symbols more than numbers. (iv) Programmers spend much time debugging.

(b) Which of the following is true (that is, most false) about Prolog?

(i) It emphasizes "how" instead of "what".

(ii) It's usually interpreted, not compiled.

(iii) It's often good for combinatorial problems.

(iv) It's a relatively new language and can be developed further.

I-36. (for Miscellaneous problems) Linked lists are important in many ways in artificial intelligence. Give an important use of linked lists for each of the following.

(a) relaxation

(b) rule-based expert systems with probabilities

(c) breadth-first search, besides the state representation

(d) automatic backtracking

I-37. (for Miscellaneous problems) Which of the following techniques (not necessarily just one) can get into an infinite loop on some practical problems having solutions? For each answer "yes", explain how in a few words. (Assume there are no infinite processing loops in any problem-dependent code.)

(a) depth-first search with heuristics and a finite branching factor

(b) A* search with an always-positive cost function and a finite branching factor

(c) means-ends analysis

(d) rule-cycle hybrid chaining on rules without arguments

(e) inheritance of a property with respect to "part_of", for a finite set of facts

(f) inheritance of a property with respect to "is_another_name_for", for a finite set of facts

(g) proving something by resolution on clauses without arguments

I-38. (for Miscellaneous problems) Consider the data-flow diagram on the next page. The circles are data objects, and the rectangles are processes operating on the data objects.

.PA .PA

(a) Represent the entire diagram as a set of Prolog facts in the form

c(<input-object>,<process>,<output-object>).

where "c" stands for "connection". Some processes take two inputs; represent these with two facts. (Remember, "<" and ">" are not Prolog symbols, but a way to describe something.)

(b) Suppose that for some objects and some processes, we have additional facts of the form

defective(<object>). faulty(<process>).

An object is defective if it is the output of a process that is faulty. An object is also defective if it is the output of a process for which at least one input is defective. Write these as two Prolog inference rules that use the "c" facts. ("Defective" and "faulty" don't necessarily mean all the time; that is, the object or process could be correct part of the time. But you don't need to use probabilities in your rules.)

(c) Suppose process p is faulty; that is, we have the fact

faulty(p).

Explain how a Prolog interpreter would prove that object d is defective, using the facts from part (a) and the rules from part (b), and no additional rules. Give each step that the interpreter would take, including every variable binding. And give the starting query.

(d) Write the rules of part (b) in clause form.

(e) Prove by resolution that if object f is not defective, then object c is not defective either. Use the clauses from (d) and the facts from (a). (Warning: don't ever resolve clauses with variables in common; rename the variables first.)

(f) Suppose that in some data-flow diagram (not necessarily that in parts (a)-(e)) the objects X and Y are defective. Consider the problem of determining whether defective object X could explain defective object Y; that is, whether there's some path from X to Y. This can be considered a search problem in which the states are objects, the starting state is X, and the goal state is Y. If the branching factor is the same in both directions, would bidirectional search be a good idea? Why?

(g) Assume we are searching left-to-right with the diagram considered in parts (a) through (e). Assume we want to find a connection between an X and Y picked randomly, with a pick of any object in the diagram equally likely. What is the average branching factor for states (including state f)?

(h) Suppose we want a Prolog program to do this search. To use the search programs in the book, we must define a "successor" predicate of two arguments, an input state and an immediate successor output state. Write its Prolog definition, using the "c" predicate of part (a).

(i) Now suppose we want to find all the defective objects and faulty processes, given a partial set of "defective" and "faulty" facts and the inference rules of part (b). Explain how relaxation could do this. What would be the variables? What would be the values? What would be the constraints?

(j) Suppose we know many facts about the processes (rectangles) in a data-flow diagram; then frames could represent processes. Consider the superframe (generalization frame) representing all occurrences of the "q" process in a data-flow diagram. Give examples (one each) of value and slot inheritance from this frame to a more specialized frame.

(k) Frames could represent sections of a data-flow diagram; these would be "part_of" superframes. For instance, the example diagram for part (a) could be considered one big process with "a" as input and "f" as output. This leads to hierarchical data-flow diagrams. Suppose for some process in a diagram there was an multiple-inheritance conflict between the "core-memory-required" value inherited from this "part_of" superframe and the "core-memory-required" value inherited from the generalization superframe of part (j). How would you resolve the conflict in a general and efficient way?

(l) Explain how to get a helpful lower-bound evaluation function for the search problem described in part (d), one that will work for any X and Y on any diagram. Assume the diagram is described by Prolog facts like those in part (a). To make your evaluation function work, you can store extra facts with each node of the diagram (but you can't select the facts based on knowledge of X and Y--the same facts must be used for every X and Y). Your evaluation function shouldn't actually solve the problem because then it would be a solution method, not an evaluation function.

.PA

ANSWERS TO MORE TEXT PROBLEMS

(This, and Appendix G in the book, cover the answers to nearly all the problems. The only exceptions are the problems that require very long answers.)

2-3. See the next page.

2-4. One way:

circle(c1). circle(c2). circle(c3). circle(c4). big(c1). small(c2). small(c3). small(c4). inside_of(c2,c1). right_of(c3,c1). above(c4,c3). touching(c1,c3). touching(c3,c4). same_center(c2,c1).

Note it's important to specify the type of c1, c2, c3, and c4 as circles, and important to specify "touching" relationships in addition to "right_of" relationships. Additional redundant facts could also be included beyond the preceding.

.PA .PA

2-8. Facts about the crime and the nature of the crime, how it was committed, and what objects were involved in it. These things don't require any common arguments except perhaps a specific code distinguishing the specific crime. Reports from witnesses can also include this code, but they will also all require two additional arguments: the name of the witness, and the time the specific observation was made. It's important to record the time, because the same action may occur more than once, and it's important to record the name of the witnesses because some witnesses lie, and their reports cannot be taken as facts. (By "time" we mean a unique identification for every unique time, so this may require a date as well as hours, minutes and seconds.)

2-9. Both can be expressed with type predicates, with predicate name "mayor". But "Clint" is a man's name and must represent a specific entity; "someone" is an English pronoun that can stand for any person. In fact, "someone" doesn't mean anything here because any "mayor" is a person. So "someone" is like a placeholder or "variable", to which a word can later be assigned. In particular, it's like an existentially quantified variable (see Appendix A). In Prolog, existentially quantified facts like that cannot be represented directly.

2-10. In the first sentence, "boss" indicates a relationship-predicate fact about two people; "boss" is like the predicate name. In the second sentence, it represents an action of ordering people around, and could be either a predicate name or an argument. (A "boss" does not necessarily "boss" people around, and not only "bosses" boss people around.) In the third sentence it refers to a symbol in English used to represent a word, so it's a statement about the symbols in a language; the word would be represented as an argument, not a predicate name. In the fourth sentence it refers to an abstract concept of being a boss, something related to but distinct from the bossing relationship in the first sentence; and the concept would be an argument, not a predicate. These four meanings are so different it would be poor judgment to use the same word for any of them in a computer representation, in line with the suggestion #3 in section 2.4 to avoid names with multiple meanings.

2-11. (a) It's more "natural", more like how people talk. So syntax errors are less likely than with predicates.

(b) There are many different ways of expressing things in English, and you must handle all of them.

(c) English is a lot more ambiguous than predicates. The word "is" is used in many different ways, as well as many other words like "the" and "at". So to make such an approach work, systems have to restrict the English input from the user. These restrictions are usually hard to remember and easy to get confused about.

3-3. The answers to query 2 are a subset or the same as the answers to query 1. Query 2 just "filters out" some of the answers to Query 1 by its additional predicates.

3-4. "The" signals a reference to something used before, unlike "a" and "an"; "the memo" refers to the previous "a memo", and "the meeting" to the previous "a meeting". So it's like the way variables are handled in a query, the "generate-and-test" approach: "the" corresponds to a bound variable, and "a" to an unbound variable.

3-7. (a)

married(jason,phoebe). married(phoebe,jason). married(perry,stacey). married(stacey,perry). married(eulalie,lane). married(lane,eulalie). loves(phoebe,perry). loves(zack,phoebe). loves(lane,phoebe). loves(jason,eulalie).

(b) Jason and Phoebe only, the only A and B values found below:

?- married(A,B), loves(A,C), loves(B,D), not(loves(A,B)), not(loves(B,A)). A=jason B=phoebe C=eulalie D=perry ;

A=phoebe B=jason C=perry D=eulalie ;

no

(c) The jealous people are Phoebe, Zack, Lane, Jason, and Stacey. They are the possible A values in the following:

?- (loves(A,B); married(A,B)), loves(C,B).

A=phoebe B=perry C=phoebe ;

A=zack B=phoebe C=zack ;

A=zack B=phoebe C=lane ;

A=lane B=phoebe C=zack ;

A=lane B=phoebe C=lane

A=jason B=eulalie C=jason ;

A=jason B=phoebe C=zack ;

A=jason B=phoebe C=lane ;

A=stacey B=perry C=phoebe ;

A=lane B=eulalie C=jason ;

no

3-8.(a) The query asks for "a" facts whose arguments match those of "b" facts. So there are two answers: X=3 and Y=2, and X=4 and Y=4.

(b) This query imposes an additional restriction on the solutions to part (a); X=4 and Y=4 is the only possibility that satisfies the additional predicate expression.

3-10.(a) Twice: once for X=a and Y=b, and once for X=b and Y=b. The binding X=c and Y=d succeeds.

(b) Three times: once for X=a, Y=b, and Z=b; once for X=b, Y=b, and Z=b; and once for X=c, Y=d, and Z=a.

(c) None--Prolog interpreters don't backtrack to the last query predicate unless a semicolon is typed.

(d) Once: for the last binding X=c and Y=d. Then the binding X=c and Y=a is found and it succeeds.

(e) Once: for the last binding X=c, Y=d, and Z=a.

3-11. One way:

subdepartment(<department>,<subdepartment>). project(<project-name>,<project-code>, <starting-date>, <completion-date>,<subdepartments>). employee(<employee-name>,<ss-number>,<birth-date>, <home-address>,<office>,<job-skills>). employee_project_match(<ss-number>,<project-code>).

Note that since usually a project has only one subdepartment, we might as well make that attribute an argument in the "project" facts; but since the number of employees in a project could vary widely, the association of employees to projects should be handled by separate "employee_project_match" facts in which both the same employee and the same project can be mentioned multiple times. Note that job skills can go in the "employee" facts since they can efficiently be stored by a bit array, giving a fixed record length for all employees.

3-12. (a) The computer took the question too literally; it should have realized that what the person really

wanted to know was the name of that commander and his rank, not whether it had the capability of finding out those things. So a natural language front end must have expectations about what it will be called upon to do, and it must live up to those expectations rather than answering a question literally.

(b) Again the computer is taking the user's questions too literally, but now the bug is more subtle because the answers given are correct for sailboats and other small ships that don't have recorded locations and commanders. In order to answer the question, the computer discovered that the Pequod does not exist, and then it concluded that there are never any commanders for ships that do not exist. But that first conclusion is much more interesting than the second, so the computer should have given it instead. That feature of computer behavior is called being "cooperative".

4-1. For one thing, the ":-" has a procedural meaning, not just a declarative one like logical implication. For another, the ":-" also includes a causal relation between the thing implying and the thing implied. A logical implication has no such causal meaning, and can in fact be equally well used in a backwards or "contrapositive" way, whereby if A implies B, then not(B) implies not(A). Chapter 14 discusses this more.

4-3. One way:

father(X,Y) :- child(X,M,Y,S).

mother(X,Y) :- child(F,X,Y,S).

son(X,Y) :- child(Y,M,X,male). son(X,Y) :- child(F,Y,X,male).

grandfather(X,Y) :- father(X,Z), parent(Z,Y). parent(X,Y) :- father(X,Y). parent(X,Y) :- mother(X,Y).

sister(X,Y) :- child(F,M,X,female), child(F,M,Y,S), not(same(X,Y)). same(X,X).

uncle(X,Y) :- parent(P,Y), brother(X,P). uncle(X,Y) :- parent(P,Y), sister(Z,P), married(Z,X). brother(X,Y) :- child(F,M,X,male), child(F,M,Y,S), not(same(X,Y)). married(X,Y) :- child(X,Y,C,S). married(X,Y) :- child(Y,X,C,S).

ancestor(X,Y) :- parent(X,Y). ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

halfsister(X,Y) :- child(F,M,X,female), parent(P,X), parent(P,Y), not(sister(X,Y)), not(same(X,Y)).

4-4. Code:

on(b,a). on(c,b). on(d,c). on(f,e). on(g,f). on(i,h). left(a,e). left(e,h). above(X,Y) :- on(X,Y). above(X,Y) :- on(X,Z), above(Z,Y). stackleft(X,Y) :- left(X,Y). stackleft(X,Y) :- left(X,W), above(Y,W). stackleft(X,Y) :- left(Z,Y), above(Z,X). stackleft(X,Y) :- left(Z,W), above(Z,X), above(Y,W).

Run:

?- on(X,b).

X=c ;

no ?- on(a,X).

```
no ?- on(X,Y).

X=b Y=a ;

X=c Y=b ;

X=d Y=c ;

X=f Y=e ;

X=g Y=f ;

X=i Y=h ;

no ?- on(X,Y),left(Y,Z).

X=b Y=a Z=e ;

X=f Y=e Z=h ;

no ?- above(X,Y).

X=b Y=a ;

X=c Y=b ;

X=d Y=c ;

X=f Y=e ;

X=g Y=f ;

X=i Y=h ;

X=c Y=a ;

X=d Y=b ;

X=d Y=a ;

X=g Y=e ;

no ?- above(X,f); (left(Y,Z),above(X,Y),above(f,Z)).

X=g Y=_5 Z=_6 ;

X=b Y=a Z=e ;

X=c Y=a Z=e ;
```

X=d Y=a Z=e ;

no ?- above(X,Y), not(left(X,Z)).

X=b Y=a Z=_6 ;

X=c Y=b Z=_6 ;

X=d Y=c Z=_6 ;

X=f Y=e Z=_6 ;

X=g Y=f Z=_6 ;

X=i Y=h Z=_6 ;

X=c Y=a Z=_6 ;

X=d Y=b Z=_6 ;

X=d Y=a Z=_6 ;

X=g Y=e Z=_6 ;

no

4-8. X=3, Y=3. X=1 and Y=2 can't match either of the "b" facts or the "b" rule. X=3 and Y=5 can't match the "b" facts, and matching the "b" rule would require that "b(2,5)" be provable, and it isn't. So the "a" rule must be used if there is to be any answer to the query. R=1 and S=3 won't work because there isn't any "b(2,1)" fact; R=2 and S=3 won't work because there isn't any "b(2,2)" fact. So the "b" rule must be used to get an R and an S. But on its right side, T can be bound to 3 and match the two "b" facts. So "b(3,3) is proved, and hence "a(3,3)" is proved.

4-9. For the relationship "part_of", the properties "author", "language_written_in", and "source_institution" inherit downwards (that is, from the program to the subprogram); the property "has_bugs" inherits upwards. For the relationship "a_kind_of", the properties "has_applications_to", "requires_128K_machine", and "Company_X_sells_a_program_of_this type" inherit upwards (from more specific program to more general program).

4-12. (a) Upwards. Properties involving existential quantification usually inherit upwards.

(b) Neither direction. Statistical properties usually don't inherit either way.

(c) Downwards. Properties involving universal quantification usually inherit downwards.

(d) Downwards. Another universal quantification.

4-14. (a) 0--it could be a fact. (Or maybe 1 if you read the question as meaning the second transitivity rule must be used once.)

(b) Twenty-nine, when the objects all form a single chain. Note that the number of connections in a chain of number N items is N-1, and we can leap the final gap with a fact instead of a rule--so that's N-2 uses of the second transitivity rule.

(c) A balanced binary tree of height 4.

(d) 3, the height of the tree minus one, (since we can leap the last gap with a fact).

(e) Once, since the only recursion necessary is done by the transitive predicate "a_kind_of" and not in the inheritance rule itself.

4-15.(a) Note that Tom's Rabbit bears the same relationship to all Rabbits as all Rabbits do to all VWs: the first thing is a kind of the second thing. Similarly, an alternator bears the same relationship to an electrical system as an electrical system does to a VW: the first thing is part of the second thing. Note also that three English phrases represent single concepts and thus can each map to a single word in Prolog: VW Rabbit, Tom's car, and Dick's car; we use the underscore character (_) in the Prolog words in these situations. So we get (noting that no characters may be capitalized):

a_kind_of(vw_rabbit,vw). a_kind_of(toms_car,vw_rabbit). a_kind_of(dicks_car,vw_rabbit). part_of(electrical_system,vw). part_of(alternator,electrical_system). defective(alternator).

The last fact does not need to be qualified as to the kind of car since we're only talking about VW's here. To use artificial intelligence terminology, VWs are the universe of discourse. You also might want to include facts of ownership, and the statement that the alternator is on the VW, though you don't need them for parts (b) and (c):

owns(tom,vw_rabbit). owns(dick,vw_rabbit). on(alternator,vw).

Note that our predicates are quite general--this is the usual way that artificial intelligence knowledge representation is done. The specific things mentioned like VW Rabbits, Tom's car, and electrical systems should be arguments to predicates, not the names of predicates. Then we don't have to write a lot of different inferences rules for all our predicate names. So all the following would be poor knowledge representation:

vw(vw_rabbit). vw_rabbit(tom). vw_part_of(electrical_system). has_electrical_system(vw). alternator_defective(vw).

Also note that nothing is asserted about VWs being cars or VW Rabbits being cars--only that Tom has a car and Dick has a car.

(b) You need to say that a_kind_of and part_of are transitive:

a_kind_of(A,C) :- a_kind_of(A,B), a_kind_of(B,C). part_of(D,F) :- part_of(D,E), part_of(E,F).

and also the defectiveness inherits with respect to them. That is, that something is defective if it is a member of a class of things that are defective, or if any part of it is defective:

defective(G) :- a_kind_of(G,H), defective(H). defective(I) :- part_of(J,I), defective(J).

Note the reversed order of arguments in the second rule--defectiveness inherits upwards for part_of, and

downwards for a_kind_of.

Alternatively, you could use the on(alternator,vw) information with this rule:

part_of(K,L) :- on(K,L)

and eliminate the second rule above. But this is controversial--something could be "on" a car, like a ski rack, and not be part of the car.

(c) The goal is to prove defective(dicks_car). By the fourth rule above, it is sufficient to show the some part of Dick's car is defective. Using the part_of transitivity rule (the second rule above) with D=alternator, E=vw, F=electrical_system, or the last rule above, we see that the alternator is part of the VW. Since all alternators are defective, it follows all VWs are defective. Now using the first rule with A=dicks_car, B=vw, C=vw_rabbit, we see that Dick's car is a VW. Hence by the third rule, Dick's car is defective.

4-16. One Prolog database:

limit(15) :- near_railroad, railroad_view_blocked, not(railroad_warning). railroad_warning :- railroad_gates. railroad_warning :- railroad_signal. railroad_warning :- flagman. limit(15) :- near_intersection, intersection_view_blocked, not(right_of_way). right_of_way :- cross_street_yield. right_of_way :- cross_street_stop. limit(15) :- in_alley. limit(X) :- sign(limit,X), X<25. limit(25) :- sign(school), children_danger. children_danger :- children_present. children_danger :- unfenced-schoolyard. limit(X) :- sign(limit,X), X<56. limit(25) :- district(residential). limit(55).

First database:

sign(limit,55). district(residential).

Second database:

sign(school). children_present. sign(limit,55). district(residential).

Third database:

sign(limit,35). in_alley. sign(school). children_present. sign(limit,55). district(residential).

With the first situation:

?- limit(X).

X=55

yes

With the second situation:

?- limit(X).

X=25

yes

With the third situation:

?- limit(X).

X=15

yes

4-17. Legal definitions are much more flexible, because they deal with generally much more abstract notions. It's not a weakness of either legal definitions or Prolog it's just that they deal with different things. It probably wouldn't be a good idea to make legal definitions more precise, because then they couldn't be applied to all the cases they need to be applied to. It probably wouldn't be a good idea to make Prolog less precise because precision is why people use computers in the first place.

4-18. Often the terms discussed are not clearly defined, and mean different things to different people. So people are mapping the same sentences into different internal representations. And because such terms are often abstract, it's hard for people to understand the different meanings that each is assigning.

4-19. (a) A piece of the rock.

(b) Chemical decomposition of the rock into silicon and oxygen molecules.

4-20. This is the famous "barber" paradox of Bertrand Russell. (a) saves(X,God) :- not(saves(X,X)).

(b) The rule becomes: saves(God,God) :- not(saves(God,God)).

(c) No, the proper conclusion is that the original rule is faulty. Just because we can write something with Prolog notation doesn't make it true.

5-3.

better_divide(X,Y,Q) :- not(var(X)), not(var(Y)), not(var(Q)), not(Y=0), Z is X / Y, Z=Q. better_divide(X,Y,Q) :- not(var(X)), not(var(Y)), var(Q), not(Y=0), Q is X / Y. better_divide(X,Y,Q) :- not(var(X)), var(Y), not(var(Q)), Y is Q * X. better_divide(X,Y,Q) :- var(X), not(var(Y)), not(var(Q)), X is Q * Y.

5-4. Use "better_add" and:

tax(G,D,R,T) :- better_add(D,N,G), better_multiply(N,R,T). tax(G,D,R,T) :- better_multiply(N,R,T), better_add(D,N,G). better_multiply(X,Y,S) :- not(var(X)), not(var(Y)), not(var(S)), Z is X * Y, Z=S. better_multiply(X,Y,S) :- not(var(X)), not(var(Y)), var(S), S is X * Y. better_multiply(X,Y,S) :- not(var(X)), var(Y), not(var(S)), not(X=0), Y is S / X. better_multiply(X,Y,S) :- var(X), not(var(Y)), not(var(S)), not(Y=0), X is S / Y.

Notice no "var" predicate expressions are needed in the "tax" rules; if more than one argument to "tax" is unbound, both "better_add" and "better_multiply" will fail.

5-5.

square(X,Y) :- not(var(X)), not(var(Y)), Z is X*X, Z=Y. square(X,Y) :- not(var(X)), var(Y), Y is X*X. square(X,Y) :- var(X), not(var(Y)), square_bisection(X,Y,0,Y). square(X,Y) :- var(X), var(Y), generate_pair(X,Y).

square_bisection(X,Y,Lo,Hi) :- X is (Lo+Hi)/2.0, square(X,S), close(S,Y). square_bisection(X,Y,Lo,Hi) :- Midpoint is (Lo+Hi)/2.0, square(Midpoint,S), S<Y, square_bisection(X,Y,Midpoint,Hi). square_bisection(X,Y,Lo,Hi) :- Midpoint is (Lo+Hi)/2.0, square(Midpoint,S), S>=Y, square_bisection(X,Y,Lo,Midpoint).

close(X,Y) :- D is X-Y, D > -0.001, D < 0.001.

generate_pair(1,1). generate_pair(X,Y) :- generate_pair(A,B), X is A+1, Y is (A+1)*(A+1).

5-6. (a) Use the four lines defining "better_add", but change the name to "integer_better_add" and add one additional line:

integer_better_add(X,Y,S) :- var(X), var(Y), not(var(S)), integer_in_range(X,1,S), Y is S - X.

where "integer_in_range" is defined as in the chapter.

(b) Let S be the characteristic sum for the magic square, and let the letters of the alphabet starting with A represent the cells in the magic square in row-major order. Then:

magic_square(S,A,B,C,D,E,F,G,H,I) :- add_3(A,B,C,S), add_3(A,D,G,S), add_3(D,E,F,S), add_3(B,E,H,S), add_3(G,H,I,S), add_3(C,F,I,S), add_3(A,E,I,S), add_3(C,E,G,S). add_3(X,Y,Z,S) :- integer_better_add(X,W,S), integer_better_add(Y,Z,W).

Note that the order of the expressions in the first rule is very important to the efficiency.

5-7. (a) X=3, Y=2. Notice the query contains three terms "and"ed together. The first two predicate-a facts fail to satisfy the second condition in the query because c(0) is a fact. The third a fact fails because of the d condition. So we must use the rule for a. The first condition matches with P=3 and Q=1, but the second condition fails. So P=2 and Q=1 is tried and a(3,2) succeeds. The "not" condition and the d(3,2) succeed so the whole query succeeds, and X=3 and Y= 2.

(b) X=2, Y=3. Obtained by the next possible match for b(P,Q), that of P=1 and Q=2. This succeeds with the other two query terms similarly.

5-9.(a) speed(<gear_number>,<speed_in_rpm>). Positive speeds can be clockwise, negative counterclockwise.

(b) speed(G,S) :- same_shaft(G,G2), speed(G2,S).

(c) speed(G,S) :- meshed(G,G2), teeth(G,TG), teeth(G2,TG2), speed(G2,S2), S is 0 - ( S2 * TG2 / TG ).

(d) Use the following database:

speed(g1,5000). same_shaft(g2,g1). meshed(g3,g2). meshed(g4,g2). teeth(g1,100). teeth(g2,30). teeth(g3,60). teeth(g4,90). speed(G,S) :- same_shaft(G,G2), speed(G2,S). speed(G,S) :- meshed(G,G2), teeth(G,TG), teeth(G2,TG2), speed(G2,S2), S is 0 - ( S2 * TG2 / TG ).

and issue this query:

?- speed(g4,S).

The one speed fact doesn't apply, so the first speed rule is tried. But there is no same_shaft fact with g4 as first argument, so the rule fails. Now the second speed rule is tried. G2 can be matched to g2, with TG=90 and TG2=30. Then we must find the speed of G2=g2. No fact applies, but we can use the first speed rule: g2 is the first argument to a same_shaft fact having g1 as second argument, so its speed is the speed of g1. And a fact says the speed of g1 is 5000. So the speed of g2 is 5000, and we can perform the calculation in the second speed rule as

S = 0 - ( 5000 * 30 / 90 ) = -1667 rpm

and the variable S in the original query is bound to that number.

(e) You could get infinite loops if you had extra redundant facts, like if you had both meshed(g2,g1) and meshed(g1,g2). You could also get infinite loops if you had more than one inference rule of each of the two above types, as if you had both

speed(G,S) :- same_shaft(G,G2), speed(G2,S). speed(G,S) :- same_shaft(G2,G), speed(G2,S).

(f) The gears wouldn't turn--such a contradiction of rotation speeds is an impossibility.

5-10. It's easiest to just convert everything to meters, since you must worry about infinite loops otherwise.

convert(M1,U1,M2,U2) :- conversion_factor(meters,U1,F1), conversion_factor(meters,U2,F2), M2 is M1*F2/F1.

conversion_factor(meters,meters,1). conversion_factor(meters,decimeters,10). conversion_factor(meters,centimeters,100). conversion_factor(meters,millimeters,1000). conversion_factor(meters,decameters,0.1). conversion_factor(meters,kilometers,0.001). conversion_factor(meters,inches,39.37). conversion_factor(meters,feet,3.281). conversion_factor(meters,yards,1.094). conversion_factor(meters,miles,0.0006214).

?- convert(2,feet,X,meters).

X=0.609571

yes ?- convert(2,feet,X,inches).

X=23.9988

yes ?- convert(20,kilometers,X,miles).

X=12.428

yes ?- convert(42.1,yards,X,decameters).

X=3.84826

yes

5-11.(a)

route(Town1,Town2,Routename,County,Distance), stateof(County,State), countryof(State,Country), limit(State,Limit), limit(Country,Limit).

(b)

limit(R,L) :- route(T1,T2,R,C,D), stateof(C,S),limit(S,L). limit(R,L) :- route(T1,T2,R,C,D), stateof(C,S),countryof(S,K), limit(K,L).

(c)

distance(T1,T2,D) :- route(T1,T2,R,C,D). distance(T1,T2,D) :- route(T1,T3,R,C,D3), distance(T3,T2,D4), D is D3+D4.

(d) r+c facts

(e) infinity, (if routes car contain same town many times). Otherwise, between 0 and t*t - t - r.

Alternative: assign county to town

(a) route(Town1,Town2,Routename,Dist), countyof(Town,County), stateof(County,State), countryof(State,Country), limit(CountyorState,Limit)

(b) limit(R,L) :- route(T1,T2,R,D), countyof(T1,C), countyof(T2,C), stateof(C,S), limit2(S,L). limit2(S,L) :- limit(S,L). limit2(S,L) :- countryof(S,K),limit(K,L).

(c) distance(T1,T2,D) :- route(Tl,T2,R,D). distance(T1,T2,D) :- route(T2,T1,R,D). distance(T1,T2,D) :- route(T1,T3,R13,D13), distance(T3,T2,D32), D is D13 + D32.

5-12. Use:

inference_distance(X,X,0). inference_distance(X,Y,N) :- a_kind_of(X,Z), inference_distance(Z,Y,N2), better_add(N2,1,N).

where "better_add" is defined in the chapter. For a demonstration, assume we have these facts in our database:

a_kind_of(a,c). a_kind_of(b,c). a_kind_of(c,d). a_kind_of(c,e).

Here is a demonstration, showing the testing of all eight cases. (It's important to test all of them, because even a simple Prolog program can surprise you.)

?- inference_distance(a,c,1).

yes ?- inference_distance(a,c,N).

N=1 ;

no ?- inference_distance(a,Y,2).

Y=d ;

Y=e ;

no ?- inference_distance(X,d,2).

X=a ;

X=b ;

no

?- inference_distance(X,Y,2).

X=a Y=d ;

X=a Y=e ;

X=b Y=d ;

X=b Y=e ;

no ?- inference_distance(X,c,N).

X=c N=0 ;

X=a N=1 ;

X=b N=1 ;

no ?-inference_distance(a,Y,N).

Y=a N=0 ;

Y=c N=1 ;

Y=d N=2 ;

Y=e N=2 ;

no ?- inference_distance(X,Y,N).

X=_0 Y=_0 N=0 ;

X=a Y=c N=1 ;

X=a Y=d N=2 ;

X=a Y=e N=2 ;

X=b Y=c N=1 ;

X=b Y=d N=2 ;

X=b Y=e N=2 ;

X=c Y=d N=1 ;

X=c Y=e N=1 ;

no

5-13. The function cannot be a strong homomorphism. That is, it cannot map a lot different situations onto a small set of values. If so, odds are it is difficult, or perhaps even impossible, to fill in other arguments given the value of the last argument and perhaps a few others. Notice that much of arithmetic involves strong homomorphisms: there are many ways to add numbers to get the same sum or the same product. So it is understandable that the Prolog "is" usually used with arithmetic isn't reversible.

5-15. There are two different ways the Prolog interpreter can recurse in in this situation, on lists of increasing size for its first argument and on lists of increasing size for its second argument. It can only recurse one way at a time. So it picks the second way and only generates appending situations where the first argument is the empty list.

5-16. (a) It finds an item, if any, that it can delete from the second-argument list to create the third-argument list.

(b) It will delete only the first occurrence of the first argument from the second-argument list.

5-17. (a) It replaces every other word in a list with the word "censored", binding the result to the second argument. The rule (third line) just says if you can find an X and Y at the front of your list first argument, change the Y to the word "censored" after recursively doing the same on the rest of the list.

The easiest way to see this is figure out what happens with simple example lists:

?- mystery([],Z). Z=[] ?- mystery([a],Z). Z=[a] ?- mystery([a,b],Z). Z=[a,censored]

For the last example the third rule applies for the first time. So X is matched to a, Y is matched to b, and L is matched to [], the empty list. The rule involves a recursive call with first argument L, but L is the empty list, it's the same situation as the first example above, and M is bound to the empty list too. So the rule says to take M and X and the word "censored" and assemble them into the result, the list whose first two items are X and "censored", and whose remainder is M. But M is empty, so the result is [a,censored].

Now consider the next hardest case, an example list with three items:

?- mystery([a,b,c],Z).

The third rule must be again used. It matches X to a, Y to b, and L to [c], the list containing only c. The rule says to do a recursion with [c] the first argument. But to solve that the second rule will do, so M is bound to [c]. So the answer is assembled from X, the word "censored", and that M, and the result for Z is [a,censored,c].

Now consider an example list with four items:

?- mystery([a,b,c,d],L).

In the third rule, X is matched to a, Y is matched to b, and [c,d] is matched to L. The recursion applies mystery to the two-item list [c,d], which by analogy to the above two-item example gives [c,censored]. So we assembled the answer from X, the word "censored", and the list [c,censored], and the result for Z is [a,censored,c,censored]. So it looks like the program changes every alternate word in a list to the word "censored".

(b) One for even-length lists, one for odd-length. The rule lops off two items from the list on each recursion, so you can get into two different final situations.

5-18. See the next page.

.PA .PA

5-19. (a) N + 1 times, once for every item in the list plus once for the empty list; all these invocations will fail.

(b) (N-1)/2. There are 0 calls if the item is first in the list, one call if the item is second in the list, two calls if the item is third in the list, and so on. These terms form an arithmetic series. The average value of an arithmetic series is the average of the first and last numbers, in this case 0 and N - 1.

5-20. Write substitution rules with variables instead of substitution facts. The right sides of the rules can require that the variable bindings be drawn from lists of appropriate words, using the "member" predicate. For example:

substitution([X,transition],[X,change]) :-

member(X,[will,must,should,can,might,perhaps,always,never]).

5-21. The basis condition is wrong. The predicate only makes sense when you can compare horses in a set. For that, you need a basis step concerning at least two horses. But obviously two horses aren't necessarily the same color. Moral: even though basis conditions are usually simple, don't slight them in writing recursive programs: they can have some subtle bugs.

5-22. Use a list to keep arguments previously used, and just check to make sure a new value found is not in the list. At every recursion, add the new argument to the list. So:

```
a3(X,Y) :- a2(X,Y,[]). a2(X,Y,L) :- a(X,Y). a2(X,Y,L) :- not(member(X,L)), a(X,Z), a2(Z,Y,[X|L]).
member(X,[X|L]). member(X,[Y|L]) :- not(X=Y), member(X,L).
```

Then you query a3 to compute relationships by transitivity. For instance, suppose you have the database:

```
a(r,s). a(s,t). a(t,r).
```

(For instance, predicate "a" might mean "equals" or "is near".) Then the query

```
?- a(r,t).
```

will succeed as it should, as will

```
?- a(r,r).
```

But the query

```
?- a(r,u).
```

will fail as it should.

5-23. First define:

```
alldifferent([]). alldifferent([X|L]) :- not(member(X,L)), alldifferent(L).
```

Then define rhymes by facts with a list argument, for instance:

```
rhymes([ooga,booga,dooga,googa]). rhymes([itty,mitty,litty,sitty,ritty]).
rhymes([drallak,pallak,thrallak,chrallak]). rhymes([morade,brocade,hodade,volade]).
rhymes([fufu,rufu,nufu,pufu]). onerhymes([bo,go,toe,joe,moe]).
```

To pick combinations of two or three rhyming words, we need:

```
rhymes2(X,Y) :- rhymes(L), member(X,L), member(Y,L), alldifferent([X,Y]). rhymes2(X,Y,Z) :- rhymes(L),
member(X,L), member(Y,L), member(Z,L), alldifferent([X,Y,Z]). onerhymes2(X,Y,Z) :- onerhymes(L),
member(X,L), member(Y,L), member(Z,L), alldifferent([X,Y,Z]).
```

Finally we must define a predicate that puts the poem together:

```
poem([[A1,A2,B1],[C1,C2,B2],[D1,D2],[D3,D4],[E1,E2,B3]]) :- rhymes2(D1,D2), rhymes2(D3,D4),
alldifferent([D1,D2,D3,D4]), onerhymes2(B1,B2,B3), rhymes2(A1,A2), alldifferent([A1,A2,D1,D2,D3,D4]),
rhymes2(C1,C2), alldifferent([C1,C2,A1,A2,D1,D2,D3,D4]), rhymes2(E1,E2),
alldifferent([E1,E2,C1,C2,A1,A2,D1,D2,D3,D4]).
```

Then to write a poem we just query

```
?- poem(P).
```

6-2. The first fact data(3,0,1) can match the first predicate on the right side of the second rule, with A=3 and

B=1, giving a new rule:

bottom(3,1,7,D) :- data(3,D,1).

But the same fact can also match this new rule too with D=0. So we can prove the fact bottom(3,1,7,0). We have exhausted the first fact, so we now consider this new fact. It can match the right side of the "top" rule, so a new fact top(0,7,3) is proved. We now consider the only remaining fact, data(3,2,1), the second of the original two facts. This can match the original "bottom" rule, giving:

bottom(A,B,7,2) :- data(A,0,B).

But data(3,2,1) can also match the first-listed rule above, giving a new fact bottom(3,1,7,2). This then matches the right side of the "top" rule giving the final new fact top(2,7,3).

6-3.(a) M*min(F,S) matchings. The number of facts bounds the number of predicate names involved, but so does S, so we must take the minimum.

(b) L facts. Even if F=1 there exist rule-based systems for which every conclusion can be reached.

(c) T matchings. If the right side predicates all have the same predicate name, they could all match a single fact.

(d) If all predicates are single-argument, each of the L distinct right sides could be bound in F+T-1 ways, if all F facts and T right-side expressions have the same predicate name, and there are T-1 distinct constants on right sides, giving L*(F+T-1) maximum conclusions.

6-4.(a) d(5), c(2), c(5), b(5), a(5). You cycle through the rules in order. The third rule succeeds first, then the fourth rule (which can be used in two different ways). Returning to the top of the list of rules on the second cycle, the second rule can now succeed. Then on the third cycle the first rule can succeed.

(b) c(2), c(5), d(5), b(5), a(5). The first fact is g(2) which can match the right side of the last rule, so c(2) is proved. This implies the new rule b(2) :- d(2). The second fact is f(5) which can match part of the third rule right side, giving d(5) :- e. The third fact is g(5) which can match the last rule, so c(5) is proved. This implies the new rule b(5) :- d(5) analogous to the earlier offspring of this rule. The last fact is e, which can be used to proved d(5) from our previously derived d rule. This new fact d(5) can match the right side of another derived rule, proving b(5). This matches the right side of the first rule, proving a(5).

6-6. The facts true at some point in hybrid chaining are like the object being worked on an assembly line. New facts represent new parts or new connections made to the objects on the assembly line. As the rules proceed a major conclusion is "built up" like an object assembly. Some rules may not succeed in producing new facts, and they are like optional equipment on the assembly line that is only used for production of certain kinds of objects.

6-10. (a) In decision lattices a single best path is taken through the lattice based on questions asked at each node. In the and-or-not all possible paths are explored in parallel (often by electronic signals). So decision lattices require user interaction, while and-or-not lattices don't. Decision lattices require a lot of work to construct from Prolog rules, while and-or-not lattices are a simple translation of what Prolog rules mean. Decision lattices can represent nearly any rule-base system, whereas and-or-not lattices can only represent

rule-base systems with no or limited variables. Decision lattices are good when you want to save space because their implementation is very simple, whereas and-or-not are good when you want to save time, and especially good when you have access to special-purpose hardware.

(b) Both represent "compilation" of rule-base systems, more efficient but harder to understand and modify than Prolog rules describing the same rule-base system.

6-11.(a) One solution: assign a fixed rule order of H, I, J, M, K, L, D, C, E, F, B, G, A. Then with backwards chaining, rules would be invoked in order H, I, D, J, E, M, D, K, F, L, A. Note you don't actually have to do any work to satisfy C and E here.

(b) Forwards chaining and rule-cycle hybrid chaining would also work well. Another example: use a "use-same-room-if-possible" meta-rule as the basis of a control structure. That is, prefer to apply rules the mention the current room (and if there are still two rules that apply, chooses the first one in rule order). Then the above rules would be invoked in order H, I, D, J, E, M, D, K, F, L, A (same order as the preceding, but rules could be listed in any order).

6-13. (a) plaintiff_money(M) :- award(A), pool(P), M is A+25+(P/2). defendant_money(M) :- award(A), pool(P), M is 25+(P/2). pool(0) :- award(A), A>=500. pool(P) :- award(A), A<500, P is 500-A.

(b) plaintiff_money(M) :- award(A), A>=500, M is A+25. defendant_money(25) :- award(A), A>=500. plaintiff_money(M) :- award(A), A<500, M is 275+(A/2). defendant_money(M) :- award(A), A<500, M is 275-(A/2).

(c) Yes, a large rule-based system is needed to apply laws of mathematics to simplify the arithmetic expressions created by the substitution process, simply because there are so many laws of mathematics. So-called "symbolic algebra" systems can do this, and they're generally big, complex programs.

(d) You lose the ability to name parts of the computation, so it's harder to debug and modify such programs. Also, the mathematical expressions tend to get more complicated (though it didn't happen above), and it's harder to see what's happening.

(e) This is for compiling arithmetic, not logic. It doesn't map to hardware so easily. It does handle variable bindings.

6-14. (a) It's a meta-rule, a rule controlling the selection of other rules.

(b) "Simplest" is very hard to define. It can't just mean the rules with the fewest predicates on their right sides, because the complexity of those predicates matters even more. You can't just assign overall "complexity numbers" to those predicates either, because the complexity is a function of the context in which the predicates occur and the way they relate to each other as much as in the predicates themselves. Also, there's another problem: the simplest explanation of something is often the most general explanation, and such explanations aren't very useful.

6-15. Bureaucracies are less deterministic: if orders are given, they may not be followed, or may be misunderstood and misapplied when followed. Still, there are analogies. Policies, regulations, and orders are formulated to constrain the activities of subordinates in certain desirable ways. So they can eliminate the possibility of applying certain action methods or rules (like meta-rules), or they can set the priority of

consideration of action methods or rules (like rule-ordering criteria).

6-16. Good names are important in any programming. So we'll define:

"above_limit"=S1, "at_limit"=S2, "below_limit"=S3, "two_lane"=S4, "four_lane"=S5, "intersection"=S6, "car_close"=S7, "2_to_4"=S8, "4_to_2"=S9, "brake_lights"=S10, "closing"=S11, "passing"=S12; and "speed_up"=A1, "slow_down"=A2, "maintain_speed"=A3, "pass"=A4. Conditions for slowing down are most critical, so their rules should go first. One possible set of rules:

action(slow_down) :- danger. action(slow_down) :- above_limit. danger :- car_close, brake_lights. danger :- 4_to_2. danger :- intersection. danger :- confused. action(speed_up) :- not(danger), below_limit. action(pass) :- passing. action(pass) :- not(4_to_2), not(intersection), car_close, closing. action(maintain_speed). confused :- 2_to_4, 4_to_2. confused :- above_limit, below_limit. confused :- above_limit, at_limit. confused :- at_limit, below_limit.

Many more rules defining "confused" can be added; they're error checkers. You query these rules by:

?- action(X).

much like the traffic-lights program of Chapter 4.

6-17. (a) Random changes to a system that always does completely thorough reasoning with a set of goals or facts just changes the order of that reasoning, not the set of conclusions reached. But if we assume people do incomplete reasoning much of the time--that is they don't follow their reasoning chains very far--then random rearrangement might make certain conclusions possible that weren't reasonable before. In fact, there's much evidence that people do have such limitations, tending to get lost in complicated reasoning chains. Of course, such random rearrangements could also make other inferences that used to be easy become harder.

(b) Again, if a system always does complete reasoning on a set of facts or a set of goals, a speedup of a factor of K in the reasoning for all the steps in a chain means a speedup of K for the entire chain. But if we assume that people don't do complete reasoning, and in fact tend to get increasingly confused the longer the reasoning goes on (which is documented by experiments), then the faster the reasoning steps occur the more humans can reason. This might permit all sorts of new conclusions that human beings couldn't make before, and give effects far more than proportionate to the speedup.

7-3.(a) Yes, it's often typical; for instance, many auto repair books speak only in terms of symptoms and fixes. One reason is that people often have trouble thinking up the appropriate intermediate predicates, because they're often abstract and they're not strictly necessary. This presents a problem in building expert systems, because they can't be efficient unless intermediate predicates are defined.

(b) Intermediate predicates are a form of abstraction, and abstraction makes software easier to build, easier to understand, and more reliable. While abstraction does have a cost overhead in design, and also perhaps an overhead in implementation too, good software engineering practice has long emphasized its importance and necessity.

7-4. [Too long to give here]

7-6. [Too long to give here]

7-7. Deletion can be done whenever all ways of using a rule have been exhausted. That occurs when either:

--the exact left side of the rule is now a fact;

--there exists another rule with the exact same left side, whose right side predicate expressions are exactly a subset of those of this rule;

--the rule was created from another rule by matching a predicate expression without binding variables, or only binding variables that did not occur on the left side of the rule;

--the rule contains a "not" whose argument is now a fact;

--the rule right side includes a predicate name that does not occur in a fact or in the left side of any other current rule;

--the rule right side contains a predicate expression with at least one variable, with a predicate that only appears in facts, and every possible binding of the variable(s) has been used to generate a new rule;

--some rule-based systems may have rules in partitions such that if no facts of a particular type are present, then the rules of that partition are useless;

--some rule-based systems may have rules in "levels" such that once all the initial facts have been considered, all the bottom-level rules that refer only to facts can be eliminated. 7-8. [Too long to give here]

7-9. [Too long to give here]

7-10. [Too long to give here]

7-12. [Too long to give here]

7-13. Just change the first term on the right side of the "pursuit" rule to "pick_rule(L,R)", defined analogously to the hybrid version:

pick_rule(L,R) :- rule(L,R), not(better_rule(L,R)). better_rule(L,R) :- rule(L2,R2), prefer(L2,R2,L,R).

7-14. (a) Each "prefer" rule represents conditions for ruling out a rule. So just write another one:

prefer(L,R,L2,R2) :- proved(L2), not(proved(L)). proved(F) :- fact(F). proved(F) :- usedfact(F).

(b) Use the meta-rule:

prefer(L,R,L2,R2) :- not(last_used(L,R)), last_used(L2,R2).

where we modify "pick_rule" as follows:

pick_rule(L,R) :- rule(L,R), not(better_rule(L,R)), checkretract(last_used(L2,R2)), asserta(last_used(L,R)).

where "checkretract" is defined as

checkretract(F) :- call(F), retract(F). checkretract(F) :- not(call(F)).

7-15. To make this simpler, let askif(device_dead) = a, askif(lights_out) = b, askif(smell_smoke) = c, askif(heats) = d, askif(has('knobs or switches')) = e, askif(knobs_do_something) = f, askif(powerful) = g, askif(hear(pop)) = h, askif(cord_frayed) = i, askif(handyperson) = j, askif(familiar_appliance) = k, askif(device_on) = l, askif(has('an on-off switch or control')) = m.

Then after substitution of intermediate predicates, and elimination of redundant rules (two rules for cord breaks), 18 rules remain:

diagnosis('Fuse blown') :- a, b. diagnosis('Fuse blown') :- e, not(f), b. diagnosis('Fuse blown') :- c, not(d), b. diagnosis('Fuse blown') :- c, not(g), b. diagnosis('Fuse blown') :- a, h. diagnosis('Fuse blown') :- e, not(f), h. diagnosis('Fuse blown') :- c, not(d), h. diagnosis('Fuse blown') :- c, not(g), h. diagnosis('Break in cord') :- a, i. diagnosis('Break in cord') :- e, not(f), i. diagnosis('Break in cord') :- c, not(d), i. diagnosis('Break in cord') :- c, not(g), i. diagnosis('Device not turned on') :- a, not(j), m, not(l). diagnosis('Device not turned on') :- a, not(k), m, not(l). diagnosis('Device not turned on') :- e, not(f), not(j), m, not(l). diagnosis('Device not turned on') :- e, not(f), not(k), m, not(l). diagnosis('Device not turned on') :- c, not(d), not(j), m, not(l). diagnosis('Device not turned on') :- c, not(g), not(j), m, not(l). diagnosis('Device not turned on') :- c, not(d), not(k), m, not(l). diagnosis('Device not turned on') :- c, not(g), not(k), m, not(l).

From counting the occurrences of terms on right sides, we have the following statistics (assuming, as was stated, omission of a fact in a rule means it must be false if it occurs true in other rules, and means it must be true if it occurs false in other rules):

(predicate,count-of-occurrences, count-of-negation-occurrences): a,5,15 b,4,16 c,10,10 d,15,5 e,5,15 f,15,5 g,15,5 h,4,16 i,4,16 j,16,4 k,16,4 l,12,8 m,8,12

Predicate c seem be the most even split, so we'll pick it for the first question in the decision lattice. A "yes" answer to question c means that one of these rules applies:

diagnosis('Fuse blown') :- c, not(d), b. diagnosis('Fuse blown') :- c, not(g), b. diagnosis('Fuse blown') :- c, not(d), h. diagnosis('Fuse blown') :- c, not(g), h. diagnosis('Break in cord') :- c, not(d), i. diagnosis('Break in cord') :- c, not(g), i. diagnosis('Device not turned on') :- c, not(d), not(j), m, not(l). diagnosis('Device not turned on') :- c, not(g), not(j), m, not(l). diagnosis('Device not turned on') :- c, not(d), not(k), m, not(l). diagnosis('Device not turned on') :- c, not(g), not(k), m, not(l).

Here predicate expressions m, not(l), not(j), and not(k) are separately sufficient to distinguish the not-turned-on conclusion. Further questions can be asked to validate it. Otherwise, if predicate expression i is true, further questions can be asked to validate a break in the cord. Otherwise, we can try to validate a blown fuse.

Now suppose c is false. Then the applicable rules are:

diagnosis('Fuse blown') :- a, b. diagnosis('Fuse blown') :- e, not(f), b. diagnosis('Fuse blown') :- a, h. diagnosis('Fuse blown') :- e, not(f), h. diagnosis('Break in cord') :- a, i. diagnosis('Break in cord') :- e, not(f), i. diagnosis('Device not turned on') :- a, not(j), m, not(l). diagnosis('Device not turned on') :- a, not(k), m, not(l). diagnosis('Device not turned on') :- e, not(f), not(j), m, not(l). diagnosis('Device not turned on') :- e, not(f), not(k), m, not(l).

Again predicate expressions m, not(l), not(j), and not(k) are sufficient to distinguish the not-turned-on condition, and expression i for the break-in-cord condition.

The final decision lattice can be described this way:

1. Ask if the user smells smoke. If yes, go to step 2; else go to step 3.

2. Ask if the device has an on-off switch or control. If yes, go to step 4; else go to step 5.

3. Ask if the device has an on-off switch or control. If yes, go to step 8; else stop.

4. Verify if the user is either not good at fixing things or is not familiar with the appliance, and if the device is not turned on; if so, that's the problem. Otherwise, stop.

5. Ask if the cord is frayed. If so, go to step 6; else go to step 7.

6. Verify that either the device does not heat or does not take a lot of power; if so, it's a break in the cord. Otherwise, stop.

7. Verify that (1) either the device heats or takes a lot of power, and (2) the lights are out in the house or the user heard a pop sound; if both are true, it's a blown fuse. Otherwise, stop.

8. Verify that (1) either the device seems dead, or the device has knobs and switches and they don't do anything; (2) the user is either not good at fixing things or is not familiar with the appliance; and (3) the device is not turned on. If all are true, the last is the problem. Otherwise, stop.

9. Ask if the cord is frayed. If so, go to step 10; else go to step 11.

10. Verify that either (1) the device is dead, or (2) the device has knobs or switches and they don't do anything; if so, there's a break in the cord. Otherwise, stop.

11. Verify that (1) either the lights are out or the user heard a pop sound, and (2) either the device is dead or the device has knobs or switches that don't work; if both are true, it's a blown fuse. Otherwise, stop.

7-16. [Too long to give here]

7-17. One way to do it, among many ways, with sample data built in (Chapter 10 will develop a better way to write similar state-changing programs):

```
robot :- repeat, move, goal_reached. goal_reached :- position(30,50). move :- position(X,Y),
stepoptions(DX,DY), X2 is X + DX, Y2 is Y + DY, trymove(X2,Y2), retractblocks(DX,DY), !.
stepoptions(1,0) :- position(X,Y), X < 30, not(verticalblock). stepoptions(0,1) :- not(horizontalblock).
stepoptions(-1,0) :- verticalblock. stepoptions(-1,0) :- not(verticalblock), asserta(verticalblock),
write([verticalblock,asserted]). stepoptions(0,-1) :- horizontalblock. stepoptions(0,-1) :-
not(horizontalblock),asserta(horizontalblock), write([horizontalblock,asserted]). trymove(X,Y) :- newcraters,
not(cratered(X,Y)), not(ravined(X,Y)), withinarea(X,Y), retract(position(X2,Y2)), asserta(position(X,Y)),
time(T), T2 is T + 1, asserta(time(T2)), retract(time(T)), write([x,X,y,Y,t,T2]),nl. retractblocks(0,1) :-
verticalblock, retract(verticalblock), write([verticalblock,retracted]). retractblocks(-1,0) :- horizontalblock,
retract(horizontalblock), write([horizontalblock,retracted]). retractblocks(DX,DY). cratered(X,Y) :-
```

crater(X2,Y2), withintwo(X,X2,Y,Y2), write('We are on the edge of a crater.'), nl. withintwo(X,X2,Y,Y2) :- deviation(X,X2,DX), DX < 2, deviation(Y,Y2,DY), DY < 2. deviation(A,B,D) :- D is A - B, D >= 0. deviation(A,B,D) :- D is B - A, D >= 0. newcraters :- time(45), position(X,Y), X2 is X - 3, asserta(crater(X2,Y)). newcraters :- time(73), position(X,Y), X2 is X + 4, asserta(crater(X2,Y)). newcraters. ravined(X,Y) :- Y = 30, X > 9, write('We are on the edge of a ravine.'), nl. withinarea(X,Y) :- X > -1, X < 51, Y > -1, Y < 51. position(25,0). time(0). crater(20,10). crater(10,25). crater(20,40).

7-18. Causal chain reasoning is only desirable when greater flexibility of reasoning is needed. For instance, when many diseases share common symptoms, and it's hard to give a specific conjunction of symptoms for each. The appliance program is a simplification in which only major symptom-to-cause paths are recorded--that is, summary is done of many causal links. So clearly a better program could reason at a finer level of detail.

8-1. The probability the battery is defective from the second rule is 0.9*0.8 = 0.72. Doing an orcombine with the number from the first rule, we get 1 - 0.5*0.28 = 0.86.

8-2. The first rule gives the probability of "a" as 0.6, and the second as 0.8. We must "orcombine" these two numbers to get a cumulative probability, since they are two pieces of evidence confirming the same thing.

(a) 1 - (1-0.6)(1-0.8) = 1 - 0.08 = 0.92

(b) max(0.6,0.8) = 0.80

(c) min(1,0.6+0.8) = min(1,1.4) = 1.00 8-7. (a) PR (product of the probability and the number of things it can apply to)

(b) Among the R/D rules for a diagnosis, the probabilities of failure are independent too. So the total probability that all R/D rules fail is (1-P) to the R/D power. Hence the probability of that the diagnosis succeeds (it just takes one rule) is 1-(1 - P) to the R/D. That is the inverse of the probability that none of the rules proving a diagnosis will succeed. So the average number of diagnoses that succeed is D(1-(1-P) to the R/D.

8-8. The simplest way is to consider only direct positive evidence for some prediction (that is, no "cross terms").

predict(W,P) :- bagof(X,total_predict(W,X),XL), orcombine(XL,P). total_predict(W,P) :- bagof(X,previous_predict(W,X),XL), orcombine(XL,P). total_predict(W,P) :- bagof(X,current_predict(W,X),XL), orcombine(XL,P). current_predict(W,0.7) :- current_west_view(W). current_predict(W,0.5) :- current_east_view(W). current_predict(cloudy,0.9) :- raining(X). previous_predict(W,0.3) :- weatherman_prediction(W). previous_predict(W,0.1) :- grandma_memory, grandma_prediction(W). previous_predict(W,0.8) :- secretary_has_radio, not(secretary_out_to_lunch), radio_prediction(W).

Then to use these rules, just query:

?- predict(W,P).

type semicolons, and the probabilities obtained tell you how likely each kind of sky is. For instance, if you

have the facts

current_west_view(cloudy). current_east_view(sunny). weatherman_prediction(partly_cloudy). secretary_has_radio. radio_prediction(sunny).

then you get:

?- predict(W,P). W=cloudy, P=0.7; W=partly_cloudy, P=0.3; W=sunny, P=0.9;

8-10. (a) The problem arises because students are not typical adults, but are exceptions to a usually-true rule. So just as with the traffic lights program we can write an exception rule and modify the old rule accordingly, as for instance:

employed(X,P) :- student(X,P2), P2>0.1, P is 1.05-P2. employed(X,P) :- adult(X,P2), student(X,P3), P3=<0.1, P is P2 * 0.9.

We can also do much the same thing with just one rule:

employed(XP) :- adult(X,P2), student(X,P3), P is P2*(1-P3)*0.9.

but then we have to commit ourselves to a theory of how the two probabilities interact.

(b) The conservative assumption applied to Joe gives 0.9 + 0.95 - 1 = 0.85. Not much less than the independence-assumption value of 0.855.

(c) The problem is that rule-strength probabilities is not quite the same as additional conjuncts in a rule, for purposes of probability combination. So this is different than a rule that says X is employed if X is an adult and X pays property taxes, where each of those two things has an associated probability. A true combination for rule strength the value might be 0, because the evidence of the incoming conclusion probability could be arbitrarily diluted in reaching the second conclusion. (Actually, it's never possible to exactly get 0, but 0 is a lower bound.)

One general way to handle situations like this is to extend evidence-combination to use more than just probabilities as inputs. Probability rules could return not just a probability, but a list of predicates that succeeded in getting the probability. Then a rule like "employed" above can check whether it is particularly sensitive to anything in the list (by checking prestored sensitivity-pair facts), and fail if so; special-case rules for "employed" could then do a different calculation. Another approach is to use Bayes' Rule to estimate probabilities: store certain probabilities of two things happening simultaneously, and reason from those.

8-11. (a) "Happy" and "unhappy" are not true opposites in English--a person could have neither word apply to them, so the two words don't describe possible emotional states. Notice that this is different from the words "done" and "undone" which do represent true opposites. So you sometimes have to be careful analyzing English words.

(b) It is dangerous when building rule-based expert systems with probabilities when the probability of X (e.g. "happy") plus the probability of not(X) (e.g. "unhappy") isn't 1.0. Then there's a three-valued logical situation here: yes, no or neither. So you couldn't get one from the other by subtracting the other from 1. Instead we must reason separately about each of the two rules, writing separate rules about the value of each.

8-12. [Too long to give]

9-1. (a) A state must be a snapshot of a single time, not several times.

(b) Temperature varies continuously--there can be no abrupt changes in temperature. (But a physicist might call this a physical state.)

(c) It's too complicated to describe as just a list of facts. Also, it seems to involve continuous-valued, not discrete, phenomena.

9-3. No. For example, this set of branches:

branch from "a" to "b" branch from "a" to "c" branch from "b" to "d" branch from "b" to "e" branch from "c" to "d" branch from "c" to "e" branch from "d" to "f" branch from "e" to "f"

9-4. (a) A simple way is to abandon the focus-of-attention approach, and put new facts at the end of facts. In our implementation of breadth-first search in the next chapter we'll use this same trick.

(b) A breadth-first approach might be to check all rules whose left sides match query predicates to see if their right sides are all facts: then check all rules whose left sides match the terms of all the previous right sides are all facts and so on. In other words something that sounds suspiciously like the rule-cycle hybrid. The only difference is that rules are being invoked top-down, so an unsuccessful rule need not be tried over and over again. Actually, such an invocation scheme would be a good way to organize hybrid chaining to make it more efficient.

9-5.(a) depth-first, since it would be awkward and silly to constantly return to previous nodes in real time. This is a good example of the difference between computer environments and the real world: it's easy to return to previous nodes on a computer, but hard in the real world. Note of course you shouldn't use plain depth-first search while driving, but you should use heuristics like those of part (i) to avoid getting too far away from the goal.

(b) plan routes from several other places to the goal, places just off the route

9-6.(a) state b (sequence is a, e, s, b). Costs and evaluation function values are irrelevant.

(b) state r (sequence is a, b, o, r). Costs are irrelevant.

9-7. (a) Vertical height on the page can be measured with numbers, but you can also use "above(X,Y)" to represent relationships on the page, to the same effect.

(b) Heuristics are binary: they either must apply or not apply. So to every possible permutation of success or failure of heuristics, assign a number. Store these numbers in a big table, and just look them up as an evaluation function. This evaluation is discrete and not continuous as a better one might be, but it's perfectly acceptable; you may be able to fit a curve to the numbers to get a continuous evaluation function. Note you must rank permutations of success/failure for all the heuristics, not just rank the heuristics, because a branch choice is usually determined by several heuristics working together. It also won't work in general to assign weights to heuristics and sum up the weights for the heuristics that apply to a state, because there are many ways to get the same sum.

(c) The mapping can often be done if you know all the possible states that occur in a problem. Then you can just write very specific heuristics that say if state X occurs then do operator Y, where this recommendation of Y is consistent with the evaluation function. But if you don't know all the possible states or if there are an infinity of them, you can't do this mapping. For instance, you can do this mapping for route planning in a city, because there are a finite number of intersections in a city and they're the only points at which choices can be made; but you can't for route planning through a forest because there are an infinite number of choices you can make about your route.

9-9. (a) branch-and-bound search

(b) best-first search

(c) breadth-first search

9-12.(a) Here are some:

--Prefer the branch that moves terminals only to their proper floor. --Prefer branches that move two terminals instead of one. --Prefer branches that move terminals to adjacent floors (but that may not be a good heuristic, since it may cause you to move terminals twice).

(b) One way is to count the number of terminals on wrong floors and divide by two (two is the maximum number of terminals that can be moved in any one step). But this doesn't take into account that terminals two floors away are going to cost 1.2 times more, so a better evaluation function (that is still a lower bound) would be to sum up over all terminals .5 for each terminal that is one floor away from its correct floor, and .6 for each terminal that is two floors away from its correct floor. (Note that an "evaluation function" is a function definition, not a number.)

(c) Yes, branching factors are similar in both directions. This is because the operators are all reversible (if you take the elevator and some terminals somewhere, you can always take them back). Note you could use A* in both directions with bidirectional search, so A* is not incompatible with bidirectional (and note this would mean fewer node expansions than A* just one-way). It's true there are three goal states (the elevator can be in three possible places), but that just means you shouldn't search as deep going backwards as going forwards, not that bidirectional search won't work.

(d) There are F different floors on which to place a single terminal. There are T terminals to place. So for T=1, there are F states; for T=2, F*F states; and for F=3, F*F*F states. There are also F places to put the elevator. So in general the number of different states is F multiplied by itself T+1 times, or F to the T+1 power. However, you could argue that all the terminals destined for a single floor are indistinguishable. This would reduce the preceding number since then some of the states considered would be identical, but figuring out exactly how many involves some difficult mathematics, so the above upper bound (or something close) is sufficient for this answer.

(e) Finding a good way to represent the state was very important for this problem. A good way is to code the terminals with digits representing what floor they belong on, and bring together all the terminals on one floor into a list. So states can be represented as [First,Second,Third,Elevator] where the first three terms are lists containing the numbers 1, 2, and 3 an indefinite number of times, and Elevator is the number of the floor the elevator is on. Then the starting state is [[2,3,3],[1],[2],1], which has evaluation 2.5 with the first-mentioned evaluation function, and cost 0 for a total of 2.5. Notice that your state representation must indicate the

position of each terminal individually because a state description should contain all the information necessary to uniquely specify a state. So it wouldn't do, say, to only indicate for each floor how many terminals are there, because some of the terminals there might be destined for different floors, and your plan to move them would be different than with other combinations of the same number of terminals on that floor.

From the starting state we can go to one of eight states (using again the first-mentioned evaluation function):

[[3,3],[1,2],[2],2] -- evaluation 2.0, cost 1, total 3.0 [[3,3],[1],[2,2],3] -- evaluation 2.5, cost 1.2, total 3.7 [[2,3],[1,3],[2],2] -- evaluation 2.5, cost 1, total 3.5 [[2,3],[1],[2,3],3] -- evaluation 2.0, cost 1.2, total 3.2 [[2],[1,3,3],[2],2] -- evaluation 2.5, cost 1, total 3.5 [[2],[1],[2,3,3],3] -- evaluation 1.5, cost 1.2, total 2.7 [[3],[1,2,3],[2],2] -- evaluation 2.0, cost 1, total 3.0 [[3],[1],[2,2,3],3] -- evaluation 2.0, cost 1.2, total 3.2

The sixth state is the lowest-cost, so we choose it for expansion. It leads to six new states (remember, there's no point writing down states we found earlier):

[[2,2],[1],[3,3],1] -- evaluation 1.5, cost 2.4, total 3.9 [[2],[1,2],[3,3],2] -- evaluation 1.0, cost 2.2, total 3.2 [[2,3],[1],[2,3],1] -- evaluation 2.0, cost 2.4, total 4.4 [[2],[1,3],[2,3],2] -- evaluation 2.0, cost 2.2, total 4.2 [[2,2,3],[1],[3],1] -- evaluation 2.0, cost 2.4, total 4.4 [[2],[1,2,3],[3],2] -- evaluation 1.5, cost 2.2, total 3.7

All these totals are larger than than the 3.0 figure found for the first and seventh nodes of the first expansion. By heuristic, choose the first of these two. This expands to:

[[1,3,3],[2],[2],1] -- evaluation 1.5, cost 2, total 3.5 [[3,3],[2],[1,2],3] -- evaluation 2.0, cost 2, total 4.0 [[3,3],[1],[2,2],3] -- evaluation 2.5, cost 2, total 4.5 [[1,2,3,3],[],[2],1] -- evaluation 2.0, cost 2, total 4.0 [[3,3],[],[1,2,2],3] -- evaluation 2.5, cost 2, total 4.5

If we choose to use the other evaluation function mentioned in part (b) -- counting 0.5 for each terminal one floor away from its correct floor, and 0.6 for each terminal two floors away from its correct floor -- we get slightly different evaluations. For the first expansion, the states are respectively evaluated to 2.2 (total 3.2), 2.7 (total 3.9), 2.6 (total 3.6), 2.1 (total 3.3), 2.5 (total 3.5), 1.5 (total 2.7), 2.1 (total 3.1), and 2.1 (total 3.3). So the sixth state is again chosen for expansion, and its possibilities evaluate to 1.5 (total 3.9), 1.0 (total 3.2), 2.1 (total 4.5), 2.0 (total 4.2), 2.1 (total 4.5), and 1.5 (total 3.7). Again, all these totals are more than a node in the first expansion, but this time it's the seventh node in the first expansion. So we expand that seventh node, getting these new states:

[[1,3],[2,3],[2],1] -- evaluation 1.6, cost 2, total 3.6 [[3],[2,3],[1,2],3] -- evaluation 2.2, cost 2, total 4.2 [[2,3],[1,3],[2],1] -- evaluation 2.6, cost 2, total 4.6 [[3],[1,3],[2,2],3] -- evaluation 2.6, cost 2, total 4.6 [[3,3],[1,2],[2],1] -- evaluation 2.2, cost 2, total 4.2 [[3],[1,2],[2,3],3] -- evaluation 1.6, cost 2, total 3.6 [[1,2,3],[3],[2],1] -- evaluation 2.1, cost 2, total 4.1 [[3],[3],[1,2,2],3] -- evaluation 2.7, cost 2, total 4.7 [[1,3,3],[2],[2],1] -- evaluation 1.7, cost 2, total 3.7 [[3],[2],[1,2,3],3] -- evaluation 1.7, cost 2, total 3.7 [[3],[1],[2,2,3],3] -- evaluation 2.1, cost 2, total 4.1

9-14. Jigsaw puzzles:

(a) all possible "partially completed puzzles", partitions of a set of puzzle pieces into physically connected groups

(b) trying to attach one side of one piece to one side of another piece

(c) a state where each piece is in its own physically-connected partition

(d) a state where all pieces are in one physically-connected partition

(e) usually. If pieces have distinct colors and textures, you can group pieces by color or texture and solve subpuzzles for each color, then connect the subpuzzles.

(f) no, because two pieces you could compare previously might get connected instead to other pieces, and no longer be available

(g) one solution

(h) depends on the puzzle, but typically 4n*4n, n the number of puzzle pieces

(i) it decreases to zero, at a progressively slower rate, because the number of sites to connect pieces decreases with each piece connection

Symbolic integration:

(a) all conceivable algebraic expressions and their integrals

(b) rules of integration, as are given in tables

(c) some particular expression with an integral

(d) any equivalent expression without any integrals

(e) yes--an integral of a sum (or difference) is the sum (or difference) of the integrals.

(f) no, rules require very specific forms, and use of one rule may make another no longer possible

(g) usually the best (simplest) form is desired

(h) depends on the problem

(i) varies with the problem, but will increase if the size of the expression increases. No obvious tendencies over time.

9-16.(a) (iv), A* search--we have both partial costs (the number of machine instructions of a partial allocation) and a lower-bound evaluation function (the number of variable mentions unassigned).

(b) (ii). The first is an evaluation function, and the third is a control strategy. The fourth is something you do automatically in any kind of allocation search (it's part of the definition of the word "allocation" and hence part of the definition of the operator), and what you'd do anyway can't be a heuristic.

(c) (i), always decreases, since each assignment rules out using that variable mention in future states. And nothing ever increases the choices available at some point.

9-17.(a) All theorems of the given postulate-and-assumption form.

(b) If there are K statements, there are K!/(K-2)!2! = K(K-1)/2 ways of taking a pair. Each step will add one more statement. Assuming we start with N postulates, the branching factor for the step S will have an upper bound of (N+S-1)(N+S-2)/2, or approximately (N+S) squared over 2.

(c) One answer: Try to combine the simplest statements that have symbols in common.

(d) One answer: the number of common symbols between the closest existing statement and the goal statement. Larger value means closer to goal.

(e) Yes, theorem-proving is standard example of monotonicity.

(f) No, no obvious subparts.

(g) Heuristics, because it's very hard to quantify closeness to the goal.

(i) No, because A* requires a lower bound on the number of steps to the goal, not an approximation.

(j) If we had a good evaluation function.

(k) Yes, if branching factor seems reasonable in backwards direction (but it will be if we assume all steps involve combination with an original statement, which is true most of the time).

9-18. (a) Suppose we have N data points with K study variables. Then a state for this problem can be expressed as a list of N data points where the first K study variables are the same as the K study variables in the starting state but where there may be additional study variables reflecting operations performed; plus the same target variable for all states. The set of all such states is the search space. (The size of the search space is infinite, because there are an infinite number of combinations and operations you can perform--you can keep squaring a number forever.) Remember, a state must include everything relevant to describe a "snapshot" of activity in the course of a problem, so you must include all the data points in a single state, and define operators to apply to every data point together (i.e., vector operations).

(b) The operators are just the operations described, as for instance squaring all the values for some study variable to get a new study variable, or taking the corresponding product of two study variables to get a new study variable.

(c) The branching factor for any state is infinite because you could use an infinite number of constants in the first of the three described classes of operators.

(d) The initial state has two successors:

data_point([1,2,1],1). data_point([2,0,4],16). data_point([3,1,9],81).

data_point([1,2,4],1). data_point([2,0,0],16). data_point([3,1,1],81).

And the first of these has three successors:

data_point([1,2,1,1],1). data_point([2,0,4,4],16). data_point([3,1,9,9],81).

data_point([1,2,1,4],1). data_point([2,0,4,0],16). data_point([3,1,9,1],81).

data_point([1,2,1,1],1). data_point([2,0,4,16],16). data_point([3,1,9,81],81).

You may not want to count the first state though--it's pointless. The second of the original successors also has three successors:

data_point([1,2,4,1],1). data_point([2,0,0,4],16). data_point([3,1,1,9],81).

data_point([1,2,4,4],1). data_point([2,0,0,0],16). data_point([3,1,1,1],81).

data_point([1,2,4,16],1). data_point([2,0,0,0],16). data_point([3,1,1,1],81).

And you may not want to count the second state here because it doesn't make much sense, nor the first state here because it duplicates (except for a switch of the third and fourth study variables) the second state of the previous group of three.

(e) Statisticians have a number of such criteria. One simple one is that the "least squares" deviation, the average square over all data points of the difference between the target variable value and its corresponding value for the closest study variable, is less than some small fixed number.

(f) The least squares deviation calculation given in part (e) is a natural evaluation function, since squares are always nonnegative and the value does go to zero when the goal is exactly reached.

(g) It's tricky to find something nonnumeric for choosing branches, because the data itself is numeric. One way is to look at the operators: say prefer any operator that involves two variables to any operator that involves only one variable. Another heuristic is to prefer using new study variables not so far used.

(h) Probably not, because we can't do backwards search easily. For one thing, it's hard to get forwards and backwards search to meet in the middle--the branching factor is so larger. For another thing, we'll have to reason somewhat differently backwards, since we don't know what the goal states will be exactly, just a criterion for checking states to see whether they are goals. We could try to reason forwards from a starting state whose study variables are all but one of the forwards-search study variables plus the original target variable, setting the new target variable to be the remaining one of the original study variables. That would be a kind of backwards searching, and if we found a solution this way we could "reverse" the operators to get a corresponding forwards solution path--it's tricky, but possible. But there would be S ways to do this if there were S original study variables, so it'll be considerably harder searching this way--it won't be worth it if S is large.

(i) Not really, because there's no obvious cost function here--we're concerned about the end result, not the difficulty in getting there.

(j) The square root operator requires that all its values be nonnegative, and the logarithm operator requires that all its values be strictly positive. In addition, the quotient operator requires that none of the values of the second variable be zero. Other than these small restrictions, operators can be used any time for this problem.

(k) No, we have no way to pick nice intermediate goals.

(l) Statisticians probably make use of strong analogies to related problems involving some of the same variables. So they "look up" a few related formulae and test them out sequentially, and usually one of them works pretty well. Occasionally they may have to tinker with the formula as a kind of searching, but this is

rare because most phenomena that statisticians attack have been quite thoroughly studied already. You could call this approach a rule-based one. What statisticians do also can often be described as "generate and test" (section 3.12). (They also seem to use means-ends analysis, a technique we'll explain in Chapter 11.)

9-19.(a) You must think ahead to what your opponent would do if you made a particular move. Each move possibility for you has a set of subsequent, and usually different, move possibilities for them. And each of these has a set of replies by you, and so on.

(b) The decisions made on alternate moves (yours and your opponent's) are made according to different criteria: you're trying to win when you're selecting a move, and they're trying to win when they're selecting a move. In terms of the evaluation function, it's like theirs is the negative of yours.

10-2. Here are the necessary definitions:

```
testdepth(L) :- depthsearch([[9,11,9,3,4,5,7,18],[12,16,4,7,2,20,5]],L). goalreached([[],[]]).
successor([LX,LY],[LX2,LY2]) :- member(X,LX), member(X,LY), deleteone(X,LX,LX2),
deleteone(X,LY,LY2). successor([LX,LY],[LX2,LY2]) :- successor2([LX,LY],[LX2,LY2]).
successor([LX,LY],[LX2,LY2]) :- successor2([LY,LX],[LY2,LX2]). successor2([LX,LY],[LX2,LY2]) :-
twomembers([X1,X2],LX), Y is X1 + X2, member(Y,LY), deleteone(X1,LX,LX3), deleteone(X2,LX3,LX2),
deleteone(Y,LY,LY2), length_ok(LX2,LY2). length_ok(L1,L2) :- length(L1,E1), length(L2,E2), E1 + 2 >
E2. twomembers([X,Y],[X|L]) :- member(Y,L). twomembers([X,Y],[Z|L]) :- twomembers([X,Y],L).
deleteone(X,[X|L],L) :- !. deleteone(X,[Y|L],[Y|L2]) :- deleteone(X,L,L2). /* Note the member predicate is
defined in depth-first search file */
```

Here's the program in action, using the depth-first search program.

```
?- testdepth(L).

L=[[[],[]],[[18],[16,2]],[[9,3,18],[12,16,2]], [[9,11,9,3,18],[12,16,2,20]],[[9,11,9,3,7,18], [12,16,7,2,20]],
[[9,11,9,3,5,7,18],[12,16,7,2,20,5]], [[9,11,9,3,4,5,7,18],[12,16,4,7,2,20,5]]] ;

L=[[[],[]],[[9,3],[12]],[[9,3,18],[12,16,2]],[[9,11,9,3,18], [12,16,2,20]],[[9,11,9,3,7,18],[12,16,7,2,20]],
[[9,11,9,3,5,7,18],[12,16,7,2,20,5]],[[9,11,9,3,4,5,7,18], [12,16,4,7,2,20,5]]] ;

L=[[[],[]],[[18],[16,2]],[[11,9,18],[16,2,20]], [[9,11,9,3,18],[12,16,2,20]],[[9,11,9,3,7,18], [12,16,7,2,20]],
[[9,11,9,3,5,7,18],[12,16,7,2,20,5]], [[9,11,9,3,4,5,7,18],[12,16,4,7,2,20,5]]] ;

L=[[[],[]],[[11,9],[20]],[[11,9,18],[16,2,20]], [[9,11,9,3,18],[12,16,2,20]],[[9,11,9,3,7,18], [12,16,7,2,20]],
[[9,11,9,3,5,7,18],[12,16,7,2,20,5]], [[9,11,9,3,4,5,7,18],[12,16,4,7,2,20,5]]]
```

10-3. (a)

```
goalreached(S) :- member(on(c,b,bolt1),S).
```

(b) One helpful one is

```
eval(S,N) :- burial(b,N2), burial(c,N3), right_bolts(N4), right_relationship(N5), N is N2+N3+N4+N5.
right_bolts(0) :- on(b,X,bolt1), on(c,Y,bolt1). right_bolts(1) :- on(b,X,bolt1), right_bolts(1) :- on(c,Y,bolt1),
right_bolts(2). right_relationship(0) :- on(c,b,bolt1). right_relationship(1).
```

(c) The shortest (and hence the breadth-first) solution is to remove all three parts from the first bolt (part "a" can go anywhere but the other two parts must go on the surface, not on "bolt2"), and then put part "b" followed by part "c" on "bolt1".

10-4. (a) This problem suffers from a common weakness of depth-first search problems: it picks a bad initial line of reasoning and gets so lost in pursuing the implications of this bad reasoning that it takes nearly forever to get around to undoing it. Things are further exacerbated by the problem mentioned in part (b).

(b) This program can't recognize that two descriptions of the same state are identical unless the facts describing that state occur in exactly the same order in both. So what we need is a more sophisticated equality check call it "set_equality", that checks that every member of one list is a member of another list and vice versa. We can then replace the "member" definition with a "set_member" that uses this "set_equality" instead. We can define the latter using the "subset" predicate of Chapter 5:

set_equality(L1,L2) :- subset(L1,L2), subset(L2,L1). subset([],L). subset([X|L],L2) :- singlemember(X,L2), subset(L,L2). singlemember(X,[X|L]) :- !. singlemember(X,[Y|L]) :- singlemember(X,L).

then set_member looks like this:

set_member(X,[Y|L]) :- set_equality(X,Y). set_member(X,[Y|L]) :- set_member(X,L).

and we just change the "member" in the definition of "depthsearch2" to "set_member".

(c) The heuristic implementation given in this chapter refers only to ordering of rules. We only have two rules for this problem, so there's just two choices for ordering them. The real problem not addressed is how to order the different ways of binding arguments to those rules.

(d) One heuristic would be to prefer operators that affect parts mentioned in the goalreached condition. To do this we could add four additional rules, two corresponding to each of the original successor rules, that apply to the more specific cases involving the "goal" parts c and e, and place these rules before their more general counterparts:

successor(S,[on(c,surface,none)|S2]) :- member(on(c,Y,B),S), not(B=none), cleartop(c,S), delete(on(c,Y,B),S,S2). successor(S,[on(e,surface,none)|S2]) :- member(on(e,Y,B),S), not(B=none), cleartop(e,S), delete(on(e,Y,B),S,S2). successor(S,[on(X,surface,none)|S2]) :- member(on(X,Y,B),S), not(B=none), cleartop(X,S), delete(on(X,Y,B),S,S2). successor(S,[on(c,Z,B2)|S2]) :- member(on(c,Y,B),S), cleartop(c,S), member(on(Z,W,B2),S, not(B2=none), cleartop(Z,S), not(c=Z), delete(on(c,Y,B),S,S2). successor(S,[on(e,Z,B2)|S2]) :- member(on(e,Y,B),S), cleartop(e,S), member(on(Z,W,B2),S), not(B2=none), cleartop(Z,S), not(e=Z), delete(on(e,Y,B),S,S2). successor(S,[on(X,Z,B2)|S2]) :- member(on(X,Y,B),S), cleartop(X,S), member(on(Z,W,B2),S), not(B2=none), cleartop(Z,S), not(X=Z), delete(on(X,Y,B),S,S2).

10-6. [too long to give]

10-8. (a) This cuts in half the number of facts that need to be represented with the "piece_cost" predicate name, since you only have to represent the costs in one direction along a piece of a route.

(b) An infinite loop could occur in cases where piece_cost would fail when there are no facts that would match a query on it. But in the searches that use costs--branch-and-bound and A* search--a successor fact

must already exist for two intersections before a cost for going between those intersections is found. So it would be easy to check to avoid infinite loops for this case: just make sure that every successor fact has a corresponding cost fact on the same pair of arguments. You could also get infinite loops if you backtrack into the cost function, but notice that our A* search program (the only program given in the chapter having costs) has a cut symbol after the call to "cost" to prevent exactly this.

(c) With the successor facts, there's no such previous checking as was mentioned in part (b). In fact finding all the possible successors of a state is essential to all search strategies, and such a commutative rule would cause you to get stuck on finding successors of the starting state forever. So you'd never get anywhere.

10-9. Here's one way:

```
robot(A) :- search([none,at(chute),dusty(o1),dusty(o2), fullbasket(o1),vacuumable(o2)],A).

goalreached([O,at(X)]) :- not(X=chute).

eval([],0). eval([X|L],N) :- piece_eval(X,N2), eval(L,N3), N is N2+N3. piece_eval(dusty(X),6).
piece_eval(vacuumable(X),10). piece_eval(fullbasket(X),24). piece_eval(holdingbasketfrom(X),1).
piece_eval(P,0).

cost([],0). cost([[O|S]|L],C) :- piece_cost(O,C2), cost(L,C3), C is C2+C3. piece_cost(none,0).
piece_cost(vacuum,10). piece_cost(dust,6). piece_cost(pickup,3). piece_cost(dispose,5).
piece_cost(putdown,1). piece_cost(go(chute,Y),8). piece_cost(go(X,chute),8). piece_cost(go(X,Y),3).

successor([O|S],[vacuum,fullbasket(X)|S3]) :- member(at(X),S), not(member(dusty(X),S)),
failing_delete(vacuumable(X),S,S2), delete(fullbasket(X),S2,S3). successor([O|S],[dust,vacuumable(X)|S3])
:- member(at(X),S), failing_delete(dusty(X),S,S2), delete(vacuumable(X),S2,S3). successor([O|S],
[pickup,holdingbasketfrom(X)|S]) :- member(at(X),S), not(member(holdingbasketfrom(X),S)),
not(member(vacuumable(X),S)), not(member(dusty(X),S)), member(fullbasket(X),S). successor([O|S],
[dispose|S2]) :- member(at(chute),S), member(holdingbasketfrom(X),S), failing_delete(fullbasket(X),S,S2).
successor([O|S],[putdown|S2]) :- member(at(X),S), not(member(fullbasket(X),S)),
failing_delete(holdingbasketfrom(X),S,S2). successor([O|S],[go(X,Y),at(Y)|S2]) :- failing_delete(at(X),S,S2),
member(Y,[chute,o1,o2]), not(X=Y).

member(X,[X|L]). member(X,[Y|L]) :- member(X,L).

delete(X,[],[]). delete(X,[X|L],L) :- !. delete(X,[Y|L],[Y|M]) :- delete(X,L,M).

failing_delete(X,[X|L],L) :- !. failing_delete(X,[Y|L],[Y|M]) :- failing_delete(X,L,M).
```

Here's this program running with the A* program:

```
?- robot(A). 73 incompletely examined states and 255 examined states A=[[putdown,at(o2)],
[go(o1,o2),at(o2),holdingbasketfrom(o2)], [putdown,at(o1),holdingbasketfrom(o2)],
[go(chute,o1),at(o1),holdingbasketfrom(o2), holdingbasketfrom(o1)],
[dispose,at(chute),holdingbasketfrom(o2), holdingbasketfrom(o1)],
[go(o2,chute),at(chute),holdingbasketfrom(o2), fullbasket(o2),holdingbasketfrom(o1)],
[pickup,holdingbasketfrom(o2),fullbasket(o2),at(o2), holdingbasketfrom(o1)],
```

[vacuum,fullbasket(o2),at(o2),holdingbasketfrom(o1)], [dust,vacuumable(o2),at(o2),holdingbasketfrom(o1)],
[go(chute,o2),at(o2),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[dispose,at(chute),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[go(o1,chute),at(chute),holdingbasketfrom(o1), fullbasket(o1),dusty(o2),vacuumable(o2)],
[pickup,holdingbasketfrom(o1),fullbasket(o1),at(o1), dusty(o2),vacuumable(o2)],
[vacuum,fullbasket(o1),at(o1),dusty(o2), vacuumable(o2)],
[dust,vacuumable(o1),at(o1),dusty(o2),fullbasket(o1), vacuumable(o2)],
[go(chute,o1),at(o1),dusty(o1),dusty(o2), fullbasket(o1),vacuumable(o2)],
[none,at(chute),dusty(o1),dusty(o2),fullbasket(o1), vacuumable(o2)]] ;

73 incompletely examined states and 257 examined states A=[[putdown,at(o1)],
[go(o2,o1),at(o1),holdingbasketfrom(o1)], [putdown,at(o2),holdingbasketfrom(o1)],
[go(chute,o2),at(o2),holdingbasketfrom(o2), holdingbasketfrom(o1)],
[dispose,at(chute),holdingbasketfrom(o2), holdingbasketfrom(o1)],
[go(o2,chute),at(chute),holdingbasketfrom(o2), fullbasket(o2),holdingbasketfrom(o1)],
[pickup,holdingbasketfrom(o2),fullbasket(o2),at(o2), holdingbasketfrom(o1)],
[vacuum,fullbasket(o2),at(o2),holdingbasketfrom(o1)], [dust,vacuumable(o2),at(o2),holdingbasketfrom(o1)],
[go(chute,o2),at(o2),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[dispose,at(chute),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[go(o1,chute),at(chute),holdingbasketfrom(o1), fullbasket(o1),dusty(o2),vacuumable(o2)],
[pickup,holdingbasketfrom(o1),fullbasket(o1),at(o1), dusty(o2),vacuumable(o2)],
[vacuum,fullbasket(o1),at(o1),dusty(o2), vacuumable(o2)],
[dust,vacuumable(o1),at(o1),dusty(o2),fullbasket(o1), vacuumable(o2)],
[go(chute,o1),at(o1),dusty(o1),dusty(o2), fullbasket(o1),vacuumable(o2)],
[none,at(chute),dusty(o1),dusty(o2),fullbasket(o1), vacuumable(o2)]] ;

71 incompletely examined states and 333 examined states A=[[putdown,at(o1)],
[go(chute,o1),at(o1),holdingbasketfrom(o1)], [dispose,at(chute),holdingbasketfrom(o1)],
[go(o1,chute),at(chute),holdingbasketfrom(o1), fullbasket(o1)],
[pickup,holdingbasketfrom(o1),fullbasket(o1),at(o1)], [vacuum,fullbasket(o1),at(o1)],
[dust,vacuumable(o1),at(o1),fullbasket(o1)], [go(o2,o1),at(o1),dusty(o1),fullbasket(o1)],
[putdown,at(o2),dusty(o1),fullbasket(o1)], [go(chute,o2),at(o2),holdingbasketfrom(o2),dusty(o1),
fullbasket(o1)], [dispose,at(chute),holdingbasketfrom(o2),dusty(o1), fullbasket(o1)],
[go(o2,chute),at(chute),holdingbasketfrom(o2), fullbasket(o2),dusty(o1),fullbasket(o1)],
[pickup,holdingbasketfrom(o2),fullbasket(o2),at(o2), dusty(o1),fullbasket(o1)],
[vacuum,fullbasket(o2),at(o2),dusty(o1), fullbasket(o1)],
[dust,vacuumable(o2),at(o2),dusty(o1),fullbasket(o1)], [go(chute,o2),at(o2),dusty(o1),dusty(o2),
fullbasket(o1),vacuumable(o2)], [none,at(chute),dusty(o1),dusty(o2),fullbasket(o1), vacuumable(o2)]] ;

71 incompletely examined states and 336 examined states A=[[putdown,at(o2)],
[go(chute,o2),at(o2),holdingbasketfrom(o2)], [dispose,at(chute),holdingbasketfrom(o2)],
[go(o2,chute),at(chute),holdingbasketfrom(o2), fullbasket(o2)],
[pickup,holdingbasketfrom(o2),fullbasket(o2),at(o2)], [vacuum,fullbasket(o2),at(o2)],
[dust,vacuumable(o2),at(o2)], [go(o1,o2),at(o2),dusty(o2),vacuumable(o2)],
[putdown,at(o1),dusty(o2),vacuumable(o2)], [go(chute,o1),at(o1),holdingbasketfrom(o1),dusty(o2),
vacuumable(o2)], [dispose,at(chute),holdingbasketfrom(o1),dusty(o2), vacuumable(o2)],
[go(o1,chute),at(chute),holdingbasketfrom(o1), fullbasket(o1),dusty(o2),vacuumable(o2)],

[pickup,holdingbasketfrom(o1),fullbasket(o1),at(o1), dusty(o2),vacuumable(o2)],
[vacuum,fullbasket(o1),at(o1),dusty(o2), vacuumable(o2)],
[dust,vacuumable(o1),at(o1),dusty(o2),fullbasket(o1), vacuumable(o2)],
[go(chute,o1),at(o1),dusty(o1),dusty(o2), fullbasket(o1),vacuumable(o2)],
[none,at(chute),dusty(o1),dusty(o2),fullbasket(o1), vacuumable(o2)]] ;

no

This program run in one dialect took 336K of heap storage, 5K of global stack, and 22K of local stack, and took 154 seconds to reach an answer.

10-11.(a) 1, if the states form a chain

(b) N-1, for a one-level state tree

(c) 1, as before

(d) N-1, as before

10-13. (a) Here B is the branching factor and K is the level number. As explained in Chapter 9. The number of states in such a search lattice up through and including the level K states is ((B to the K+1 )-B) / (B-1). If fraction P of these B states are visited, then the new branching factor is B(1-P), leading to a total number of states ((B to the K+1) ((1-P) to the K+1) - B(1-P)) / (B(1-P)-1).

(b) The ratio of the old to new number of states (the two formulae of the last section) is approximately (B(1-P)-1) / ((1-P) to the K+1 - (1-P)B to the -K) (B-1) for either K or B large. This ratio will be greater than or equal to 1, and we want to ensure that it is more than the ratio of costs for the visited-state checking, (E+C)/C = 1+(E/C).

(c) Now the second ratio is 1+(AK/C) for level K. So since the first ratio does not depend on level, there will now be some threshold level K at which visited state checking no longer pays off. So to decide whether to use it, you have to guess in advance how far the nearest goal state is from the startingstate.

10-14. (a) No, the "not" term in the second rule fulfills the function of a "cut" in the first rule, and the second and third rules will never apply to the same situation.

(b) It doesn't have a logical meaning like the rest of Prolog--it has a procedural meaning, i.e. it's a programming trick. Also, it can have effects far beyond its immediate vicinity, since it prevents backtracking in any rule with the same left side as the one we're in.

10-16.

f_of_list([],[]). f_of_list([X|L],[FX|FL]) :- call(f(X,FX)), f_of_list(L,FL).

10-17.(a)

prefer(X1,Y1,E1,X2,Y2,E2) :- member(zzxxy(A),X1), not(member(zzxxy(A),X2)).

(b) Use the "length" definition of Chapter 5 and:

prefer(X1,Y1,E1,X2,Y2,E2) :- length(X1,L1), length(X2,L2), L1<L2.

(c) Sum up the number of changes in each path list, and compare:

prefer(X1,Y1,E1,X2,Y2,E2) :- numchanges(Y1,L1), numchanges(Y2,L2), L1<L2. numchanges([S],0) :- !. numchanges([S1,S2|SL],N) :- numchanges2(S1,S2,N2), numchanges2(S2,S1,N3), numchanges([S2|SL],N4), N is N2+N3+N4. numchanges2([],L,0) :- !. numchanges2([S|L1],L2,N) :- not(member(S,L2)), numchanges2(L1,L2,N2), N is N2+1, !. numchanges2([S|L1],L2,N) :- not(member(S,L2)), numchanges2(L1,L2,N).

(d) The last state to find successors of is not one of the six arguments to "prefer" so we must first figure it out. A way that will usually work is to look at the first item on the agenda (which must have been the last successor found), and find the second-to-last state in its path list. This will only be incorrect when the last state chosen had no successors.

prefer(X1,Y1,E1,X2,Y2,E2) :- recent(X2), not(recent(X1)). recent(S) :- first_agenda_item(X,[S2,S3|SL],E), successor(S3,S). first_agenda_item(X,Y,E) :- agenda(X,Y,E), !. 10-18. Breadth-first search is the tool to use, because it always finds minimum-distance paths. So use the problem-independent breadth-first code from the chapter, and define:

inference_distance(X,Y,N) :- setup_goal(Y), breadthsearch(X,Anslist), length(Anslist,N). setup_goal(G) :- retract(goal(X)), !, asserta(goal(G)). setup_goal(G) :- asserta(goal(G)). successor(X,Y) :- a_kind_of(X,Y). goalreached(X) :- goal(X). length([],0). length([X|L],N) :- length(L).

10-19. (a) The state list supplied to the cost function will be in reverse order for the backwards search. So you must reverse this list of states before calling upon the old cost function in order to get the new cost function.

(b) Just take the negative old evaluation function.

(c) You basically want to reverse the arguments to each successor definition. This is easy when the successor definition is a fact or a rule without arithmetic or side effects. If a successor rule has arithmetic you will have to figure out a reverse arithmetic formula to do the calculation in the opposite direction (something not always possible). If there are side effects like assertions or retractions or file loads or cut symbols. The successor rule must pretty well be rewritten because there is usually no obvious way to convert it to the backwards form.

10-20.(a) Change pick_best_move so that it calls upon a two-argument "prefer" predicate, in the manner of section 10.11:

pick_best_move(S,NS) :- bagof(X,successor(S,X),XL), member(NS,XL), not(better_pick_than(NS,XL)). better_pick_than(S,L) :- member(S2,L), not(S=S2), prefer(S2,S).

(b) Just the beginning of the program needs to be changed, so "get_move" is called to generate moves for both players:

gamesearch(State) :- goalreached(State), print_win(State), !. gamesearch([-1,M|B]) :- not(get_move(B,1,S,NB)), write('Player X cannot move so Player O gets another turn.'), nl, !,gamesearch([1,M|B]). gamesearch([P,M|B]) :- print_board(B), pick_best_move([P,M|B],Newstate),

write_move(Newstate), !, gamesearch(Newstate). gamesearch([1,M|B]) :- not(get_move(B,-1,S,NB)), not(get_move(B,1,S,NB)), print_win([1,M|B]), !. gamesearch([1,M|B]) :- not(get_move(B,-1,S,NB)), write('Player O cannot move so Player X gets another turn.'), nl, !, gamesearch([-1,M|B]). gamesearch(S) :- print_win(S), !. write_move([P,[X,Y]|B]) :- Move is (10*X)+Y, write('Player '), decode(P,C), write(C), write(' makes move '), write(Move), nl, !.

(c) Redefine pick_best_move for one player:

pick_best_move([-1|S],NS) :- asserta(ebest(none,1000000)), successor([-1|S],S2), one_ply_eval(S2,E2), write_eval(S2,E2), ebest(S1,E1), E2<E1, retract(ebest(S1,E1)), asserta(ebest(S2,E2)), fail. pick_best_move([-1|S],NS) :- ebest(NS,E), retract(ebest(NS,E)). one_ply_eval(S,E) :- asserta(e2best(1000000,0)), successor(S,S2), eval(S2,E2), e2best(E1,N), min2(E1,E2,E3), retract(e2best(E1,N)), N2 is N+1, asserta(e2best(E3,N2)), fail. one_ply_eval(S,E) :- e2best(NE,N), E is (60.0/(N+1.0))-NE, retract(e2best(NE,N)), !.

(d) There are many ways to do this.

11-2. You might decompose a problem into five parts when two equally important operators seem suggested for a problem. Two operators that you must do in a particular order. Then the five subproblems are: l) satisfy the conditions of the first 2) apply the first operator 3) satisfy the preconditions of the second operator from there 4) apply the second operator 5) go from this state to the goal state. An example would be the state where both the batteries and the light are defective and you have to replace both, and the replace_batteries and replace_light are equally important. To implement this in Prolog you just create another table, "doublerecommended" that gives conditions for these operator pairs to apply. Then you write a third means_ends rule that comes between the two previous rules that checks the "doublerecommended" rules before trying the regular "recommended" rules, recursing as necessary. This way if no "doublerecommended" rules apply to a situation, the "recommended" rules will be used instead.

11-3. (a) The operators are something like:

If in the current state Sir Reginald is alive and in the goal state he is dead then the recommended operator is to stab him with a knife. If you need to get a knife, then the pick up knife operator is recommended. If you need to put down the knife and hide it, then the hide knife operator is recommended. If your goal involves being in a different room than you're at now, then the go operator is recommended. If you need to go to some other room and that door is locked then the knock operator is recommended.

And the informal preconditions and postconditions are:

preconditions of stabbing with a knife: the person you want to stab is alive and you are in the room with them. Postconditions of stabbing with a knife: the person you want to stab is dead. Preconditions of picking up a knife: you do not have a knife, and you are in the room with a knife. Postconditions of picking up a knife: you have a knife. Preconditions of hiding the knife: you have the knife. Postconditions of hiding the knife: you do not have the knife, and the knife is hidden. Preconditions of going from one room to another: that you are at the first room, and that there is no locked door in between. Postconditions of going from one room to another: you are at the second room. Preconditions of knocking on the door: the door is locked. Postconditions of knocking on the door: the door is unlocked. Preconditions of waiting: there are people in the corridor. Postconditions of waiting: there is no one in the corridor.

(b) The final operator sequence is to wait, pick up the knife, knock on his door, go in when he opens it, stab him, then return to your room and hide the knife.

(c) You could figure out a set of plans instead of just one plan, one for each eventuality. Then in actually executing the plan, you follow one along until there is discrepancy between what facts are true and what facts the plan assumes. Then you transfer to another plan that does not have this discrepancy.

11-5. (a) The answer found is to disassemble the case, turn over the case, replace the batteries, disassemble the top, replace the light, assemble the top, and assemble the case.

(b) Reassembling the case is given a higher priority than replacing the light, so after replacing the batteries the case is reassembled even though it needs to be disassembled to replace the light. So two actions are inserted unnecessarily in the operator lists.

(c) Now fixing the light is given the highest priority and fixing the batteries is given lowest priority, lower even than assembling the case. So fixing the light is done first and two unnecessary actions are inserted in the operator sequence similar to the ones in part (b). The operator sequence is to disassemble the case, disassemble the top, replace the light, reassemble the top, assemble the case, disassemble the case, turn over the case, replace the batteries, and assemble the case.

11-8. (a) A breadth-first approach would be to find the preconditions for a given operator then figure out the operator most relevant to finding those preconditions. But instead of now figuring out the preconditions for this new operator we instead turn our attention to the postconditions of the original operator. While we can't figure out exactly what this state will be (because deeper level of recursions that we haven't looked at yet may add some "extra side effect" facts), we can be sure that everything mentioned in postconditions for the original operator must hold in the state resulting from it. So on that basis we can recursively figure out a recommended operator to get to the goal from there, which implies certain postconditions and preconditions of its own. Now we go back to the precondition recursion of the original operator and figure out how to satisfy its preconditions, then how to satisfy its postconditions. So we gradually build up a list of operators, one level at a time. But as we do we may have to redo some later sections of the operator list as assumptions that they made about their states turn out to be violated by earlier states. So you may have to jump around a bit.

(b) As you can see, this approach is much more awkward than depth-first search and involves an added complication in the necessity to redo work that was done previously on occasion. The control structure of depth-first search is better suited to the way computers work, and while it does have the disadvantage of possibly getting into infinite loops if the rules aren't written properly, this seems unlikely. Depth-first search may also require considerable backtracking, but this application doesn't seem like one where this would happen. Mean-ends analysis is a more "intelligent" kind of search, one that is deliberately set up in an intelligent way. So that the best choice is made first and as often as possible, and backtracking is unnecessary most of the time.

11-9. Mean-ends analysis is search with recursive decomposition based on prerequisites and postconditions of "key" actions. A table is used to define these key actions for all start-goal pairs in the problem domain. So whatever application you found to answer this problem, it must involve: (1) recursive decomposition, not just one level of decomposition; (2) prerequisites and postconditions used to determine the decomposition and each level. And also it must involve (3) operating systems and (4) user-friendliness as stated in the problem.

You can use means-ends analysis to let the user to specify high-level goals (like "Run me off a copy of file p375, wherever it is") without requiring him to worry about the steps needed or the many prerequisites. For this example command, for instance, you could first check to see if the file was in any directory on the current computer. If not, you log in on the next most likely computer (which requires that the computer be up), giving password, and check to see if it's there, and so on until you find it on some computer that is up. Then you log out of the remote computer, and enter the network file transfer program to get the file to the local computer. This may involve figuring out file formats for the remote file and the computer you're on, which may involve looking up information in online directories or manuals. Also, you may not find what you're looking for in the first place in the manual that you look, and you may have to follow up on cross-references. Now you want to print, so you first check to see if the text of the file has formatting commands, in which case you send the file to an appropriate formatter. In any event, you then check to find a printer that is working, and if so, send the file, now in printable form, to that printer. What's happening is that certain goals involve certain subgoals (not all of which you know in advance), which in turn involve other subgoals (which you also may not know in advance), so this is a good situation for recursive decomposition as you proceed.

You can also use means-ends analysis to navigate over complicated computer networks, much in the manner of the route-finding example in Chapter 9. To do this, you would have to consider certain computers as high-level goals that you were trying to reach, and then plan how to reach them, and plan what to do after you reach them. An important example of such a high-level goal computer is a gateway to another network; you could plan to get to that computer, plan to log on and get through to the other network, then plan to get to where you're going on the other network. But if there aren't any such "bottleneck" computers you should probably just do a depth-first or breadth-first search.

11-10. Both involve finding a string of words that solve some problem, in the first case an operator sequence to accomplish some goal in the second case a sentence in some language. But for means-ends analysis, the objective is to discover this string, whereas in parsing the string is already known and you want to find the route to it. Both essentially involve substitution of strings of words for other words, though means-ends analysis is not so obvious about it. Both usually imply depth-first control structure because it's easy to use. But means-ends analysis makes its decision about what to do next based on its analysis of preconditions and postconditions, whereas top-down parsing must rely on heuristics for efficiency because there are too many alternative word sequences for one to accurately speak of preconditions and postconditions.

11-11. [too complicated to give]

12-1.(a) Some possibilities: purpose (filled in with "education"), scheduled (filled in with "yes"), sponsoring-institution (filled in with "university")

(b) Superconcept possibilities: class meeting at any university, meeting of any sort at this school. Subconcept possibilities: class meeting for CS3310, class meeting today.

(c) script inherits from class meeting at any university, and latitude-longitude position inherits from class meeting at this school; location, professor, and course number do not inherit from either.

(d) the classroom-lecture script

12-2. People could carry little credit cards containing all answers to questions they are commonly asked, like name, social security number, address, birth date, employer, etc. Then inheritance could have the physical meaning of "look it up on the card and print it on the form". There could be different levels of inheritance: for

instance, in filling out a government form, an employee could provide three credit cards: one themselves, one for their department, and one for their organization, all providing different default values.

12-6. The intension includes definitional information like that it's a form of business communication, it's written in paper with words, and it's usually a formal document. The extension includes the information that memos are common in big organizations and that memos are increasingly being replaced by computer message systems.

12-7. See the next page.

12-8. (a) A value in some slot. Usually the slot is definitional, since adjectives usually help specify something.

(b) The name of a slot. For instance, "of" often signifies a "part_of" relationship; "for" signals a purpose property; "to" signals a destination property or relationship; and "at" signals a directional relationship either in space or in time.

12-10. Note it's important to give unique names to frames (frames are like global variables). It's also important not to use the general English words for instances of a concept; for instance, the frame describing the particular letter should be called something like "letter1" and not just "letter". Also note the experiences reported in the letter occurred in a specific order, and that order must be shown somehow. One solution:

frame-name: letter sender: addressees: subject: reason:

frame-name: letter1 a_kind_of: letter sender: Irate-Programmer addressees: [Fly-by-Nite-Software] subject: refund reason: false-claims1

frame-name: false-claims1 a_kind_of: contract-violation plaintiff: Irate-Programmer defendant: Fly-by-Nite-software summary-report: letter1 violations: [bug1,bug2,bug3]

frame-name: bug product: seller: observer: description:

frame-name: bug1 a_kind_of: bug a_kind_of: numerical-error product: 'Amazing Artificial Intelligence Software' seller: Fly-by-Nite-software observer: Irate-Programmer before: bug2 after: none description: 'attempt to divide by zero'

frame-name: bug2 a_kind_of: bug a_kind_of: lineprinter-output product: Amazing Artificial Intelligence Software seller: Fly-by-Nite-software observer: Irate-Programmer before: bug3 after: bug1 description: '100,000 linefeeds on lineprinter'

frame-name: bug3 a_kind_of: bug a_kind_of: file-system-side-effect product: Amazing Artificial Intelligence Software seller: Fly-by-Nite-software observer: Irate-Programmer before: none after: bug2 description: 'all user files destroyed'

12-11. (a) Intersect the lists inherited. (An empty intersection would mean a contradictory database, so we don't have to worry about it.)

(b) Intersect the ranges inherited, analogous to (a).

12-12. (a) This suggests a priority ranking on the three dimensions, since the user dimension is assumed to have precedence on this issue with most operating systems. Under some circumstances, we might want to leave all the slots empty above our frame in the user hierarchy, in which case the time hierarchy might have the next priority since users are creatures of habit. And if these frames are blank in this slot too, we can defer to the program and facility hierarchy.

(b) Such a warning is very serious business, so any reason at all to give this warning should be heeded. In other words, use the "or" of the Boolean values inherited along the three dimensions.

(c) This parameter is just a guide in allocating resources for the operating system, so just some reasonable estimate is sufficient. That suggests taking the average of the three fractions inherited.

12-14. Rules can be grouped into modules, and each appliance can be characterized by the set of modules that it requires, defining a frame. Thus generalizations over similar appliances (e.g., "drill" and "kitchen appliance") can be characterized by some common set of module frames required by each appliance in the class. Each such frame could be an instance of a multiple-instance slot with the name "rule-module", similar to "part_of". Such module frames will need to also include specifications of when to invoke other frames. They can also include slots for core required, expected execution time, predicate names referred to, a list of facts assumed by all rules in the frame, and an indication of whether backward, forward, or hybrid chaining is best for their rules.

12-15. [many answers possible]

12-16. The general idea is to characterize the two mappings and then compose them (that is, use the results of one mapping as input to the other). The mappings you need to characterize are from A to B and from A to C. To characterize a mapping you must analyze what is different between the two frames in slots and slot values. For this problem, we are especially interested in what new slots are present in the second and are not in the first, and what slot values are filled in in the second frame that are not filled in in the first frame. So we make up a list of such differences from frame A to frame B, a list of differences from frame A between frame C and combine them, eliminating duplicates and finding compromise values for any contradictions. Artificial-intelligence researchers have done some interesting work on this problem.

13-2. Rearrangement of the original query will give a much better execution time than dependency-based backtracking could. To rearrange, first consider which predicates can bind variables. Predicate b cannot bind Y and predicate d cannot bind either X or Z. But the other predicates can be freely moved left in the query. Clearly best among these is predicate e which can match only a single value. It makes predicate a unnecessary, so we can just put e first and c second. So the only remaining question is the order of the final two predicates, b and d. Rule 1 isn't really relevant because we don't know probabilities of success, but using Rule 4 we should put d before b. So the value of Z can be tested right away after the binding of Z. So the best rearrangement of the query is

?- e(X), c(Z), d(X,Z), b(X,Y).

13-3. The dependencies among predicates in this query form a cycle. There's no way a cycle can be turned into a sequence without eliminating at least one of the dependencies in that original cycle.

13-5. Usually the artist (whether verbal, visual, or whatever) has a general idea which they try to implement in an aesthetic manner. This means making choices about what to do in parts of the whole based on the

relationship based on those parts to other parts--in other words constraints. As choices are made further limitations are imposed on later choices so relaxation methods might help.

13-7.(a) examples: sky, cloud, grass, vehicle, road, water, water-vehicle, pier, path, building

(b) Examples: none of these labels can be the same for adjacent regions. Sky and cloud must be above grass, road, water, path, and building. Vehicle must be in or touching road or path, and water-vehicle must be in or touching water. Vehicle cannot be in sky, cloud, or grass; water-vehicle cannot be in sky, cloud, grass, or building. Pier must be touching water, and must also be touching either grass, path, or road. Water cannot be in sky, cloud, vehicle, road, water-vehicle, pier, path, or building. Grass cannot be in sky, cloud, vehicle, road, water, pier, path, or building.

(c) [answer depends on the constraints chosen in (b)]

13-8. The variables are the letters of the acronym; call them N, C, SA, and SB. C1 and C2 are the single-variable constraints, so we apply them first, to get the possibilities: N:[national,naval,new], C:[computer,crazy], SA:[science], SB:[science,school,staff]. Starting with N, "new" is impossible by C6. For C, "crazy" is impossible. For SB, "science" is impossible by C3, and "staff" is impossible by C5. That's all we can do. Final possibilities for the acronym: "national computer science school" or "naval computer science school".

13-9. One way (some steps can be interchanged in the following):

(1) The three small touching objects must be ships by C3.

(2) The region surrounding them must be water by C2.

(3) The "border" region surrounding the entire picture cannot be water by C6, it can't be a cloud by C9, so it's land or marsh.

(4) The small region below the ships is land, marsh, or cloud by C6. But it can't be marsh by C5, so it's land or cloud.

(5) The middle region is land, marsh, or cloud by C6. But it can only be land or cloud by C4 applied to the upper-right T vertex.

(6) The big upper left region is land, marsh, or cloud by C6. But it can only be land or cloud by C4.

(7) The middle region cannot be land any more because marsh has been ruled out for the upper-left region, using C4 part (b). So it must be a cloud.

(8) That means that the semi-small region upper middle of the picture must be cloud by C4 part (b).

(9) The medium-size region left and below the middle region is land, marsh, or cloud by C6. But it can't be marsh by C4 (since then it would violate both parts of C4), so it's land or cloud. But it can't be land, since then either the border region or the big upper left region would need to be water to satisfy C4 part (b) on the left side, and both those labels are impossible. So it must be cloud.

(10) Hence the small region inside the last-considered region is an airplane.

(11) Hence the lower left region must be a cloud too by C4 part (b) applied at the left of the two T vertices.

(12) The small region in the upper left must be an airplane by C2.

13-10. (a) 8 multiplied ten times, or 8 to the 10th power = 2 to the 30th power, or about one billion.

(b) Each propagation cuts the size of the search space (number of possible combinations) in half, so 30 steps are needed on the average.

13-11. It's like the "dual" of the original picture graph, as they say in graph theory. Regions become vertices, vertices become regions, and edges become edges (though with a different meaning).

13-12. The objective is to find some set of weak constraints that when assumed give a unique solution to the problem. For this search problem, the states are subsets of the set of all weak constraints. Goal states are the states whose constraints, plus the original "strong" constraints, give one and only one interpretation to the problem. An evaluation function is the number of solutions possible with the full set of constraints. A cost function in the total probability that a set of weak constraints hold simultaneously. Computing it may be a little tricky, though; it's a form of "and"-combination, so we can use the conservative independence or liberal assumption from Chapter 8. But a more general approach would be to divide the weak constraints into classes where each class is relatively independent in probability from every other class, then use the conservative assumption to combine probabilities within each class, and the independence-assumption combination to combine those. Search with both an evaluation and cost function suggests the A* search strategy. The evaluation function is not a lower bound on the cost function, so we can't be sure the first solution we find is the optimal solution, but that's all right because we are more concerned about getting a reasonable solution than an optimal solution.

13-15. You can reuse much of the code in the cryptarithmetic example. The last eight lines of that file can be copied directly, the lines concerning satisfiability of the uniqueness condition. The other "satisfiable" rule can look very similar to that for "sum":

satisfiable(product,L,[D1,D2,Carryin,Digit,Carryout]) :- some_bindings(L,[D1,D2,Carryin,Digit,Carryout]), P is (D1*D2)+Carryin, P2 is Digit + (Carryout*10), P=P2. satisfiable(unique,L,VL) :- unique_values(L,VL, []).

There are eight distinct digits in the problem so let's call them a, c, d, e, f, g, and h. We'll call the carries c1, c2, and so on. The the choices are:

choices(D,[0,1,2,3,4,5,6,7,8,9]) :- member(D,[c,d,e,f,g]). choices(D,[1,2,3,4,5,6,7,8,9]) :- member(D,[a,h]). choices(C,[0,1,2,3,4,5,6,7,8]) :- member(C,[c1,c2,c3,c4,c5,c6,c7]).

And the constraints can be written:

constraint(product,[h,h,0,a,c1]). constraint(product,[g,h,c1,a,c2]). constraint(product,[f,h,c2,a,c3]). constraint(product,[e,h,c3,a,c4]). constraint(product,[d,h,c4,a,c5]). constraint(product,[c,h,c5,a,c6]). constraint(product,[2,h,c6,a,c7]). constraint(product,[a,h,c7,a,a]). constraint(unique,[a,c,d,e,f,g,h]).

The solution found is 12345679 * 9 = 111111111.

13-16. [too hard to give]

13-17. [too hard to give]

14-3.(a)

(1) on(a,table). (2) on(b,a). (3) on(c,a). (4) on(d,c). (5) color(a,blue). (6) color(b,blue). (7) color(c,red). (8) color(d,green). (9) above(X,Y); not(on(X,Y)). (10) above(X,Y); not(on(X,Z)); not(above(Z,Y)).

(b) The goal is "above(d,Y), color(Y,blue)"; we use proof by contradiction.

(11) Assume the negation of the goal:

not(above(d,Y)); not(color(Y,blue)).

(12) Resolve 11 with 10, taking X=d and Y=Y, giving

not(on(d,Z)); not(above(Z,Y)); not(color(Y,blue).

(13) Resolve 12 with 9, taking X=Z and Y=Y, giving

not(on(Z,Y)); not(on(d,Z)); not(color(Y,blue)).

(14) Resolve 13 with 4, taking c=Z, giving

not(on(c,Y)); not(color(Y,blue)).

(15) Resolve 14 with 5, taking Y=a, giving

not(on(c,a)).

(16) Resolve 15 with 3, giving the null clause.

14-4. No, resolution can only cross out one thing at a time. To see this, suppose we match just the "a" and "not(a)". Then we get as a resolvent "b; not(b)." which is always true (a "tautology"), the exact opposite of the null clause. A tautology is useless because it always is true; it doesn't need to be derived.

14-6. (a)

a; not(b); not(c). a; b; not(d).

(b) a :- c, d.

(c) Given two rules with the same left side, and some term that occurs in unnegated form in one rule and negated form in the other, you can write a new rule with that same left side, whose right side consists of all the terms on both right sides, eliminating duplicates. If furthermore some P and not(P) occur on the new right side, then the left side is a fact.

Before making this new rule a part of the database, we should check that it is not implied by any other single rule. Generally speaking, it's not a good idea to delete the old rules when this new rule is added to the database because it won't often be better than the old rules. (It's likely it has as many or more terms than the

old rules.)

14-7. (a) weekend(T); holiday(T); not(empty(spanagel421,T)).

(b) holiday(T) :- not(weekend(T)), empty(spanagel421,T).

(c) (holiday(T),weekend(T),empty(spanagel421,T),full expression): (true,true,true,true) (true,true,false,true) (true,false,true,true) (true,false,false,true) (false,true,true,true) (false,true,false,true) (false,false,true,false) (false,false,false,true)

(d) The original weekend(T) term was unnegated, so the negated form in the new rule must be a "real" or "true" not--something impossible to ensure in standard Prolog. In other words, the new rule will succeed in more situations than the first rule.

14-8. It requires a more sophisticated kind of reasoning than Prolog is capable of without considerable additional support. We could prove that p(7) is true when p(6) is true using Prolog, but we can't prove that p(N+1) is true for some arbitrary N when p(N) is true, because adding the fact "p(N)" to the database would mean N+1 could match N too (query variables are local to a query, and database variables are local to the database rule or fact). The axiom also requires reasoning about a hypothetical situation, something Prolog can't do directly (though creating "possible worlds" in search processes can come close to this). These are "second-order-logic" and "existential quantification" issues, but they're in a different form from that discussed in Section 14.1. A number of resolution-reasoning systems do allow you to add similar axioms to your clauses and reason about them.

14-10. (a) Change the first line of "resolution" to:

resolution(C1,C2,Cnew) :- twoclauses(C1,C2),

And use a "bagof" to collect all clauses, and then the "append":

twoclauses(C1,C2) :- bagof(C,clause(C),CL), append(L1,[C1|L2],CL), append(L3,[C2|L4],L2). append([],L,L). append([X|L],L2,[X|L3]) :- append(L,L2,L3).

(b) Add "fix_subsets(Cnew)" just before the "asserta(clause(Cnew))" in "resolution", defined as:

fix_subsets(C) :- not(fix_subsets2(C)). fix_subsets2(C) :- clause(C2), subset(C,C2), retract(clause(C)), fail.

15-2. Add an extra argument to all Prolog rules to hold the trace list, and assemble this list using the "append" predicate (section 5.6) as rules succeed. For instance, if an original rule named "r5" was

a(X) :- b(X,Y), c(X).

then change it to

a(X,T) :- b(X,Y,T1), c(X,T2), append([r5|T1],T2,T).

And add another argument "fact" to all facts.

15-4. Pick an arbitrary example case for each conclusion you want the rule-based system to make. Then for

each case, write a rule whose left side is desired conclusion and whose right side is the conjunction of all facts true for that case. For instance, if diagnosis "a" should be made when facts "b", "c", and "e" are true and "d" is false, write a rule

a :- b, c, not(d), e.

15-5. (a) R1 could have the same predicate name on its left side as R2, but R1 could have filled in (that is assigned to constants) one or more arguments that are variables in R2.

(b) First we need a definition: a predicate expression P covers a predicate expression Q if either a Prolog query on P can match a database fact Q, or if Q and P can be written respectively as "not(R)" and "not(S)" and S covers R. We can then say that rule R1 is redundant with respect to rule R2 if (1) every predicate expression on the right side of R2 covers an expression on the right side of R1, and (2) the left side of R2 covers the left side of R1. (The right side of R2 may include additional terms not on the right side of R1.)

15-7. [too complicated to give]

M-2. (a) Search can always be thought of as a problem of traversing a lattice.

(b) Frames form lattices based on "a_kind_of" and "part_of" links.

(c) Decision lattices and and-or-not lattices are important compilation tricks. Also, the partitions created when partitioning a rule-based system could form a lattice.

(d) Resolution is a kind of search, and so can be viewed as lattice traversal where each node is a set true clauses at some point. But also a lattice can model the dependencies between clauses, where each node is a single clause: if a clause X is proved by resolving clauses Y and Z, then you can draw arrows from Y to X and from Z to X.

M-6.(a) Yes, if A causes B, and B causes C, then A causes C (though C may have other causes too).

(b) In two cases. But if A causes B, and C is part of B, then A causes C. And if A causes B, and B is part of C, you could argue that A is one of the causes of C, and consistent with our definition of "A causes C". But if A causes B, and A is part of C or C is a part of A, it doesn't make sense that C could cause B.

(c) One thing is that you can't reason backwards with rules; they're a "one-way" (unidirectional) way of reasoning, whereas often it's useful to reason from effects to causes as well as from causes to effects (as in expert systems for diagnosis of problems). So writing causes as two-argument facts

causation(<cause>,<effect>).

would be more flexible. Also, the Prolog ":-" symbol just reflects one aspect of causation, and it doesn't represent well the notion that events have times, and the time of the effect cannot be before the time of the cause.

M-7.(a) "But" is like an "and" or two assertions (that is, two separate assertions) but where the second assertion violates the usual consequences of the first assertion. Typically, the statement "x but y" means that x and y are facts, yet there's a rule with probabilities (or a rule could be written) that says something like "yo(P) :- x.", where "yo" means the opposite of "y", or "yo" implies the opposite of "y" by a chain of forward-

chaining inferences.

(b) "But" would be useful in summarizing facts, when certain facts don't make an analogy that usually holds. For instance, the facts

a(1,4). a(1,3) a(1,2). a(7,4). a(7,3).

could be summarized as "Predicate a holds for 1 and 7 as first argument, with 2, 3, and 4 as second argument, but not for (7,2)." "But" would also be useful in describing exceptions to general rules, for instance, "the 'flying-ability' predicate usually inherits with value "yes" for birds, but not for penguins, when it is overridden."

(c) No, because it's too much a natural-language ("discourse") signal used to facilitate communication, and it doesn't have much logical meaning beyond an "and".

M-8(a). Sounds like a classic application for a rule-based system with forwards chaining. Probabilities are essential since evidence is weak. Frames for different kinds of objects and their behavior might help organize facts. Blackboards could help if a lot of hypotheses are generated, as well as focus-of-attention conflict resolution. Generate-and-test could be used to explain sensor readings based on observations, but it makes more sense to use it to write rules than to execute them, which doesn't really count. Inheritance might apply if you have hierarchies of object classes. The other techniques listed aren't very helpful for this problem.

(b) Sounds like a classic search problem. It's a search that tries to optimize something, but apparently no path costs are involved, so best-first search is the only applicable search strategy from the choices given. You could use generate-and-test to guess good solutions and verify them, desirable if you have a good generator provided by an expert in the domain of the problem. If you have a lot of constraints (like protecting certain positions sufficiently), relaxation would be helpful. Frames for your resources (personnel and weapons) could summarize their properties, and you could have hierarchies of them. Inheritance and defaults might be helpful on these hierarchies. The A* algorithm could be used if you took into account costs of steps necessary to achieve a configuration. A rule-based system with either forwards or backwards chaining might suggest good approaches, but wouldn't guarantee optimality of a configuration.

M-9. [too complicated to give]

.PA

ANSWERS TO THE ADDITIONAL PROBLEMS IN THIS MANUAL

I-1. (a) "yes", since X, Y, and Z can be bound to 2

(b) "Q=4". For this X=4, Y=2, and Z=2.

(c) Three times, once for every fact that can match the first predicate expression in the rule, b(X,Y). This makes sense because the interpreter can only answer "no" when it's tried every "b" fact.

I-2. (a) The maximum number of answers occurs when there are the maximum number of ways to link the two predicate expressions. This occurs when there are the minimum number of B values. There can't be only one B value, since then there could only be one fact that could satisfy the query, the fact with both arguments identical. So there must be two possible B values; call them "a" and "b".

For the maximum number of query answers, we shouldn't have anything besides a and b as a first argument in a fact, for otherwise those facts couldn't match any queries. So the first argument to every fact must be either a or b.

This gives as the solution the facts involving the only four possible pairs of the two values:

link(a,a). link(a,b). link(b,a). link(b,b).

For this there are eight answers:

A=a, B=a, C=a; A=a, B=a, C=b; A=a, B=b, C=a; A=a, B=b, C=b; A=b, B=a, C=a; A=b, B=a, C=b; A=b, B=b, C=a; A=b, B=b, C=b;

(b) If the only answer to the query is "no", four matchings for the first predicate expression will be tried. That means three backtracks to get the four answers, plus one more backtrack to fail the first predicate expression and fall off the left end of the query.

I-3. (a;not(c)),b,c = (a,b,c);(not(c),b,c) = a,b using the distributive law and then recognizing an "or" of E with a contradiction must be E.

I-4. (a) R=north, S=hall, T=poindexter

(b) 2 times, because it eventually uses the third fact to match the first predicate expression.

I-5.(a) yes, the extra extra just has the potential to rule out previous answers, and can't add new answers

(b) always more time, since all the same work as before is being done before the new expression is reached

(c) yes, for the same reason; the 26-7=19 new failures are all due to the new expression

(d) now sometimes more and sometimes less, because the new expression could decrease time by quickly rule out cases that took a lot of work to rule out previously

I-6.(a) The fourth rule.

(b) Yes: Q=usnavy, T=enterprise, P=usgovernment.

I-7. (a) Yes, in the sense that if A is necessary for B, and B is necessary for C, then A is necessary for C:

necessary(A,C) :- necessary(A,B), necessary(B,C).

It doesn't make sense to call this either "upwards" or "downwards", since events can't be more general than other events. Maybe call this "sideways" transitivity.

(b) Yes, if the "before9" property holds for event B, and A is necessary for B to have occurred, then the before9 property holds for event A:

before9(X) :- necessary(X,Y), before9(Y).

But it doesn't work reasoning in the other direction. Note that even though we wrote the "before9(X)" predicate expression with one argument, you can think of it as equivalent to a predicate "before(X,9amfeb101986)" in the more conventional property-predicate form.

I-8.(a) delete(a,[b,c],[a,c]); this call fails.

(b) X=b

I-9. For instance, we want to convert the list

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

to the list

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

We can write the first list as [a,b|L] and the second as [[a,b]|PL]. And there's the additional condition that PL is the pairing of list L. Hence:

pair([],[]). pair([X,Y|L],[[X,Y]|PL]) :- pair(L,PL).

I-10. intersection([],S2,[]). intersection([X|S1],S2,[X|I]) :- member(X,S2), intersection(S1,S2,I). intersection([X|S1],S2,I) :- intersection(S1,S2,I). member(X,[X|L]). member(X,[Y|L]) :- member(X,L).

I-11. It says the reverse of its list first argument is its second argument. Items are transferred one by one from the first argument to the second (initially empty) second argument, so the second argument, at the bottom level of recursion, ends up in reverse order like a stack. Then at this bottom level of recursion, the basis step binds the third argument to the current state of the second, making permanent so to speak the binding; this is then the answer. Note the program does the same thing when the first argument is bound and second unbound as when the second argument is bound and first unbound; when both arguments are bound, it verifies that one list is the reverse of the other.

I-12. Each of the three facts can in turn match the first predicate expression on the right side of the rule, giving:

bottom(north) :- not(bosses(north,U)). bottom(poindexter) :- not(bosses(poindexter,U)). bottom(hall) :- not(bosses(hall,U)).

There are no more facts, so we must handle "not"s. Only the one in the last rule above succeeds, giving the fact "bottom(hall)". That's all we can prove.

I-13.(a) Fact e(5) doesn't match anything. But f(3) matches the right side of the second rule, so c(3) is a fact. Fact c(3) gives the new rules (in order):

a(3) :- b(3). b(3) :- not(d(3)).

And nothing further can be concluded. So we handle nots. The first one not(e(Z)) fails because e(5) is a fact, but not(d(3)) succeeds. Hence b(3) succeeds, and then a(3) succeeds. If rules are never deleted, the remaining rule

b(Y) :- c(Y), not(d(Y)).

can lead to another rule

b(Y) :- c(Y).

but this doesn't lead to anything. Overall order of new facts: c(3), b(3), a(3).

(b) On the first cycle, the first rule fails, but the second succeeds and proves the fact c(3). The third and fourth rules can't be used yet, and the second cycle doesn't accomplish anything. So now we consider nots for a third cycle. There aren't any in the first two rules, and the not in the third rule fails, but the fourth rule now succeeds and proves b(3). And then on the fourth cycle a(3) is proved. That's all. Overall order of new facts is the same as part (a): c(3), b(3), a(3).

I-14.(a)

diagnosis(chicken_pox) :- rash, not(fever(high)). diagnosis(german_measles) :- rash, cold_symptom, fever(mild). diagnosis(measles) :- rash, cold_symptom, fever(high). diagnosis(mumps) :- fever(mild), sore_throat. diagnosis(scarlet_fever) :- fever(medium), sore_throat. diagnosis(cold) :- cold_symptom, fever(X). fever(high) :- temperature(T), T>102. fever(medium) :- temperature(T), T>100.5, T=<102. fever(mild) :- temperature(T), T>99, T=<100.5. cold_symptom :- sneezing. cold_symptom :- headache. cold_symptom :- sore_throat.

(b) That is, we query

diagnosis(X).

with the above rules and the four specified facts, and keep typing semicolons until we find all answers. However, we are told to take the diseases in a different order than the above, so it's like we rearrange the rules before querying to

diagnosis(measles) :- rash, cold_symptom, fever(high). diagnosis(scarlet_fever) :- fever(medium), sore_throat. diagnosis(chicken_pox) :- rash, not(fever(high)). diagnosis(mumps) :- fever(mild), sore_throat. diagnosis(german_measles) :- rash, cold_symptom, fever(mild). diagnosis(cold) :- cold_symptom, fever(X). fever(high) :- temperature(T), T>102. fever(medium) :- temperature(T), T>100.5, T=<102. fever(mild) :- temperature(T), T>99, T=<100.5. cold_symptom :- sneezing. cold_symptom :- headache. cold_symptom :- sore_throat.

Then the measles diagnosis fails (but establishes the cold_symptom conclusion); scarlet fever succeeds (after first establishing the fever(medium) conclusion); chicken pox succeeds; mumps fails; German measles fails; and cold succeeds.

(c) The sore throat fact matches predicate expressions in the scarlet_fever and last cold_symptom rules, and the new fact cold_symptom is proved. This last is now pursued, and it matches expressions in the measles and german_measles rules. Next temperature(101) is pursued, giving fact fever(medium). This fact causes first the scarlet_fever diagnosis and then the cold diagnosis to be proved.

The desire to go to the circus doesn't match anything in the rules, but the rash fact does match expressions in the measles, chicken pox, and German measles rules. We've run out of facts, so we not handle nots. In this

case, there is a single not expression in the chicken pox rule, and its argument is false, so the expression can be removed, proving the chicken pox diagnosis. Nothing further can be proved.

(d) On the first cycle only the fever(medium) and cold_symptom facts are proved. On the second cycle, the diagnoses of scarlet fever and cold are proved. The third cycle doesn't prove anything, so nots are then handled. On the fourth cycle the chicken pox diagnosis succeeds.

(e) The diseases are considered in order of severity, so the first diagnosis reached is the one you should be most concerned about. That's helpful to know when using the program.

(f) Yes, asking questions of a user is important in this system, and decision lattices are good for question-asking applications. In fact, some "home medicine" books are organized like decision lattices. Decision lattices are particularly good for small computers, and a medical system like this could have a market on home computers. The variable can be easily removed by using predicates like "high_fever" and including ranges in temperature questions, as "Is the fever 99 to 100.5 degrees?"

While just what questions to ask first in the lattice are not immediately obvious, techniques exist to find them by While just what questions to ask first in the lattice are not immediately obvious, techniques exist to find them by focussing on the predicates appearing most often in rules. Just looking at the 12 rules in this problem, a good first question would be whether there's a rash, since three diagnoses mention it and three don't; the next question could be about temperature, because that helps a lot to distinguish among the diseases; then one more question will establish the most serious possible disease.

Decision lattices implicitly represent the answers to previous questions from their location in the tree, so inability to cache is not usually a major obstacle. Inability to backtrack doesn't seem important in this application.

(g) No. Clearly medical rules like these must ask questions of a user since their main application is for non-medical users (doctors have memorized these basic things), and medical facts require judgement by an observer. Computation speed is unimportant when you're asking questions and must wait for the user to answer. Also, you can't ask a user two questions simultaneously, and nearly all predicate expressions in a rule will require a question of the user. And speed of the rule-based system isn't important: medical actions rarely require action within seconds, whereas a well-implemented 1000-rule system should be able to complete its job in at most a second. So if you want a compiled form, the decision lattice sounds much better.

I-15. Or-parallelism is possible between q and u to begin; if u succeeds in binding its argument, q can be terminated. Otherwise, if q succeeds in finding an X, wait until u has failed, and then proceed to query r to bind Y, and simultaneously check s with and-parallelism. If either process fails, terminate both processes and backtrack to q to get another X. If r succeeds and s has either succeed or not finished, start verifying condition t on the same processor that bound Y. If s fails, kill both processes and start q again; if t fails, use the processor that tried to verify it to run r again. If both s and t succeed, the query succeeds.

I-16. Have "impermissible" facts that say what combinations don't go together. Then when parsing or generating a sentence, keep a record via "asserta" of what substitutions were made, and compare to this.

I-17.(a) forward chaining, since it starts from facts and works to a conclusion; it doesn't try to "confirm" anything

(b) forward chaining too, since signals coming in represent facts, which then are propagated through gates to conclusion lines

I-18. Both these questions just involve redefinition of the expected cost R of the rules.

(a) R=MQH + 2(1-MQ)H. So caching is desirable if P(2-MQ)H>1. (b) R=Q*4H/M*M + Q*2*4H/M*M + Q*3*4H/M*M + ... + (1-MQ)*4H/M or R=(2H/M)[Q+2-MQ].

I-19. The intermediate predicates (plus one diagnosis) are the only things that might be useful to cache. But so little computation is needed for any of them that caching would be useless--after all, it takes a lot more microseconds for a user to type answers to queries than for Prolog to explore all the nooks and crannies of this code. Notice there isn't any list processing in the program, a major time-waster in many Prolog programs.

I-20. Any arithmetic computations cannot be "matched" for, but should still be kept in the rule, having their variables substituted for as the variables become bound. Whenever all but arithmetic-computation expressions have been matched for on the right side of some rule, the computations should be performed, and the left side made a fact with the appropriate numerical bindings.

I-21. Note the double "not" prevents backtracking. prog([]). prog([P|L]) :- not(negcall(P)), prog(L). negcall(P) :- not(poscall(P)). poscall(P) :- call(P). poscall(P).

I-22. [specific to your elevator]

I-23. [it depends on your advertisement]

I-24. The rules look like:

russian(0.5) :- russian_maneuvers. russian(P) :- waveform(z37,P2), andcombine([0.8,P2],P).

And we're told we have the facts

russian_maneuvers. waveform(z37,0.5).

The independence assumption would give an and-combination of 0.8*0.5 = 0.4 in the second rule, and the or-combination of this with 0.5 would be 1-((1-0.5)(1-0.4)) = 0.7. That's the answer in part (a).

The conservative assumption would give max(0.8+0.5-1,0) = 0.3 in the second rule, and the or-combination of this with 0.5 is max(0.3,0.5) = 0.5. That's not the answer in part (b), but you didn't need to compute it anyway, because 1.0 is more than 0.7, and conservative combination always gives the smallest possible values.

The liberal assumption would give min(0.8,0.5) = 0.5 in the second rule, and the or-combination of this with 0.5 is min(0.5+0.5,1) = 1.0. That's the answer in part (b).

I-25. (a)

c(P) :- bagof(P,c2(P),PL), orcombine(PL,P). c2(P) :- a(P1), b(P2), andcombine([P1,P2],P). c2(P) :- a(P1), b(P2), NP1 is 1-P1, NP2 is 1-P2, andcombine([NP1,NP2,0.8],P). d(P) :- a(P1), b(P2), NP2 is 1-P2, andcombine([P1,NP2,0.5],P). e(P) :- a(P1), b(P2), NP1 is 1-P1, andcombine([NP1,P2],P).

And define "andcombine" and "orcombine" differently for parts (b) and (c) from parts (d) and (e).

For parts (b)-(e), it helps to define:

test(Pb) :- checkretract(b(X)), asserta(b(Pb)), spectrum(L), test2(L). test2([]). test2([PalL]) :- checkretract(a(Y)), asserta(a(Pa)), b(Pb), c(Pc), d(Pd), e(Pe), write([Pa,Pb,Pc,Pd,Pe]), nl, test2(L). spectrum([0.0,0.2,0.4,0.6,0.8,1.0]). checkretract(P) :- call(P), retract(P). checkretract(P) :- not(call(P)).

which automatically generates the results needed in a list form shown below. The Figures on the next page show parts (b)-(e).

(b)

?- test(0.4). [0,0.4,0.48,0,0.4] [0.2,0.4,0.433279,0.0599999,0.32] [0.4,0.4,0.401919,0.12,0.24] [0.6,0.4,0.38592,0.18,0.16] [0.8,0.4,0.38528,0.24,0.08] [1,0.4,0.4,0.3,0]

(c)

?- test(0.9). [0,0.9,0.0800002,0,0.9] [0.2,0.9,0.23248,0.01,0.719999] [0.4,0.9,0.39072,0.0200001,0.539999] [0.6,0.9,0.554719,0.0300001,0.36] [0.8,0.9,0.724479,0.0400001,0.18] [1,0.9,0.9,0.0500002,0]

.PA .PA

(d) Now "andcombine" and "orcombine" are changed to refer to conservative combination.

?- test(0.4). [0,0.4,0.4,0,0.4] [0.2,0.4,0.199999,0,0.2] [0.4,0.4,0,0,0] [0.6,0.4,0,0,0] [0.8,0.4,0.2,0,0] [1,0.4,0.4,0.0999999,0]

(e) ?- test(0.9). [0,0.9,0,0,0.9] [0.2,0.9,0.0999995,0,0.699999] [0.4,0.9,0.299999,0,0.5] [0.6,0.9,0.5,0,0.3] [0.8,0.9,0.699999,0,0.0999999] [1,0.9,0.9,0,0]

(f) The curves show sharp corners, which doesn't seem too reasonable for an evidence combination method. It's not fair to cite being close to zero as a disadvantage of the conservative method, because maybe the true probabilities really are close to zero; strong correlations are not unusual between terms in expert systems, and then the conservative formula may best describe things.

I-26. (a) c

(b) b

(c) b

I-27. (a) Probably not, because you don't know what your goal states are in advance; they're just any state (object) that has a filled-in value for the property you're interested in. (Note "bidirectional" for search doesn't mean downwards and upwards inheritance, but start to goals and goals to start.)

(b) Upwards reasoning to general or bigger concepts from specific or smaller concepts would have a smaller branching factor, since there are fewer general and bigger concepts. As for start to goals reasoning vs. goals to start reasoning, assuming you could do them both, you can't reach any conclusion about which is better

because goals could be either above or below the start on the inheritance hierarchy.

(c) One example: don't look for an inheritance path that is more than three links long.

(d) One example: in inheriting properties of parts of a car, try a_kind_of links before part_of links.

(e) No, because there are no obvious intermediate states where paths converge.

(f) No, because if an earlier state had a "part_of" link departing from it, that doesn't mean every later state on a path does.

I-28.

```
start(Ans) :- bagof(F,fact(F),FL), bagof(R,rule(A,B),RL), search([FL,[],RL],Ans).
successor([[F|FL],UFL,RL],[NFL,[F|UFL],RL2]) :- pursue(F,RL,RL2,NFL). pursue(F,[],[],F) :- !. pursue(F,
[[Rleft,Rright]|RL],[[Rleft,Rright]|NRL],NFL) :- not(member(F,R)), !, pursue(F,RL,NRL,NFL). pursue(F,
[Rleft,Rright]|RL],NRL,[Rleft|NRL]) :- delete(F,R,[]), pursue(F,RL,NRL,NFL). pursue(F,
[Rleft,Rright]|RL],NRL,[[Rleft,NRright]|NRL]) :- delete(F,Rright,NRright), pursue(F,RL,NRL,NFL).
eval([A,B],N) :- length(A,N1), length(B,N2), N is N1+N2. I-29. [too long to give]
```

I-30.

```
go(Ops) :- means_ends([fire(3),closed,dry], [fire(0),dry],Ops,Finalstate). recommended([fire(X)],hose).
recommended([open],drain). recommended([dry],wait). precondition(hose,[dry,open]). precondition(drain,
[closed]). precondition(wait,[]). deletepostcondition(hose,[dry]). addpostcondition(hose,[flooded]).
deletepostcondition(drain,[closed]). addpostcondition(drain,[open]). deletepostcondition(wait,[flooded]).
addpostcondition(wait,[dry]). transform(hose,L,[fire(0)|L2]) :- singlemember(fire(1),L), delete(fire(1),L,L2).
transform(hose,L,[fire(Y)|L2]) :- singlemember(fire(X),L), X>1, delete(fire(X),L,L2), Y is X-2.
transform(wait,L,L) :- singlemember(fire(0),L). transform(wait,L,[fire(Y)|L2]) :- singlemember(fire(X),L),
X>0, delete(fire(X),L,L2), Y is X+1. transform(drain,L,L).
```

I-31. One way:

```
/* The robot housekeeper problem by means-ends analysis */ test(A,B) :-
means_ends([lastact(none),robot(chute),basket(1,1), basket(2,2),dusty(1),dusty(2),trashy(1),untrashy(2),
notholding(1),notholding(2),vacuumed(1),vacuumable(2)], [robot(1),dusted(1),dusted(2),vacuumed(1),
vacuumed(2),basket(1,1),basket(2,2),untrashy(1), untrashy(2),notholding(1),notholding(2)],A,B).

recommended([dusted(X)],dust(X)). recommended([vacuumed(X)],vacuum(X)).
recommended([untrashy(X)],dispose(X)). recommended([holdingbasket(X)],pickup(X)).
recommended([notholding(X)],putdown(X)). recommended([basket(X,Y)],carry(X,Y)).
recommended([robot(X)],go(X)).

precondition(dust(X),[robot(X),dusty(X)]). precondition(vacuum(X),[robot(X),vacuumable(X),dusted(X)]).
precondition(dispose(X),[robot(chute),trashy(X),
holdingbasket(X),basket(X,chute),dusted(X),vacuumed(X)]). precondition(pickup(X),
[basket(X,Y),robot(Y),trashy(X), notholding(X)]). precondition(putdown(X),[basket(X,X),robot(X),
holdingbasket(X),untrashy(X)]). precondition(carry(X,Y),[basket(X,Z),robot(Z), holdingbasket(X)]).
```

precondition(go(X),[notholding(1),notholding(2)]).

deletepostcondition(dust(X),[dusty(X),vacuumed(X), lastact(A)]). deletepostcondition(vacuum(X), [vacuumable(X),untrashy(X), lastact(A)]). deletepostcondition(dispose(X),[trashy(X),lastact(A)]). deletepostcondition(pickup(X),[notholding(X),lastact(A)]). deletepostcondition(putdown(X), [holdingbasket(X), lastact(A)]). deletepostcondition(carry(X,Y),[basket(X,Z),robot(Z), lastact(A)]). deletepostcondition(go(X),[robot(Y),lastact(A)]).

addpostcondition(dust(X),[dusted(X),vacuumable(X), lastact(dust(X))]). addpostcondition(vacuum(X), [vacuumed(X),trashy(X), lastact(vacuum(X))]). addpostcondition(dispose(X),[untrashy(X), lastact(dispose(X))]). addpostcondition(pickup(X),[holdingbasket(X), lastact(pickup(X))]). addpostcondition(putdown(X),[notholding(X), lastact(putdown(X))]). addpostcondition(carry(X,Y), [robot(Y),basket(X,Y), lastact(carry(X,Y))]). addpostcondition(go(X),[robot(X),lastact(go(X))]).

Here is it running:

?- test(A,B). A=[go(1),dust(1),go(2),dust(2),go(1),vacuum(1),go(2), vacuum(2),go(1),pickup(1),carry(1,chute),dispose(1), carry(1,1),putdown(1),go(2),pickup(2),carry(2,chute), dispose(2),carry(2,2),putdown(2),go(1)] B=[robot(1),lastact(go(1)),notholding(2),basket(2,2), untrashy(2),notholding(1),basket(1,1),untrashy(1), vacuumed(2),vacuumed(1),dusted(2),dusted(1)]

In one version of Prolog, this program used 45K of heap storage, 31K of global stack, and 41K of local stack, and took 3.9 seconds. That's a lot better than the A* version of the same program in Exercise 10-9, but the above answer isn't optimal.

I-32. This means that goal descriptions will also need to contain "not" expressions. So we must add one more rule to the definition of the "difference" predicate:

difference([],S,[]). difference([not(P)|G],S,G2) :- not(singlemember(P,S)), !, difference(G,S,G2). difference([P|G],S,G2) :- singlemember(P,S), !, difference(G,S,G2). difference([P|G],S,[P|G2]) :- difference(G,S,G2).

(In some Prolog dialects you must write "not P" above rather than "not(P)".) You'll also probably want to modify the "recommended" rules to exploit these "not" terms.

I-33. There are no single-variable constraints. Picking variable X first, the first and third expressions both give the possibility list [1,2,3,5]. Picking variable Y next, value 4 is possible by the first expression, but not by the second; value 5 is possible by both; value 3 is possible by both; value 6 is possible by the first but not the second; hence the list [3,5]. Picking variable Z next, 3 satisfies all expressions; 6 fails the third expression; 4 fails the fourth expression. So Z must be 3. Returning to X, it must be 3 by the third expression. And Y must be 3 by the second expression.

I-34. Just subdivide the duplication-checking in the query into checking for each of ten pairs. Move the pair checking to right after the pairs values are bound:

test([T1,T2,T3,T4,T5]) :- classtime(T1), not(occupied(T1)), classtime(T2), not(occupied(T2)), not(T1=T2), classtime(T3), not(occupied(T3)), not(T3=T1), not(T3=T2), classtime(T4), not(occupied(T4)), not(T4=T1), not(T4=T2), not(T4=T3), classtime(T5), not(occupied(T5)), not(T5=T1), not(T5=T2), not(T5=T3),

not(T5=T4), not(two_consecutive_hours([T1,T2,T3,T4,T5])),
not(three_classes_same_day([T1,T2,T3,T4,T5])).

I-35. (a) (ii)

(b) (i)

I-36.(a) possibility lists

(b) lists of probabilities for combination

(c) the search agenda or the paths found to states

(d) the places to backtrack to or the pointers to the database for each previous predicate expression

I-37.(a) yes, if the operator definitions can create a cycle of branches; this is a common problem of depth-first

(b) no, costs would have to keep increasing for any cycle until they were more than the solution path

(c) yes, subtle ordering problems can create loops (remember the means-ends discussed in the book is depth-first)

(d) no, each cycle must cross something out or the cycling stops, and there are a finite number of things to cross out

(e) no, there can't be cycles involving "part_of" because if X is part of Y, then Y can't be part of X, or else the "part_of" concept wouldn't be useful (or else there's a user mistake)

(f) yes, since "is_another_name_for" can make cycles

(g) no, since repeated actions or duplicate clauses are not allowed by any sensible resolution strategy

I-38.(a)

c(a,p,b). c(a,q,c). c(b,q,e). c(b,r,d). c(c,r,d). c(d,r,f). c(e,r,f).

(b) defective(O2) :- c(O1,P,O2), faulty(P). defective(O2) :- c(O1,P,O2), defective(O1).

(c) The query is

?- defective(d).

We can try the first rule, but it fails since process "r" isn't faulty. We can try the second rule, binding O2=d, O1=b, and P=r. Then the new query is

?- defective(b).

And this can match the first rule with O2=b, O1=a, and P=p, since we know that p is faulty. So the b query succeeds, and hence the "d" query succeeds.

(d) defective(O2); not(c(O1,P,O2)); not(faulty(P)). defective(O2); not(c(O1,P,O2)); not(defective(O1)).

(e) Obviously we have to reason backward from f through r through d through r again to c. The "faulty" clause obviously can't help since the given and the conclusion are both about defective objects. So we just use the second clause in part (d) twice. Resolve it first with the fact not(defective(f)) to get

not(c(O1,P,f)); not(defective(O1)).

Resolve this with the fact c(d,r,f) to get

not(defective(d)).

as is obvious from the diagram. Now resolve this fact with again the second clause in part (d):

not(c(O1,P,d)); not(defective(O1)).

And resolve that with the fact c(c,r,d) to get

not(defective(c)).

(f) Yes if we use something other than depth-first and if we suspect that X and Y are connected, since the goal state is known and is unique, and the branching factor is similar in both directions. But if it's more likely that X and Y aren't connected, then the bidirectional search could require significantly more work than a unidirectional search.

(g) (2+2+1+1+1) / 6 = 7/6 (h) successor(X,NX) :- c(X,P,NX).

(i) Designate a variable for each object and each process (the letters in the diagram for part (a)). The values for object variables are "defective" and "nondefective"; the values for process variables are "faulty" and "nonfaulty". The constraints are the connection facts for part (a) and the rules for part (b). But there's a subtle point: the rules can be used backwards as well as forwards, so there should really be four Prolog-style rules total.

(j) Slot inheritance examples: input, output, number of inputs, number of outputs, process name, process type. Value inheritance examples: number of inputs, number of outputs, process name, process type.

(k) Inherit the value from the "a_kind_of" superframe. The "a_kind_of" relationship is usually more fundamental, and in this case the "part_of" superframe can include a lot of irrelevant processes.

(l) Some possibilities:

--Draw the data-flow diagram and assign Cartesian coordinates to the objects; the evaluation function is the straight-line distance between states in the diagram.

--Designate all no-input objects as level 0; then assign level numbers to every other object in the diagram as in breadth-first search. If there are two or more level numbers for a state, pick the smallest. Then the evaluation function is the difference between level numbers of start and goal.

--Group objects into modules, and create a graph indicating which modules connect directly to which other

modules. Let M be the maximum number of nodes in a module. Then an evaluation function is M times the number of links from the module containing X to the module containing Y.

--Let the evaluation function be 1 for objects that are not defective, 0 for objects that are. This is a weak help, but it does qualify as an evaluation function.

Go to book index