Instructor's Manual:
Exercise Solutions
for

Artificial Intelligence
A Modern Approach
Second Edition

Stuart J. Russell and Peter Norvig

PEARSON
Prentice Hall

Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data

Russell, Stuart J. (Stuart Jonathan)
Instructor's solution manual for artificial intelligence : a modern approach (second edition) / Stuart Russell, Peter Norvig.

Includes bibliographical references and index.
1. Artificial intelligence I. Norvig, Peter. II. Title.

Vice President and Editorial Director, ECS: *Marcia J. Horton*
Publisher: *Alan R. Apt*
Associate Editor: *Toni Dianne Holm*
Editorial Assistant: *Patrick Lindner*

Vice President and Director of Production and Manufacturing, ESM: *David W. Riccardi*
Executive Managing Editor: *Vince O'Brien*
Managing Editor: *Camille Trentacoste*
Production Editor: *Mary Massey*
Manufacturing Manager: *Trudy Pisciotti*
Manufacturing Buyer: *Lisa McDowell*
Marketing Manager: *Pamela Shaffer*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN: 0-13-090376-0

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

Preface

This Instructor's Solution Manual provides solutions (or at least solution sketches) for almost all of the 400 exercises in *Artificial Intelligence: A Modern Approach (Second Edition)*. We only give actual code for a few of the programming exercises; writing a lot of code would not be that helpful, if only because we don't know what language you prefer.

In many cases, we give ideas for discussion and follow-up questions, and we try to explain *why* we designed each exercise.

There is more supplementary material that we want to offer to the instructor, but we have decided to do it through the medium of the World Wide Web rather than through a CD or printed Instructor's Manual. The idea is that this solution manual contains the material that must be kept secret from students, but the Web site contains material that can be updated and added to in a more timely fashion. The address for the web site is:

`http://aima.cs.berkeley.edu`

and the address for the online Instructor's Guide is:

`http://aima.cs.berkeley.edu/instructors.html`

There you will find:

- Instructions on how to join the **aima-instructors** discussion list. We strongly recommend that you join so that you can receive updates, corrections, notification of new versions of this Solutions Manual, additional exercises and exam questions, etc., in a timely manner.
- Source code for programs from the text. We offer code in Lisp, Python, and Java, and point to code developed by others in C++ and Prolog.
- Programming resources and supplemental texts.
- Figures from the text; for overhead transparencies.
- Terminology from the index of the book.
- Other courses using the book that have home pages on the Web. You can see example syllabi and assignments here. Please *do not* put solution sets for AIMA exercises on public web pages!
- AI Education information on teaching introductory AI courses.
- Other sites on the Web with information on AI. Organized by chapter in the book; check this for supplemental material.

We welcome suggestions for new exercises, new environments and agents, etc. The book belongs to you, the instructor, as much as us. We hope that you enjoy teaching from it, that these supplemental materials help, and that you will share your supplements and experiences with other instructors.

Solutions for Chapter 1
Introduction

1.1

a. Dictionary definitions of **intelligence** talk about "the capacity to acquire and apply knowledge" or "the faculty of thought and reason" or "the ability to comprehend and profit from experience." These are all reasonable answers, but if we want something quantifiable we would use something like "the ability to apply knowledge in order to perform better in an environment."

b. We define **artificial intelligence** as the study and construction of agent programs that perform well in a given environment, for a given agent architecture.

c. We define an **agent** as an entity that takes action in response to percepts from an environment.

1.2 See the solution for exercise 26.1 for some discussion of potential objections.

The probability of fooling an interrogator depends on just how unskilled the interrogator is. One entrant in the 2002 Loebner prize competition (which is not quite a real Turing Test) did fool one judge, although if you look at the transcript, it is hard to imagine what that judge was thinking. There certainly have been examples of a chatbot or other online agent fooling humans. For example, see See Lenny Foner's account of the Julia chatbot at foner.www.media.mit.edu/people/foner/Julia/. We'd say the chance today is something like 10%, with the variation depending more on the skill of the interrogator rather than the program. In 50 years, we expect that the entertainment industry (movies, video games, commercials) will have made sufficient investments in artificial actors to create very credible impersonators.

1.3 The 2002 Loebner prize (www.loebner.net) went to Kevin Copple's program ELLA. It consists of a prioritized set of pattern/action rules: if it sees a text string matching a certain pattern, it outputs the corresponding response, which may include pieces of the current or past input. It also has a large database of text and has the Wordnet online dictionary. It is therefore using rather rudimentary tools, and is not advancing the theory of AI. It *is* providing evidence on the number and type of rules that are sufficient for producing one type of conversation.

1.4 No. It means that AI systems should avoid trying to solve intractable problems. Usually, this means they can only approximate optimal behavior. Notice that humans don't solve NP-complete problems either. Sometimes they are good at solving specific instances with a lot of structure, perhaps with the aid of background knowledge. AI systems should attempt to do the same.

1.5 No. IQ test scores correlate well with certain other measures, such as success in college, but only if they're measuring fairly normal humans. The IQ test doesn't measure everything. A program that is specialized only for IQ tests (and specialized further only for the analogy part) would very likely perform poorly on other measures of intelligence. See *The Mismeasure of Man* by Stephen Jay Gould, Norton, 1981 or *Multiple intelligences: the theory in practice* by Howard Gardner, Basic Books, 1993 for more on IQ tests, what they measure, and what other aspects there are to "intelligence."

1.6 Just as you are unaware of all the steps that go into making your heart beat, you are also unaware of most of what happens in your thoughts. You do have a conscious awareness of some of your thought processes, but the majority remains opaque to your consciousness. The field of psychoanalysis is based on the idea that one needs trained professional help to analyze one's own thoughts.

1.7

a. (ping-pong) A reasonable level of proficiency was achieved by Andersson's robot (Andersson, 1988).

b. (driving in Cairo) No. Although there has been a lot of progress in automated driving, all such systems currently rely on certain relatively constant clues: that the road has shoulders and a center line, that the car ahead will travel a predictable course, that cars will keep to their side of the road, and so on. To our knowledge, none are able to avoid obstacles or other cars or to change lanes as appropriate; their skills are mostly confined to staying in one lane at constant speed. Driving in downtown Cairo is too unpredictable for any of these to work.

c. (shopping at the market) No. No robot can currently put together the tasks of moving in a crowded environment, using vision to identify a wide variety of objects, and grasping the objects (including squishable vegetables) without damaging them. The component pieces are nearly able to handle the individual tasks, but it would take a major integration effort to put it all together.

d. (shopping on the web) Yes. Software robots are capable of handling such tasks, particularly if the design of the web grocery shopping site does not change radically over time.

e. (bridge) Yes. Programs such as GIB now play at a solid level.

f. (theorem proving) Yes. For example, the proof of Robbins algebra described on page 309.

g. (funny story) No. While some computer-generated prose and poetry is hysterically funny, this is invariably unintentional, except in the case of programs that echo back prose that they have memorized.

h. (legal advice) Yes, in some cases. AI has a long history of research into applications of automated legal reasoning. Two outstanding examples are the Prolog-based expert systems used in the UK to guide members of the public in dealing with the intricacies of the social security and nationality laws. The social security system is said to have saved the UK government approximately $150 million in its first year of operation. However, extension into more complex areas such as contract law awaits a satisfactory encoding of the vast web of common-sense knowledge pertaining to commercial transactions and agreement and business practices.

i. (translation) Yes. In a limited way, this is already being done. See Kay, Gawron and Norvig (1994) and Wahlster (2000) for an overview of the field of speech translation, and some limitations on the current state of the art.

j. (surgery) Yes. Robots are increasingly being used for surgery, although always under the command of a doctor.

1.8 Certainly perception and motor skills are important, and it is a good thing that the fields of vision and robotics exist (whether or not you want to consider them part of "core" AI). But given a percept, an agent still has the task of "deciding" (either by deliberation or by reaction) which action to take. This is just as true in the real world as in artificial micro-worlds such as chess-playing. So computing the appropriate action will remain a crucial part of AI, regardless of the perceptual and motor system to which the agent program is "attached." On the other hand, it is true that a concentration on micro-worlds has led AI away from the really interesting environments (see page 46).

1.9 Evolution tends to perpetuate organisms (and combinations and mutations of organisms) that are succesful enough to reproduce. That is, evolution favors organisms that can optimize their performance measure to at least survive to the age of sexual maturity, and then be able to win a mate. Rationality just means optimizing performance measure, so this is in line with evolution.

1.10 Yes, they are rational, because slower, deliberative actions would tend to result in more damage to the hand. If "intelligent" means "applying knowledge" or "using thought and reasoning" then it does not require intelligence to make a reflex action.

1.11 This depends on your definition of "intelligent" and "tell." In one sense computers only do what the programmers command them to do, but in another sense what the programmers consciously tells the computer to do often has very little to do with what the computer actually does. Anyone who has written a program with an ornery bug knows this, as does anyone who has written a successful machine learning program. So in one sense Samuel "told" the computer "learn to play checkers better than I do, and then play that way," but in another sense he told the computer "follow this learning algorithm" and it learned to play. So we're left in the situation where you may or may not consider learning to play checkers to be s sign of intelligence (or you may think that learning to play in the right way requires intelligence, but not in this way), and you may think the intelligence resides in the programmer or in the computer.

1.12 The point of this exercise is to notice the parallel with the previous one. Whatever you decided about whether computers could be intelligent in 1.9, you are committed to making the same conclusion about animals (including humans), *unless* your reasons for deciding whether something is intelligent take into account the mechanism (programming via genes versus programming via a human programmer). Note that Searle makes this appeal to mechanism in his Chinese Room argument (see Chapter 26).

1.13 Again, the choice you make in 1.11 drives your answer to this question.

Solutions for Chapter 2 Intelligent Agents

2.1 The following are just some of the many possible definitions that can be written:

- *Agent*: an entity that perceives and acts; or, one that *can be viewed* as perceiving and acting. Essentially any object qualifies; the key point is the way the object implements an agent function. (Note: some authors restrict the term to *programs* that operate *on behalf of* a human, or to programs that can cause some or all of their code to run on other machines on a network, as in **mobile agents**.) MOBILE AGENT
- *Agent function*: a function that specifies the agent's action in response to every possible percept sequence.
- *Agent program*: that program which, combined with a machine architecture, implements an agent function. In our simple designs, the program takes a new percept on each invocation and returns an action.
- *Rationality*: a property of agents that choose actions that maximize their expected utility, given the percepts to date.
- *Autonomy*: a property of agents whose behavior is determined by their own experience rather than solely by their initial programming.
- *Reflex agent*: an agent whose action depends only on the current percept.
- *Model-based agent*: an agent whose action is derived directly from an internal model of the current world state that is updated over time.
- *Goal-based agent*: an agent that selects actions that it believes will achieve explicitly represented goals.
- *Utility-based agent*: an agent that selects actions that it believes will maximize the expected utility of the outcopme state.
- *Learning agent*: an agent whose behavior improves over time based on its experience.

2.2 A performance measure is used by an outside observer to evaluate how successful an agent is. It is a function from histories to a real number. A utility function is used by an agent itself to evaluate how desirable states or histories are. In our framework, the utility function may not be the same as the performance measure; furthermore, an agent may have no explicit utility function at all, whereas there is always a performance measure.

2.3 Although these questions are very simple, they hint at some very fundamental issues. Our answers are for the simple agent designs for *static* environments where nothing happens while the agent is deliberating; the issues get even more interesting for dynamic environments.

a. Yes; take any agent program and insert null statements that do not affect the output.

b. Yes; the agent function might specify that the agent print $true$ when the percept is a Turing machine program that halts, and $false$ otherwise. (Note: in dynamic environments, for machines of less than infinite speed, the rational agent function may not be implementable; e.g., the agent function that always plays a winning move, if any, in a game of chess.)

c. Yes; the agent's behavior is fixed by the architecture and program.

d. There are 2^n agent programs, although many of these will not run at all. (Note: Any given program can devote at most n bits to storage, so its internal state can distinguish among only 2^n past histories. Because the agent function specifies actions based on percept histories, there will be many agent functions that cannot be implemented because of lack of memory in the machine.)

2.4 Notice that for our simple environmental assumptions we need not worry about quantitative uncertainty.

a. It suffices to show that for all possible actual environments (i.e., all dirt distributions and initial locations), this agent cleans the squares at least as fast as any other agent. This is trivially true when there is no dirt. When there is dirt in the initial location and none in the other location, the world is clean after one step; no agent can do better. When there is no dirt in the initial location but dirt in the other, the world is clean after two steps; no agent can do better. When there is dirt in both locations, the world is clean after three steps; no agent can do better. (Note: in general, the condition stated in the first sentence of this answer is much stricter than necessary for an agent to be rational.)

b. The agent in (a) keeps moving backwards and forwards even after the world is clean. It is better to do $NoOp$ once the world is clean (the chapter says this). Now, since the agent's percept doesn't say whether the other square is clean, it would seem that the agent must have some memory to say whether the other square has already been cleaned. To make this argument rigorous is more difficult—for example, could the agent arrange things so that it would only be in a clean left square when the right square was already clean? As a general strategy, an agent *can* use the environment itself as a form of **external memory**—a common technique for humans who use things like appointment calendars and knots in handkerchiefs. In this particular case, however, that is not possible. Consider the reflex actions for $[A, Clean]$ and $[B, Clean]$. If either of these is $NoOp$, then the agent will fail in the case where that is the initial percept but the other square is dirty; hence, neither can be $NoOp$ and therefore the simple reflex agent is doomed to keep moving. In general, the problem with reflex agents is that they have to do the same thing in situations that look the same, even when the situations are actually quite different. In the vacuum world this is a big liability, because every interior square (except home) looks either like a square with dirt or a square without dirt.

EXTERNAL MEMORY

Agent Type	Performance Measure	Environment	Actuators	Sensors
Robot soccer player	Winning game, goals for/against	Field, ball, own team, other team, own body	Devices (e.g., legs) for locomotion and kicking	Camera, touch sensors, accelerometers, orientation sensors, wheel/joint encoders
Internet book-shopping agent	Obtain requested/interesting books, minimize expenditure	Internet	Follow link, enter/submit data in fields, display to user	Web pages, user requests
Autonomous Mars rover	Terrain explored and reported, samples gathered and analyzed	Launch vehicle, lander, Mars	Wheels/legs, sample collection device, analysis devices, radio transmitter	Camera, touch sensors, accelerometers, orientation sensors, , wheel/joint encoders, radio receiver
Mathematician's theorem-proving assistant				

Figure S2.1 Agent types and their PEAS descriptions, for Ex. 2.5.

c. If we consider asymptotically long lifetimes, then it is clear that learning a map (in some form) confers an advantage because it means that the agent can avoid bumping into walls. It can also learn where dirt is most likely to accumulate and can devise an optimal inspection strategy. The precise details of the exploration method needed to construct a complete map appear in Chapter 4; methods for deriving an optimal inspection/cleanup strategy are in Chapter 21.

2.5 Some representative, but not exhaustive, answers are given in Figure S2.1.

2.6 Environment properties are given in Figure S2.2. Suitable agent types:

a. A model-based reflex agent would suffice for most aspects; for tactical play, a utility-based agent with lookahead would be useful.

b. A goal-based agent would be appropriate for specific book requests. For more open-ended tasks—e.g., "Find me something interesting to read"—tradeoffs are involved and the agent must compare utilities for various possible purchases.

Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Robot soccer	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Internet book-shopping	Partially	Deterministic*	Sequential	Static*	Discrete	Single
Autonomous Mars rover	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Mathematician's assistant	Fully	Deterministic	Sequential	Semi	Discrete	Multi

Figure S2.2 Environment properties for Ex. 2.6.

c. A model-based reflex agent would suffice for low-level navigation and obstacle avoidance; for route planning, exploration planning, experimentation, etc., some combination of goal-based and utility-based agents would be needed.

d. For specific proof tasks, a goal-based agent is needed. For "exploratory" tasks—e.g., "Prove some useful lemmata concerning operations on strings"—a utility-based architecture might be needed.

2.7 The file `"agents/environments/vacuum.lisp"` in the code repository implements the vacuum-cleaner environment. Students can easily extend it to generate different shaped rooms, obstacles, and so on.

2.8 A reflex agent program implementing the rational agent function described in the chapter is as follows:

```
(defun reflex-rational-vacuum-agent (percept)
  (destructuring-bind (location status) percept
    (cond ((eq status 'Dirty) 'Suck)
          ((eq location 'A) 'Right)
          (t 'Left))))
```

For states 1, 3, 5, 7 in Figure 3.20, the performance measures are 1996, 1999, 1998, 2000 respectively.

2.9 Exercises 2.4, 2.9, and 2.10 may be merged in future printings.

a. No; see answer to 2.4(b).

b. See answer to 2.4(b).

c. In this case, a simple reflex agent can be perfectly rational. The agent can consist of a table with eight entries, indexed by percept, that specifies an action to take for each possible state. After the agent acts, the world is updated and the next percept will tell the agent what to do next. For larger environments, constructing a table is infeasible. Instead, the agent could run one of the optimal search algorithms in Chapters 3 and 4 and execute the first step of the solution sequence. Again, no internal state is *required*, but it would help to be able to store the solution sequence instead of recomputing it for each new percept.

2.10

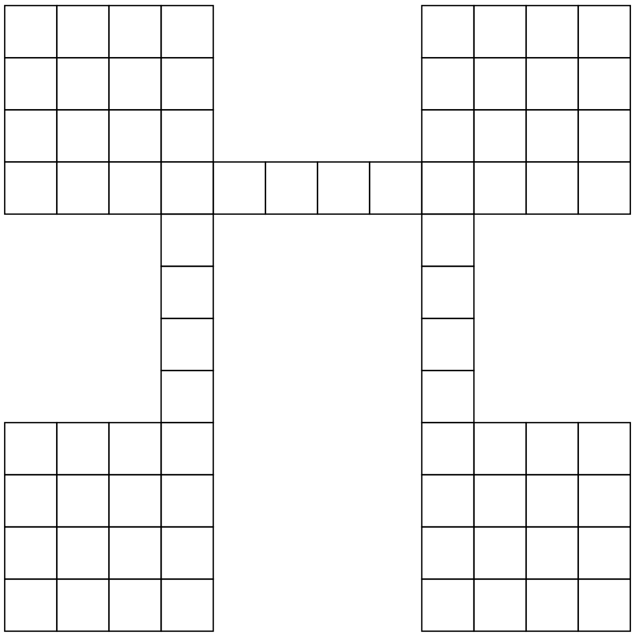

Figure S2.3 An environment in which random motion will take a long time to cover all the squares.

a. Because the agent does not know the geography and perceives only location and local dirt, and canot remember what just happened, it will get stuck forever against a wall when it tries to move in a direction that is blocked—that is, unless it randomizes.

b. One possible design cleans up dirt and otherwise moves randomly:

```
(defun randomized-reflex-vacuum-agent (percept)
  (destructuring-bind (location status) percept
    (cond ((eq status 'Dirty) 'Suck)
          (t (random-element '(Left Right Up Down))))))
```

This is fairly close to what the Roomba$^{\mathrm{TM}}$ vacuum cleaner does (although the Roomba has a bump sensor and randomizes only when it hits an obstacle). It works reasonably well in nice, compact environments. In maze-like environments or environments with small connecting passages, it can take a very long time to cover all the squares.

c. An example is shown in Figure S2.3. Students may also wish to measure clean-up time for linear or square environments of different sizes, and compare those to the efficient online search algorithms described in Chapter 4.

d. A reflex agent with state can build a map (see Chapter 4 for details). An online depth-first exploration will reach every state in time linear in the size of the environment; therefore, the agent can do much better than the simple reflex agent.

The question of rational behavior in unknown environments is a complex one but it is worth encouraging students to think about it. We need to have some notion of the prior probaility distribution over the class of environments; call this the initial **belief state**. Any action yields a new percept that can be used to update this distribution, moving the agent to a new belief state. Once the environment is completely explored, the belief state collapses to a single possible environment. Therefore, the problem of optimal exploration can be viewed as a search for an optimal strategy in the space of possible belief states. This is a well-defined, if horrendously intractable, problem. Chapter 21 discusses some cases where optimal exploration is possible. Another concrete example of exploration is the Minesweeper computer game (see Exercise 7.11). For very small Minesweeper environments, optimal exploration is feasible although the belief state update is nontrivial to explain.

2.11 The problem appears at first to be very similar; the main difference is that instead of using the location percept to build the map, the agent has to "invent" its own locations (which, after all, are just nodes in a data structure representing the state space graph). When a bump is detected, the agent assumes it remains in the same location and can add a wall to its map. For grid environments, the agent can keep track of its (x, y) location and so can tell when it has returned to an old state. In the general case, however, there is no simple way to tell if a state is new or old.

2.12

a. For a reflex agent, this presents no *additional* challenge, because the agent will continue to $Suck$ as long as the current location remains dirty. For an agent that constructs a sequential plan, every $Suck$ action would need to be replaced by "$Suck$ until clean." If the dirt sensor can be wrong on each step, then the agent might want to wait for a few steps to get a more reliable measurement before deciding whether to $Suck$ or move on to a new square. Obviously, there is a trade-off because waiting too long means that dirt remains on the floor (incurring a penalty), but acting immediately risks either dirtying a clean square or ignoring a dirty square (if the sensor is wrong). A rational agent must also continue touring and checking the squares in case it missed one on a previous tour (because of bad sensor readings). it is not immediately obvious how the waiting time at each square should change with each new tour. These issues can be clarified by experimentation, which may suggest a general trend that can be verified mathematically. This problem is a partially observable Markov decision process—see Chapter 17. Such problems are hard in general, but some special cases may yield to careful analysis.

b. In this case, the agent must keep touring the squares indefinitely. The probability that a square is dirty increases monotonically with the time since it was last cleaned, so the rational strategy is, roughly speaking, to repeatedly execute the shortest possible tour of all squares. (We say "roughly speaking" because there are complications caused by the fact that the shortest tour may visit some squares twice, depending on the geography.) This problem is also a partially observable Markov decision process.

Solutions for Chapter 3
Solving Problems by Searching

3.1 A **state** is a situation that an agent can find itself in. We distinguish two types of states: world states (the actual concrete situations in the real world) and representational states (the abstract descriptions of the real world that are used by the agent in deliberating about what to do).

A **state space** is a graph whose nodes are the set of all states, and whose links are actions that transform one state into another.

A **search tree** is a tree (a graph with no undirected loops) in which the root node is the start state and the set of children for each node consists of the states reachable by taking any action.

A **search node** is a node in the search tree.

A **goal** is a state that the agent is trying to reach.

An **action** is something that the agent can choose to do.

A **successor function** described the agent's options: given a state, it returns a set of (action, state) pairs, where each state is the state reachable by taking the action.

The **branching factor** in a search tree is the number of actions available to the agent.

3.2 In goal formulation, we decide which aspects of the world we are interested in, and which can be ignored or abstracted away. Then in problem formulation we decide how to manipulate the important aspects (and ignore the others). If we did problem formulation first we would not know what to include and what to leave out. That said, it can happen that there is a cycle of iterations between goal formulation, problem formulation, and problem solving until one arrives at a sufficiently useful and efficient solution.

3.3 In Python we have:

```
#### successor_fn defined in terms of result and legal_actions
def successor_fn(s):
    return [(a, result(a, s)) for a in legal_actions(s)]

#### legal_actions and result defined in terms of successor_fn
def legal_actions(s):
    return [a for (a, s) in successor_fn(s)]

def result(a, s):
    for (a1, s1) in successor_fn(s):
        if a == a1:
            return s1
```

3.4 From http://www.cut-the-knot.com/pythagoras/fifteen.shtml, this proof applies to the fifteen puzzle, but the same argument works for the eight puzzle:

Definition: The goal state has the numbers in a certain order, which we will measure as starting at the upper left corner, then proceeding left to right, and when we reach the end of a row, going down to the leftmost square in the row below. For any other configuration besides the goal, whenever a tile with a greater number on it precedes a tile with a smaller number, the two tiles are said to be **inverted**.

Proposition: For a given puzzle configuration, let N denote the sum of the total number of inversions and the row number of the empty square. Then $(N mod 2)$ is invariant under any legal move. In other words, after a legal move an odd N remains odd whereas an even N remains even. Therefore the goal state in Figure 3.4, with no inversions and empty square in the first row, has $N = 1$, and can only be reached from starting states with odd N, not from starting states with even N.

Proof: First of all, sliding a tile horizontally changes neither the total number of inversions nor the row number of the empty square. Therefore let us consider sliding a tile vertically.

Let's assume, for example, that the tile A is located directly over the empty square. Sliding it down changes the parity of the row number of the empty square. Now consider the total number of inversions. The move only affects relative positions of tiles A, B, C, and D. If none of the B, C, D caused an inversion relative to A (i.e., all three are larger than A) then after sliding one gets three (an odd number) of additional inversions. If one of the three is smaller than A, then before the move B, C, and D contributed a single inversion (relative to A) whereas after the move they'll be contributing two inversions - a change of 1, also an odd number. Two additional cases obviously lead to the same result. Thus the change in the sum N is always even. This is precisely what we have set out to show.

So before we solve a puzzle, we should compute the N value of the start and goal state and make sure they have the same parity, otherwise no solution is possible.

3.5 The formulation puts one queen per column, with a new queen placed only in a square that is not attacked by any other queen. To simplify matters, we'll first consider the n–rooks problem. The first rook can be placed in any square in column 1, the second in any square in column 2 except the same row that as the rook in column 1, and in general there will be $n!$ elements of the search space.

3.6 No, a finite state space does not always lead to a finite search tree. Consider a state space with two states, both of which have actions that lead to the other. This yields an infinite search tree, because we can go back and forth any number of times. However, if the state space is a finite tree, or in general, a finite DAG (directed acyclic graph), then there can be no loops, and the search tree is finite.

3.7

a. Initial state: No regions colored.
Goal test: All regions colored, and no two adjacent regions have the same color.
Successor function: Assign a color to a region.
Cost function: Number of assignments.

b. Initial state: As described in the text.
Goal test: Monkey has bananas.
Successor function: Hop on crate; Hop off crate; Push crate from one spot to another; Walk from one spot to another; grab bananas (if standing on crate).
Cost function: Number of actions.

c. Initial state: considering all input records.
Goal test: considering a single record, and it gives "illegal input" message.
Successor function: run again on the first half of the records; run again on the second half of the records.
Cost function: Number of runs.
Note: This is a **contingency problem**; you need to see whether a run gives an error message or not to decide what to do next.

d. Initial state: jugs have values $[0, 0, 0]$.
Successor function: given values $[x, y, z]$, generate $[12, y, z]$, $[x, 8, z]$, $[x, y, 3]$ (by filling); $[0, y, z]$, $[x, 0, z]$, $[x, y, 0]$ (by emptying); or for any two jugs with current values x and y, pour y into x; this changes the jug with x to the minimum of $x + y$ and the capacity of the jug, and decrements the jug with y by by the amount gained by the first jug.
Cost function: Number of actions.

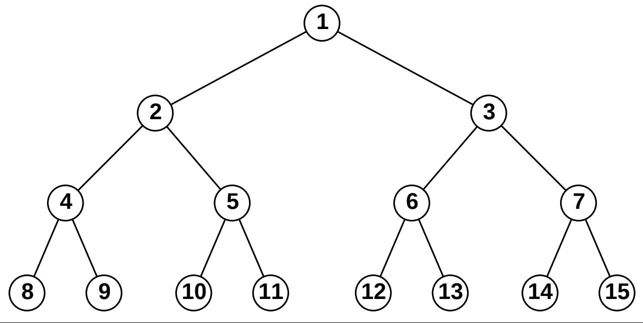

Figure S3.1 The state space for the problem defined in Ex. 3.8.

3.8

a. See Figure S3.1.

b. Breadth-first: 1 2 3 4 5 6 7 8 9 10 11
Depth-limited: 1 2 4 8 9 5 10 11
Iterative deepening: 1; 1 2 3; 1 2 4 5 3 6 7; 1 2 4 8 9 5 10 11

c. Bidirectional search is very useful, because the only successor of n in the reverse direction is $\lfloor (n/2) \rfloor$. This helps focus the search.

d. 2 in the forward direction; 1 in the reverse direction.

e. Yes; start at the goal, and apply the single reverse successor action until you reach 1.

3.9

a. Here is one possible representation: A state is a six-tuple of integers listing the number of missionaries, cannibals, and boats on the first side, and then the seond side of the river. The goal is a state with 3 missionaries and 3 cannibals on the second side. The cost function is one per action, and the successors of a state are all the states that move 1 or 2 people and 1 boat from one side to another.

b. The search space is small, so any optimal algorithm works. For an example, see the file `"search/domains/cannibals.lisp"`. It suffices to eliminate moves that circle back to the state just visited. From all but the first and last states, there is only one other choice.

c. It is not obvious that almost all moves are either illegal or revert to the previous state. There is a feeling of a large branching factor, and no clear way to proceed.

3.10 For the 8 puzzle, there shouldn't be much difference in performance. Indeed, the file `"search/domains/puzzle8.lisp"` shows that you can represent an 8 puzzle state as a single 32-bit integer, so the question of modifying or copying data is moot. But for the $n \times n$ puzzle, as n increases, the advantage of modifying rather than copying grows. The disadvantage of a modifying successor function is that it only works with depth-first search (or with a variant such as iterative deepening).

3.11 **a.** The algorithm expands nodes in order of increasing path cost; therefore the first goal it encounters will be the goal with the cheapest cost.

b. It will be the same as iterative deepening, d iterations, in which $O(b^d)$ nodes are generated.

c. d/ϵ

d. Implementation not shown.

3.12 If there are two paths from the start node to a given node, discarding the more expensive one cannot eliminate any optimal solution. Uniform-cost search and breadth-first search with constant step costs both expand paths in order of g-cost. Therefore, if the current node has been expanded previously, the current path to it must be more expensive than the previously found path and it is correct to discard it.

For IDS, it is easy to find an example with varying step costs where the algorithm returns a suboptimal solution: simply have two paths to the goal, one with one step costing 3 and the other with two steps costing 1 each.

3.13 Consider a domain in which every state has a single successor, and there is a single goal at depth n. Then depth-first search will find the goal in n steps, whereas iterative deepening search will take $1 + 2 + 3 + \cdots + n = O(n^2)$ steps.

3.14 As an ordinary person (or agent) browsing the web, we can only generarte the successors of a page by visiting it. We can then do breadth-first search, or perhaps best-search search where the heuristic is some function of the number of words in common between the start and goal pages; this may help keep the links on target. Search engines keep the complete graph of the web, and may provide the user access to all (or at least some) of the pages that link to a page; this would allow us to do bidirectional search.

3.15

a. If we consider all (x, y) points, then there are an infinite number of states, and of paths.

b. (For this problem, we consider the start and goal points to be vertices.) The shortest distance between two points is a straight line, and if it is not possible to travel in a straight line because some obstacle is in the way, then the next shortest distance is a sequence of line segments, end-to-end, that deviate from the straight line by as little as possible. So the first segment of this sequence must go from the start point to a tangent point on an obstacle – any path that gave the obstacle a wider girth would be longer. Because the obstacles are polygonal, the tangent points must be at vertices of the obstacles, and hence the entire path must go from vertex to vertex. So now the state space is the set of vertices, of which there are 35 in Figure 3.22.

c. Code not shown.

d. Implementations and analysis not shown.

3.16 Code not shown.

3.17

a. Any path, no matter how bad it appears, might lead to an arbitraily large reward (negative cost). Therefore, one would need to exhaust all possible paths to be sure of finding the best one.

b. Suppose the greatest possible reward is c. Then if we also know the maximum depth of the state space (e.g. when the state space is a tree), then any path with d levels remaining can be improved by at most cd, so any paths worse than cd less than the best path can be pruned. For state spaces with loops, this guarantee doesn't help, because it is possible to go around a loop any number of times, picking up c rewward each time.

c. The agent should plan to go around this loop forever (unless it can find another loop with even better reward).

d. The value of a scenic loop is lessened each time one revisits it; a novel scenic sight is a great reward, but seeing the same one for the tenth time in an hour is tedious, not rewarding. To accomodate this, we would have to expand the state space to include a memory—a state is now represented not just by the current location, but by a current location and a bag of already-visited locations. The reward for visiting a new location is now a (diminishing) function of the number of times it has been seen before.

e. Real domains with looping behavior include eating junk food and going to class.

3.18 The belief state space is shown in Figure S3.2. No solution is possible because no path leads to a belief state all of whose elements satisfy the goal. If the problem is fully observable,

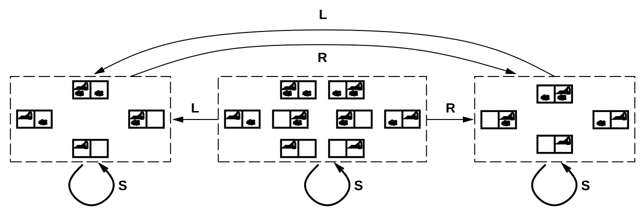

Figure S3.2 The belief state space for the sensorless vacuum world under Murphy's law.

the agent reaches a goal state by executing a sequence such that $Suck$ is performed only in a dirty square. This ensures deterministic behavior and every state is obviously solvable.

3.19 Code not shown, but a good start is in the code repository. Clearly, graph search must be used—this is a classic grid world with many alternate paths to each state. Students will quickly find that computing the optimal solution sequence is prohibitively expensive for moderately large worlds, because the state space for an $n \times n$ world has $n^2 \cdot 2^n$ states. The completion time of the random agent grows less than exponentially in n, so for any reasonable exchange rate between search cost ad path cost the random agent will eventually win.

Solutions for Chapter 4 Informed Search and Exploration

4.1 The sequence of queues is as follows:
L[0+244=244]
M[70+241=311], T[111+329=440]
L[140+244=384], D[145+242=387], T[111+329=440]
D[145+242=387], T[111+329=440], M[210+241=451], T[251+329=580]
C[265+160=425], T[111+329=440], M[210+241=451], M[220+241=461], T[251+329=580]
T[111+329=440], M[210+241=451], M[220+241=461], P[403+100=503], T[251+329=580], R[411+193=604], D[385+242=627]
M[210+241=451], M[220+241=461], L[222+244=466], P[403+100=503], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627]
M[220+241=461], L[222+244=466], P[403+100=503], L[280+244=524], D[285+242=527], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627]
L[222+244=466], P[403+100=503], L[280+244=524], D[285+242=527], L[290+244=534], D[295+242=537], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627]
P[403+100=503], L[280+244=524], D[285+242=527], M[292+241=533], L[290+244=534], D[295+242=537], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627], T[333+329=662]
B[504+0=504], L[280+244=524], D[285+242=527], M[292+241=533], L[290+244=534], D[295+242=537], T[251+329=580], A[229+366=595], R[411+193=604], D[385+242=627], T[333+329=662], R[500+193=693], C[541+160=701]

4.2 $w = 0$ gives $f(n) = 2g(n)$. This behaves exactly like uniform-cost search—the factor of two makes no difference in the *ordering* of the nodes. $w = 1$ gives A* search. $w = 2$ gives $f(n) = 2h(n)$, i.e., greedy best-first search. We also have

$$f(n) = (2 - w)[g(n) + \frac{w}{2 - w}h(n)]$$

which behaves exactly like A* search with a heuristic $\frac{w}{2-w}h(n)$. For $w \leq 1$, this is always less than $h(n)$ and hence admissible, provided $h(n)$ is itself admissible.

4.3

a. When all step costs are equal, $g(n) \propto depth(n)$, so uniform-cost search reproduces breadth-first search.

b. Breadth-first search is best-first search with $f(n) = depth(n)$; depth-first search is best-first search with $f(n) = -depth(n)$; uniform-cost search is best-first search with

$f(n) = g(n)$.

c. Uniform-cost search is A* search with $h(n) = 0$.

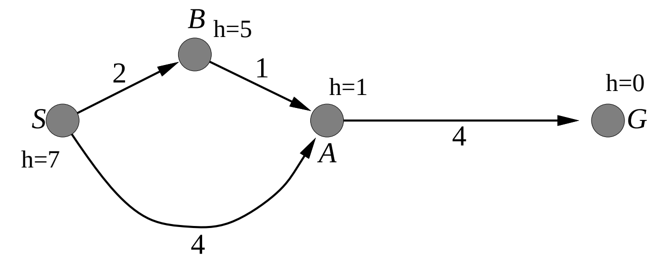

Figure S4.1 A graph with an inconsistent heuristic on which GRAPH-SEARCH fails to return the optimal solution. The successors of S are A with $f=5$ and B with $f=7$. A is expanded first, so the path via B will be discarded because A will already be in the closed list.

4.4 See Figure S4.1.

4.5 Going between Rimnicu Vilcea and Lugoj is one example. The shortest path is the southern one, through Mehadia, Dobreta and Craiova. But a greedy search using the straight-line heuristic starting in Rimnicu Vilcea will start the wrong way, heading to Sibiu. Starting at Lugoj, the heuristic will correctly lead us to Mehadia, but then a greedy search will return to Lugoj, and oscillate forever between these two cities.

4.6 The heuristic $h = h_1 + h_2$ (adding misplaced tiles and Manhattan distance) sometimes overestimates. Now, suppose $h(n) \le h^*(n) + c$ (as given) and let G_2 be a goal that is suboptimal by more than c, i.e., $g(G_2) > C^* + c$. Now consider any node n on a path to an optimal goal. We have

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &\le g(n) + h^*(n) + c \\ &\le C^* + c \\ &\le g(G_2) \end{aligned}$$

so G_2 will never be expanded before an optimal goal is expanded.

4.7 A heuristic is consistent iff, for every node n and every successor n' of n generated by any action a,

$$h(n) \le c(n, a, n') + h(n')$$

One simple proof is by induction on the number k of nodes on the shortest path to any goal from n. For $k = 1$, let n' be the goal node; then $h(n) \le c(n, a, n')$. For the inductive case, assume n' is on the shortest path k steps from the goal and that $h(n')$ is admissible by hypothesis; then

$$h(n) \le c(n, a, n') + h(n') \le c(n, a, n') + h^*(n') = h^*(n)$$

so $h(n)$ at $k+1$ steps from the goal is also admissible.

4.8 This exercise reiterates a small portion of the classic work of Held and Karp (1970).

a. The TSP problem is to find a minimal (total length) path through the cities that forms a closed loop. MST is a relaxed version of that because it asks for a minimal (total length) graph that need not be a closed loop—it can be any fully-connected graph. As a heuristic, MST is admissible—it is always shorter than or equal to a closed loop.

b. The straight-line distance back to the start city is a rather weak heuristic—it vastly underestimates when there are many cities. In the later stage of a search when there are only a few cities left it is not so bad. To say that MST dominates straight-line distance is to say that MST always gives a higher value. This is obviously true because a MST that includes the goal node and the current node must either be the straight line between them, or it must include two or more lines that add up to more. (This all assumes the triangle inequality.)

c. See `"search/domains/tsp.lisp"` for a start at this. The file includes a heuristic based on connecting each unvisited city to its nearest neighbor, a close relative to the MST approach.

d. See (Cormen *et al.*, 1990, p.505) for an algorithm that runs in $O(E \log E)$ time, where E is the number of edges. The code repository currently contains a somewhat less efficient algorithm.

4.9 The misplaced-tiles heuristic is exact for the problem where a tile can move from square A to square B. As this is a relaxation of the condition that a tile can move from square A to square B if B is blank, Gaschnig's heuristic canot be less than the misplaced-tiles heuristic. As it is also admissible (being exact for a relaxation of the original problem), Gaschnig's heuristic is therefore more accurate.

If we permute two adjacent tiles in the goal state, we have a state where misplaced-tiles and Manhattan both return 2, but Gaschnig's heuristic returns 3.

To compute Gaschnig's heuristic, repeat the following until the goal state is reached: let B be the current location of the blank; if B is occupied by tile X (not the blank) in the goal state, move X to B; otherwise, move any misplaced tile to B. Students could be asked to prove that this is the optimal solution to the relaxed problem.

4.11

a. Local beam search with $k = 1$ is hill-climbing search.

b. Local beam search with $k = \infty$: strictly speaking, this doesn't make sense. (Exercise may be modified in future printings.) The idea is that if every successor is retained (because k is unbounded), then the search resembles breadth-first search in that it adds one complete layer of nodes before adding the next layer. Starting from one state, the algorithm would be essentially identical to breadth-first search except that each layer is generated all at once.

c. Simulated annealing with $T = 0$ at all times: ignoring the fact that the termination step would be triggered immediately, the search would be identical to first-choice hill climbing because every downward successor would be rejected with probability 1. (Exercise may be modified in future printings.)

d. Genetic algorithm with population size $N = 1$: if the population size is 1, then the two selected parents will be the same individual; crossover yields an exact copy of the individual; then there is a small chance of mutation. Thus, the algorithm executes a random walk in the space of individuals.

4.12 If we assume the comparison function is transitive, then we can always sort the nodes using it, and choose the node that is at the top of the sort. Efficient priority queue data structures rely only on comparison operations, so we lose nothing in efficiency—except for the fact that the comparison operation on states may be much more expensive than comparing two numbers, each of which can be computed just once.

A* relies on the division of the total cost estimate $f(n)$ into the cost-so-far and the cost-to-go. If we have comparison operators for each of these, then we can prefer to expand a node that is better than other nodes on both comparisons. Unfortunately, there will usually be no such node. The tradeoff between $g(n)$ and $h(n)$ cannot be realized without numerical values.

4.13 The space complexity of LRTA* is dominated by the space required for $result[a, s]$, i.e., the product of the number of states visited (n) and the number of actions tried per state (m). The time complexity is at least $O(nm^2)$ for a naive implementation because for each action taken we compute an H value, which requires minimizing over actions. A simple optimization can reduce this to $O(nm)$. This expression assumes that each state–action pair is tried at most once, whereas in fact such pairs may be tried many times, as the example in Figure 4.22 shows.

4.14 This question is slightly ambiguous as to what the percept is—either the percept is just the location, or it gives exactly the set of unblocked directions (i.e., blocked directions are illegal actions). We will assume the latter. (Exercise may be modified in future printings.) There are 12 possible locations for internal walls, so there are $2^{12} = 4096$ possible environment configurations. A belief state designates a *subset* of these as possible configurations; for example, before seeing any percepts all 4096 configurations are possible—this is a single belief state.

a. We can view this as a contingency problem in belief state space. The initial belief state is the set of all 4096 configurations. The total belief state space contains 2^{4096} belief states (one for each possible subset of configurations, but most of these are not reachable. After each action and percept, the agent learns whether or not an internal wall exists between the current square and each neighboring square. Hence, each reachable belief state can be represnted exactly by a list of status values (present, absent, unknown) for each wall separately. That is, the belief state is completely decomposable and there are exactly 3^{12} reachable belief states. The maximum number of possible wall-percepts in each state is 16 (2^4), so each belief state has four actions, each with up to 16 nondeterministic successors.

b. Assuming the external walls are known, there are two internal walls and hence $2^2 = 4$ possible percepts.

c. The initial null action leads to four possible belief states, as shown in Figure S4.2. From each belief state, the agent chooses a single action which can lead to up to 8 belief states (on entering the middle square). Given the possibility of having to retrace its steps at a dead end, the agent can explore the entire maze in no more than 18 steps, so the complete plan (expressed as a tree) has no more than 8^{18} nodes. On the other hand, there are just 3^{12}, so the plan could be expressed more concisely as a table of actions indexed by belief state (a **policy** in the terminology of Chapter 17).

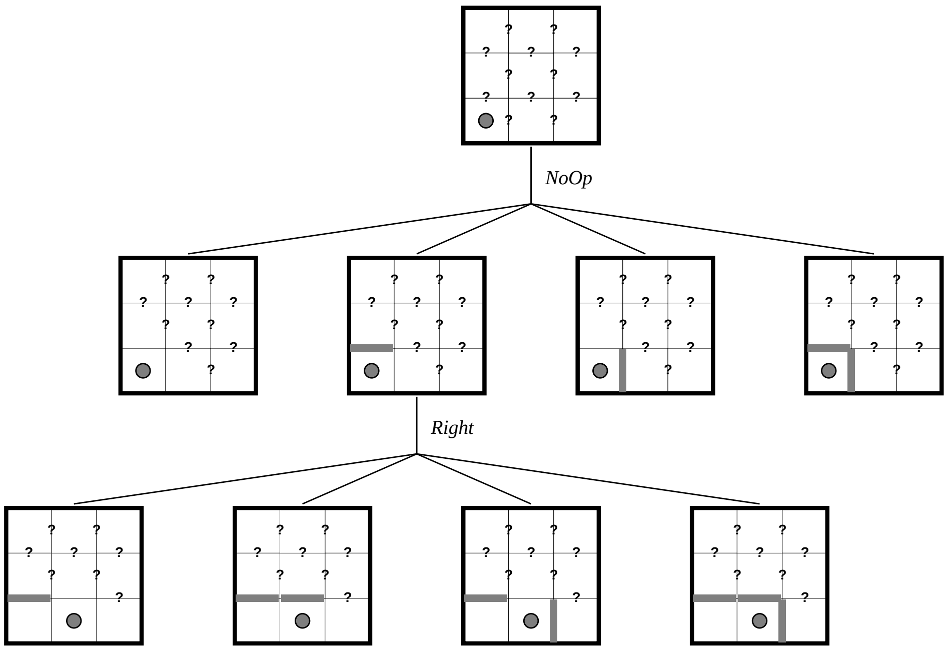

Figure S4.2 The 3×3 maze exploration problem: the initial state, first percept, and one selected action with its perceptual outcomes.

4.15 Here is one simple hill-climbing algorithm:

- Connect all the cities into an arbitrary path.
- Pick two points along the path at random.
- Split the path at those points, producing three pieces.
- Try all six possible ways to connect the three pieces.
- Keep the best one, and reconnect the path accordingly.
- Iterate the steps above until no improvement is observed for a while.

4.1 4.16 Code not shown.

4.17 Hillclimbing is surprisingly effective at finding reasonable if not optimal paths for very little computational cost, and seldom fails in two dimensions.

a. It is possible (see Figure S4.3(a)) but very unlikely—the obstacle has to have an unusual shape and be positioned correctly with respect to the goal.

b. With convex obstacles, getting stuck is much more likely to be a problem (see Figure S4.3(b)).

c. Notice that this is just depth-limited search, where you choose a step along the best path even if it is not a solution.

d. Set k to the maximum number of sides of any polygon and you can always escape.

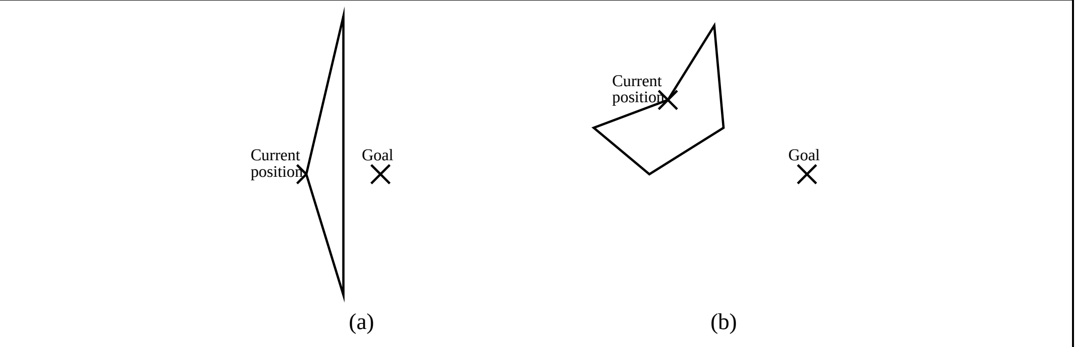

Figure S4.3 (a) Getting stuck with a convex obstacle. (b) Getting stuck with a nonconvex obstacle.

4.18 The student should find that on the 8-puzzle, RBFS expands more nodes (because it does not detect repeated states) but has lower cost per node because it does not need to maintain a queue. The number of RBFS node re-expansions is not too high because the presence of many tied values means that the best path changes seldom. When the heuristic is slightly perturbed, this advantage disappears and RBFS's performance is much worse.

For TSP, the state space is a tree, so repeated states are not an issue. On the other hand, the heuristic is real-valued and there are essentially no tied values, so RBFS incurs a heavy penalty for frequent re-expansions.

Solutions for Chapter 5
Constraint Satisfaction Problems

5.1 A **constraint satisfaction problem** is a problem in which the goal is to choose a value for each of a set of variables, in such a way that the values all obey a set of constraints.

A **constraint** is a restriction on the possible values of two or more variables. For example, a constraint might say that $A = a$ is not allowed in conjunction with $B = b$.

Backtracking search is a form depth-first search in which there is a single representation of the state that gets updated for each successor, and then must be restored when a dead end is reached.

A directed arc from variable A to variable B in a CSP is **arc consistent** if, for every value in the current domain of A, there is some consistent value of B.

Backjumping is a way of making backtracking search more efficient, by jumping back more than one level when a dead end iss reached.

Min-conflicts is a heuristic for use with local search on CSP problems. The heuristic says that, when given a variable to modify, choose the value that conflicts with the fewest number of other variables.

5.2 There are 18 solutions for coloring Australia with three colors. Start with *SA*, which can have any of three colors. Then moving clockwise, *WA* can have either of the other two colors, and everything else is strictly determined; that makes 6 possibilities for the mainland, times 3 for Tasmania yields 18.

5.3 The most constrained variable makes sense because it chooses a variable that is (all other things being equal) likely to cause a failure, and it is more efficient to fail as early as possible (thereby pruning large parts of the search space). The least constraining value heuristic makes sense because it allows the most chances for future assignments to avoid conflict.

5.4 **a.** Crossword puzzle construction can be solved many ways. One simple choice is depth-first search. Each successor fills in a word in the puzzle with one of the words in the dictionary. It is better to go one word at a time, to minimize the number of steps.

b. As a CSP, there are even more choices. You could have a variable for each box in the crossword puzzle; in this case the value of each variable is a letter, and the constraints are that the letters must make words. This approach is feasible with a most-constraining value heuristic. Alternately, we could have each string of consecutive horizontal or vertical boxes be a single variable, and the domain of the variables be words in the dictionary of the right length. The constraints would say that two intersecting words must have the same letter in the intersecting box. Solving a problem in this formulation requires fewer steps, but the domains are larger (assuming a big dictionary) and there are fewer constraints. Both formulations are feasible.

5.5 **a.** For rectilinear floor-planning, one possibility is to have a variable for each of the small rectangles, with the value of each variable being a 4-tuple consisting of the x and y coordinates of the upper left and lower right corners of the place where the rectangle will be located. The domain of each variable is the set of 4-tuples that are the right size for the corresponding small rectangle and that fit within the large rectangle. Constraints say that no two rectangles can overlap; for example if the value of variable R_1 is $[0,0,5,8]$, then no other variable can take on a value that overlaps with the $0,0$ to $5,8$ rectangle.

b. For class scheduling, one possibility is to have three variables for each class, one with times for values (e.g. MWF8:00, TuTh8:00, MWF9:00, ...), one with classrooms for values (e.g. Wheeler110, Evans330, ...) and one with instructors for values (e.g. Abelson, Bibel, Canny, ...). Constraints say that only one class can be in the same classroom at the same time, and an instructor can only teach one class at a time. There may be other constraints as well (e.g. an instructor should not have two consecutive classes).

5.6 The exact steps depend on certain choices you are free to make; here are the ones I made:

a. Choose the X_3 variable. Its domain is $\{0,1\}$.

b. Choose the value 1 for X_3. (We can't choose 0; it wouldn't survive forward checking, because it would force F to be 0, and the leading digit of the sum must be non-zero.)

c. Choose F, because it has only one remaining value.

d. Choose the value 1 for F.

e. Now X_2 and $\mathbf{X}_1$ are tied for minimum remaining values at 2; let's choose X_2.

f. Either value survives forward checking, let's choose 0 for X_2.

g. Now X_1 has the minimum remaining values.

h. Again, arbitrarily choose 0 for the value of X_1.

i. The variable O must be an even number (because it is the sum of $T+T$ less than 5 (because $O+O=R+10\times 0$). That makes it most constrained.

j. Arbitrarily choose 4 as the value of O.

k. R now has only 1 remaining value.

l. Choose the value 8 for R.

m. T now has only 1 remianing value.

n. Choose the value 7 for T.

o. U must be an even number less than 9; choose U.

p. The only value for U that survives forward checking is 6.

q. The only variable left is W.

r. The only value left for W is 3.

s. This is a solution.

This is a rather easy (under-constrained) puzzle, so it is not surprising that we arrive at a solution with no backtracking (given that we are allowed to use forward checking).

5.7 There are implementations of CSP algorithms in the Java, Lisp, and Python sections of the online code repository; these should help students get started. However, students will have to add code to keep statistics on the experiments, and perhaps will want to have some mechanism for making an experiment return failure if it exceeds a certain time limit (or number-of-steps limit). The amount of code that needs to be written is small; the exercise is more about running and analyzing an experiment.

5.8 We'll trace through each iteration of the **while** loop in AC-3 (for one possible ordering of the arcs):

a. Remove $SA - WA$, delete R from SA.
b. Remove $SA - V$, delete B from SA, leaving only G.
c. Remove $NT - WA$, delete R from NT.
d. Remove $NT - SA$, delete G from NT, leaving only B.
e. Remove $NSW - SA$, delete G from NSW.
f. Remove $NSW - V$, delete B from NSW, leaving only R.
g. Remove $Q - NT$, delete B from Q.
h. Remove $Q - SA$, delete G from Q.
i. remove $Q - NSW$, delete R from Q, leaving no domain for Q.

5.9 On a tree-structured graph, no arc will be considered more than once, so the AC-3 algorithm is $O(ED)$, where E is the number of edges and D is the size of the largest domain.

5.10 The basic idea is to preprocess the constraints so that, for each value of X_i, we keep track of those variables X_k for which an arc from X_k to X_i is satisfied by that particular value of X_i. This data structure can be computed in time proportional to the size of the problem representation. Then, when a value of X_i is deleted, we reduce by 1 the count of allowable values for each (X_k, X_i) arc recorded under that value. This is very similar to the forward chaining algorithm in Chapter 7. See ? (?) for detailed proofs.

5.11 The problem statement sets out the solution fairly completely. To express the ternary constraint on A, B and C that $A + B = C$, we first introduce a new variable, AB. If the domain of A and B is the set of numbers N, then the domain of AB is the set of pairs of numbers from N, i.e. $N \times N$. Now there are three binary constraints, one between A and AB saying that the value of A must be equal to the first element of the pair-value of AB; one between B and AB saying that the value of B must equal the second element of the value of AB; and finally one that says that the sum of the pair of numbers that is the value of AB must equal the value of C. All other ternary constraints can be handled similarly.

Now that we can reduce a ternary constraint into binary constraints, we can reduce a 4-ary constraint on variables A, B, C, D by first reducing A, B, C to binary constraints as shown above, then adding back D in a ternary constraint with AB and C, and then reducing this ternary constraint to binary by introducing CD.

By induction, we can reduce any n-ary constraint to an $(n-1)$-ary constraint. We can stop at binary, because any unary constraint can be dropped, simply by moving the effects of the constraint into the domain of the variable.

5.12 A simple algorithm for finding a cutset of no more than k nodes is to enumerate all subsets of nodes of size $1, 2, \ldots, k$, and for each subset check whether the remaining nodes form a tree. This algorithm takes time $\binom{\sum_{i=1}^{k} n}{nk}$, which is $O(n^k)$.

Becker and Geiger (1994; http://citeseer.nj.nec.com/becker94approximation.html) give an algorithm called MGA (modified greedy algorithm) that finds a cutset that is no more than twice the size of the minimal cutset, using time $O(E + V\log(V))$, where E is the number of edges and V is the number of variables.

Whether this makes the cycle cutset approach practical depends more on the graph involved than on the agorithm for finding a cutset. That is because, for a cutset of size c, we still have an exponential (d^c) factor before we can solve the CSP. So any graph with a large cutset will be intractible to solve, even if we could find the cutset with no effort at all.

5.13 The “Zebra Puzzle” can be represented as a CSP by introducing a variable for each color, pet, drink, country and cigaret brand (a total of 25 variables). The value of each variable is a number from 1 to 5 indicating the house number. This is a good representation because it easy to represent all the constraints given in the problem definition this way. (We have done so in the Python implementation of the code, and at some point we may reimplement this in the other languages.) Besides ease of expressing a problem, the other reason to choose a representation is the efficiency of finding a solution. here we have mixed results—on some runs, min-conflicts local search finds a solution for this problem in seconds, while on other runs it fails to find a solution after minutes.

Another representation is to have five variables for each house, one with the domain of colrs, one with pets, and so on.

Solutions for Chapter 6
Adversarial Search

6.1 Figure S6.1 shows the game tree, with the evaluation function values below the terminal nodes and the backed-up values to the right of the non-terminal nodes. The values imply that the best starting move for X is to take the center. The terminal nodes with a bold outline are the ones that do not need to be evaluated, assuming the optimal ordering.

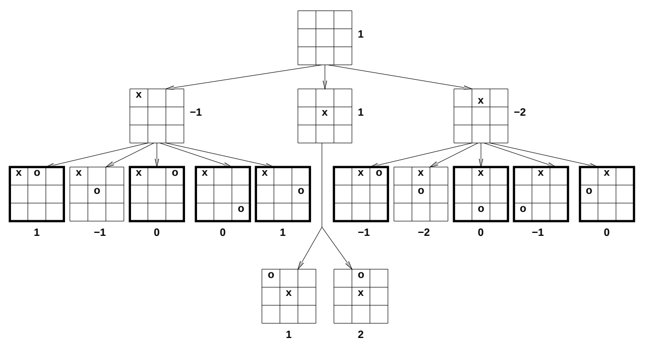

Figure S6.1 Part of the game tree for tic-tac-toe, for Exercise 6.1.

6.2 Consider a MIN node whose children are terminal nodes. If MIN plays suboptimally, then the value of the node is greater than or equal to the value it would have if MIN played optimally. Hence, the value of the MAX node that is the MIN node's parent can only be increased. This argument can be extended by a simple induction all the way to the root. *If the suboptimal play by* MIN *is predictable*, then one can do better than a minimax strategy. For example, if MIN always falls for a certain kind of trap and loses, then setting the trap guarantees a win even if there is actually a devastating response for MIN. This is shown in Figure S6.2.

6.3

a. (5) The game tree, complete with annotations of all minimax values, is shown in Figure S6.3.

b. (5) The "?" values are handled by assuming that an agent with a choice between winning the game and entering a "?" state will always choose the win. That is, min(–1,?) is –1 and max(+1,?) is +1. If all successors are "?", the backed-up value is "?".

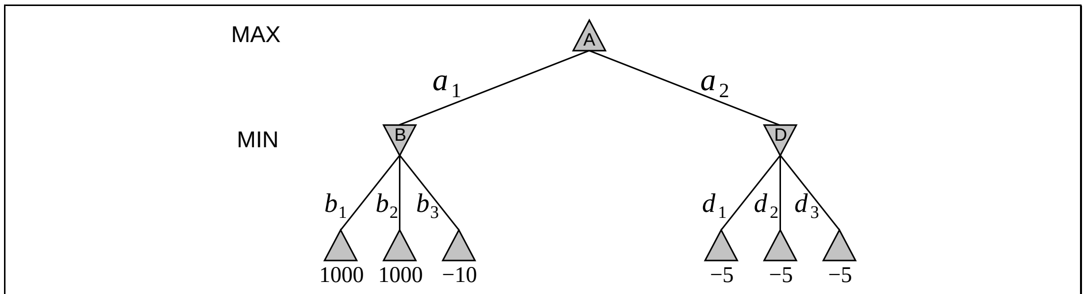

Figure S6.2 A simple game tree showing that setting a trap for MIN by playing a_i is a win if MIN falls for it, but may also be disastrous. The minimax move is of course a_2, with value -5.

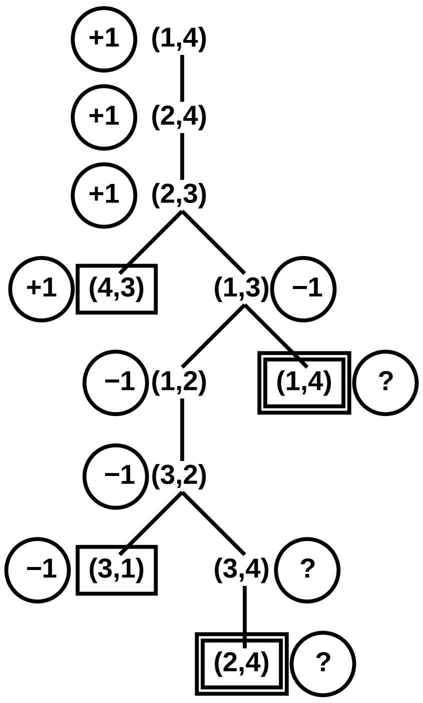

Figure S6.3 The game tree for the four-square game in Exercise 6.3. Terminal states are in single boxes, loop states in double boxes. Each state is annotated with its minimax value in a circle.

c. (5) Standard minimax is depth-first and would go into an infinite loop. It can be fixed by comparing the current state against the stack; and if the state is repeated, then return a “?” value. Propagation of “?” values is handled as above. Although it works in this case, it does not *always* work because it is not clear how to compare “?” with a drawn position; nor is it clear how to handle the comparison when there are wins of different degrees (as in backgammon). Finally, in games with chance nodes, it is unclear how to compute the average of a number and a "?". Note that it is *not* correct to treat repeated states automatically as drawn positions; in this example, both (1,4) and (2,4) repeat in the tree but they are won positions.

What is really happening is that each state has a well-defined but initially unknown value. These unknown values are related by the minimax equation at the bottom of page 163. If the game tree is acyclic, then the minimax algorithm solves these equations by propagating from the leaves. If the game tree has cycles, then a dynamic programming method must be used, as explained in Chapter 17. (Exercise 17.8 studies this problem in particular.) These algorithms can determine whether each node has a well-determined value (as in this example) or is really an infinite loop in that both players prefer to stay in the loop (or have no choice). In such a case, the rules of the game will need to define the value (otherwise the game will never end). In chess, for eaxmple, a state that occurs 3 times (and hence is assumed to be desirable for both players) is a draw.

d. This question is a little tricky. One approach is a proof by induction on the size of the game. Clearly, the base case $n=3$ is a loss for A and the base case $n=4$ is a win for A. For any $n > 4$, the initial moves are the same: A and B both move one step towards each other. Now, we can see that they are engaged in a subgame of size $n-2$ on the squares $[2, \ldots, n-1]$, *except* that there is an extra choice of moves on squares 2 and $n-1$. Ignoring this for a moment, it is clear that if the "$n-2$" is won for A, then A gets to the square $n-1$ before B gets to square 2 (by the definition of winning) and therefore gets to n before B gets to 1, hence the "n" game is won for A. By the same line of reasoning, if "$n-2$" is won for B then "n" is won for B. Now, the presence of the extra moves complicates the issue, but not too much. First, the player who is slated to win the subgame $[2, \ldots, n-1]$ never moves back to his home square. If the player slated to lose the subgame does so, then it is easy to show that he is bound to lose the game itself—the other player simply moves forward and a subgame of size $n-2k$ is played one step closer to the loser's home square.

6.4 See `"search/algorithms/games.lisp"` for definitions of games, game-playing agents, and game-playing environments. `"search/algorithms/minimax.lisp"` contains the minimax and alpha-beta algorithms. Notice that the game-playing environment is essentially a generic environment with the update function defined by the rules of the game. Turn-taking is achieved by having agents do nothing until it is their turn to move.

See `"search/domains/cognac.lisp"` for the basic definitions of a simple game (slightly more challenging than Tic-Tac-Toe). The code for this contains only a trivial evaluation function. Students can use minimax and alpha-beta to solve small versions of the game to termination (probably up to 4×3); they should notice that alpha-beta is far faster than minimax, but still cannot scale up without an evaluation function and truncated horizon. Providing an evaluation function is an interesting exercise. From the point of view of data structure design, it is also interesting to look at how to speed up the legal move generator by precomputing the descriptions of rows, columns, and diagonals.

Very few students will have heard of kalah, so it is a fair assignment, but the game is boring—depth 6 lookahead and a purely material-based evaluation function are enough to beat most humans. Othello is interesting and about the right level of difficulty for most students. Chess and checkers are sometimes unfair because usually a small subset of the class will be experts while the rest are beginners.

6.5 This question is not as hard as it looks. The derivation below leads directly to a definition of α and β values. The notation n_i refers to (the value of) the node at depth i on the path from the root to the leaf node n_j. Nodes $n_{i1} \ldots n_{ib_i}$ are the siblings of node i.

a. We can write $n_2 = \max(n_3, n_{31}, \ldots, n_{3b_3})$, giving

$$n_1 = \min(\max(n_3, n_{31}, \ldots, n_{3b_3}), n_{21}, \ldots, n_{2b_2})$$

Then n_3 can be similarly replaced, until we have an expression containing n_j itself.

b. In terms of the l and r values, we have

$$n_1 = \min(l_2, \max(l_3, n_3, r_3), r_2)$$

Again, n_3 can be expanded out down to n_j. The most deeply nested term will be $\min(l_j, n_j, r_j)$.

c. If n_j is a max node, then the lower bound on its value only increases as its successors are evaluated. Clearly, if it exceeds l_j it will have no further effect on n_1. By extension, if it exceeds $\min(l_2, l_4, \ldots, l_j)$ it will have no effect. Thus, by keeping track of this value we can decide when to prune n_j. This is exactly what α-β does.

d. The corresponding bound for min nodes n_k is $\max(l_3, l_5, \ldots, l_k)$.

6.7 The general strategy is to reduce a general game tree to a one-ply tree by induction on the depth of the tree. The inductive step must be done for min, max, and chance nodes, and simply involves showing that the transformation is carried though the node. Suppose that the values of the descendants of a node are $x_1 \ldots x_n$, and that the transformation is $ax + b$, where a is positive. We have

$$\begin{aligned} \min(ax_1 + b, ax_2 + b, \ldots, ax_n + b) &= a\min(x_1, x_2, \ldots, x_n) + b \\ \max(ax_1 + b, ax_2 + b, \ldots, ax_n + b) &= a\min(x_1, x_2, \ldots, x_n) + b \\ p_1(ax_1 + b) + p_2(ax_2 + b) + \cdots + p_n(ax_n + b) &= a(p_1x_1 + p_2x_2 + \cdots p_nx_n) + b \end{aligned}$$

Hence the problem reduces to a one-ply tree where the leaves have the values from the original tree multiplied by the linear transformation. Since $x > y \Rightarrow ax + b > ay + b$ if $a > 0$, the best choice at the root will be the same as the best choice in the original tree.

6.8 This procedure will give incorrect results. Mathematically, the procedure amounts to assuming that averaging commutes with min and max, which it does not. Intuitively, the choices made by each player in the deterministic trees are based on full knowledge of future dice rolls, and bear no necessary relationship to the moves made without such knowledge. (Notice the connection to the discussion of card games on page 179 and to the general problem of fully and partially observable Markov decision problems in Chapter 17.) In practice, the method works reasonably well, and it might be a good exercise to have students compare it to the alternative of using expectiminimax with sampling (rather than summing over) dice rolls.

6.9 Code not shown.

6.10 The basic physical state of these games is fairly easy to describe. One important thing to remember for Scrabble and bridge is that the physical state is not accessible to all players and so cannot be provided directly to each player by the environment simulator. Particularly in bridge, each player needs to maintain some best guess (or multiple hypotheses) as to the actual state of the world. We expect to be putting some of the game implementations online as they become available.

6.11 One can think of chance events during a game, such as dice rolls, in the same way as hidden but preordained information (such as the order of the cards in a deck). The key distinctions are whether the players can influence what information is revealed and whether there is any asymmetry in the information available to each player.

a. Expectiminimax is appropriate only for backgammon and Monopoly. In bridge and Scrabble, each player knows the cards/tiles he or she possesses but not the opponents'. In Scrabble, the benefits of a fully rational, randomized strategy that includes reasoning about the opponents' state of knowledge are probably small, but in bridge the questions of knowledge and information disclosure are central to good play.

b. None, for the reasons described earlier.

c. Key issues include reasoning about the opponent's beliefs, the effect of various actions on those beliefs, and methods for representing them. Since belief states for rational agents are probability distributions over all possible states (including the belief states of others), this is nontrivial.

```
function MAX-VALUE($state$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow -\infty$
  for $a,\ s$ in SUCCESSORS($state$) do
    if WINNER($s$) = MAX
        then $v \leftarrow$ MAX(v, MAX-VALUE($s$))
        else $v \leftarrow$ MAX(v, MIN-VALUE($s$))
  return $v$
```

Figure S6.4 Part of the modified minimax algorithm for games in which the winner of the previous trick plays first on the next trick.

6.12 (In the first printing, this exericse refers to WINNER($trick$); subsequent printings refer to WINNER(s), denoting the winner of the trick just completed (if any), or null.) This question is interpreted as applying only to the observable case.

a. The modification to MAX-VALUE is shown in Figure S6.4. If MAX has just won a trick, MAX gets to play again, otherwise play alternates. Thus, the successors of a MAX node

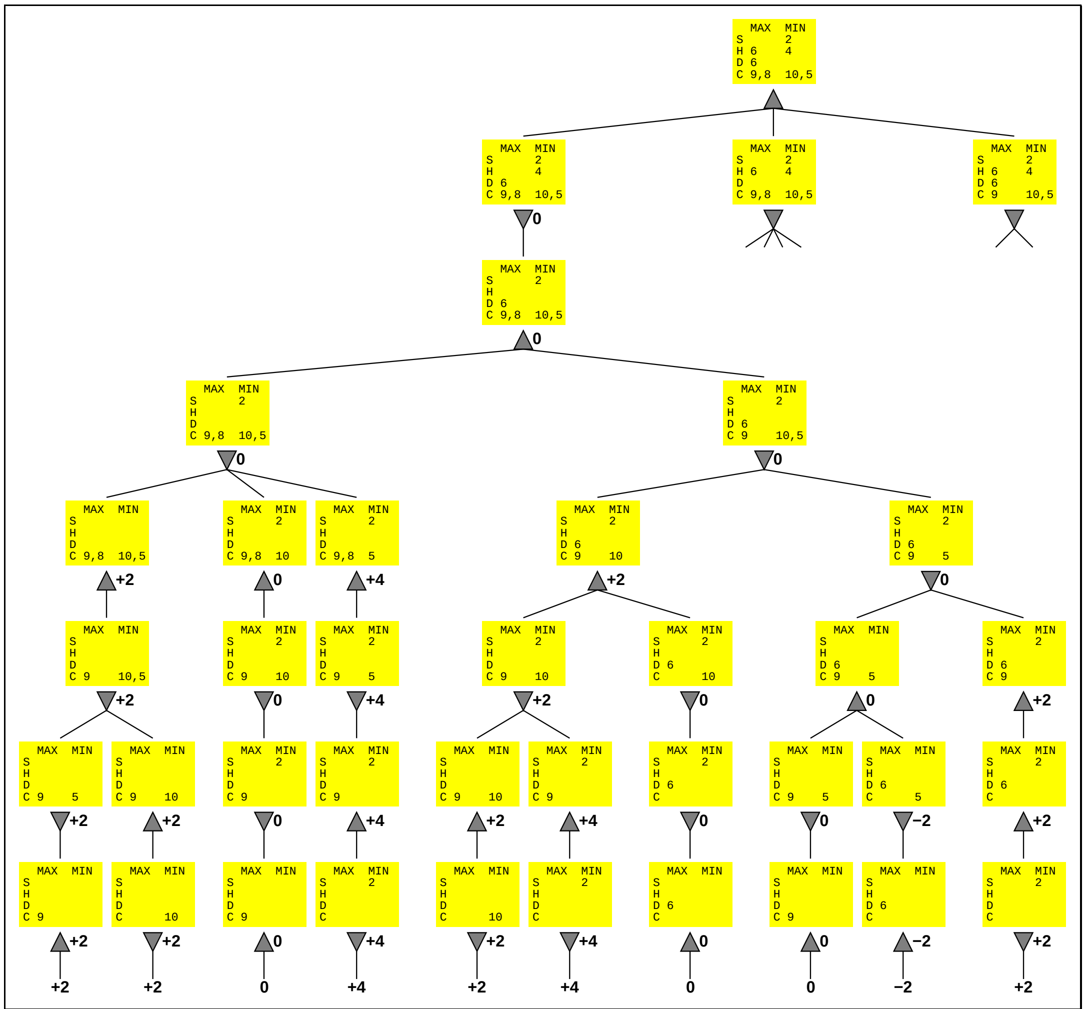

Figure S6.5 Ex. 6.12: Part of the game tree for the card game shown on p.179.

can be a mixture of MAX and MIN nodes, depending on the various cards MAX can play. A similar modification is needed for MIN-VALUE.

b. The game tree is shown in Figure S6.5.

6.13 The naive approach would be to generate each such position, solve it, and store the outcome. This would be enormously expensive—roughly on the order of 444 billion seconds, or 10,000 years, assuming it takes a second on average to solve each position (which is probably very optimistic). Of course, we can take advantage of already-solved positions when solving new positions, provided those solved positions are descendants of the new positions. To ensure that this *always* happens, we generate the *final* positions first, then their *predecessors*, and so on. In this way, the exact values of all successors are known when each state is generated. This method is called **retrograde analysis**.

RETROGRADE ANALYSIS

6.14 The most obvious change is that the space of actions is now continuous. For example, in pool, the cueing direction, angle of elevation, speed, and point of contact with the cue ball are all continuous quantities.

The simplest solution is just to discretize the action space and then apply standard methods. This might work for tennis (modelled crudely as alternating shots with speed and direction), but for games such as pool and croquet it is likely to fail miserably because small changes in direction have large effects on action outcome. Instead, one must analyze the game to identify a discrete set of meaningful local goals, such as "potting the 4-ball" in pool or "laying up for the next hoop" in croquet. Then, in the current context, a local optimization routine can work out the best way to achieve each local goal, resulting in a discrete set of possible choices. Typically, these games are stochastic, so the backgammon model is appropriate provided that we use sampled outcomes instead of summing over all outcomes.

Whereas pool and croquet are modelled correctly as turn-taking games, tennis is not. While one player is moving to the ball, the other player is moving to anticipate the opponent's return. This makes tennis more like the simultaneous-action games studied in Chapter 17. In particular, it may be reasonable to derive *randomized* strategies so that the opponent cannot anticipate where the ball will go.

6.15 The minimax algorithm for non-zero-sum games works exactly as for multiplayer games, described on p.165–6; that is, the evaluation function is a vector of values, one for each player, and the backup step selects whichever vector has the highest value for the player whose turn it is to move. The example at the end of Section 6.2 (p.167) shows that alpha-beta pruning is not possible in general non-zero-sum games, because an unexamined leaf node might be optimal for both players.

6.16 With 32 pieces, each needing 6 bits to specify its position on one of 64 squares, we need 24 bytes (6 32-bit words) to store a position, so we can store roughly 20 million positions in the table (ignoring pointers for hash table bucket lists). This is about one-ninth of the 180 million positions generated during a three-minute search.

Generating the hash key directly from an array-based representation of the position might be quite expensive. Modern programs (see, e.g., Heinz, 2000) carry along the hash key and modify it as each new position is generated. Suppose this takes on the order of 20 operations; then on a 2GHz machine where an evaluation takes 2000 operations we can do roughly 100 lookups per evaluation. Using a rough figure of one millisecond for a disk seek, we could do 1000 evaluations per lookup. Clearly, using a disk-resident table is of dubious value, even if we can get some locality of reference to reduce the number of disk reads.

Solutions for Chapter 7
Agents that Reason Logically

7.1 The wumpus world is partially observable, deterministic, sequential (you need to remember the state of one location when you return to it on a later turn), static, discrete, and single agent (the wumpus's sole trick—devouring an errant explorer—is not enough to treat it as an agent). Thus, it is a fairly simple environment. The main complication is the partial observability.

7.2 To save space, we'll show the list of models as a table rather than a collection of diagrams. There are eight possible combinations of pits in the three squares, and four possibilities for the wumpus location (including nowhere).

We can see that $KB \models \alpha_2$ because every line where KB is true also has α_2 true. Similarly for α_3.

7.3

a. There is a `pl_true` in the Python code, and a version of `ask` in the Lisp code that serves the same purpose. The Java code did not have this function as of May 2003, but it should be added soon.)

b. The sentences $True$, $P \vee \neg P$, and $P \wedge \neg P$ can all be determined to be true or false in a partial model that does not specify the truth value for P.

c. It is possible to create two sentences, each with k variables that are not instantiated in the partial model, such that one of them is true for all 2^k possible values of the variables, while the other sentence is false for one of the 2^k values. This shows that in general one must consider all 2^k possibilities. Enumerating them takes exponential time.

d. The python implementation of `pl_true` returns true if any disjunct of a disjunction is true, and false if any conjunct of a conjunction is false. It will do this even if other disjuncts/conjuncts contains uninstantiated variables. Thus, in the partial model where P is true, $P \vee Q$ returns true, and $\neg P \wedge Q$ returns false. But the truth values of $Q \vee \neg Q$, $Q \vee True$, and $Q \wedge \neg Q$ are not detected.

e. Our version of `tt_entails` already uses this modified `pl_true`. It would be slower if it did not.

7.4 Remember, $\alpha \models \beta$ iff in very model in which α is true, β is also true. Therefore,

a. A *valid* sentence is one that is true in all models. The sentence $True$ is also valid in all models. So if $alpha$ is valid then the entailment holds (because both $True$ and α hold

Model	KB	α_2	α_3
		$true$	
$P_{1,3}$		$true$	
$P_{2,2}$			
$P_{3,1}$		$true$	
$P_{1,3}$, $P_{2,2}$			
$P_{2,2}$, $P_{3,1}$			
$P_{3,1}$, $P_{1,3}$		$true$	
$P_{1,3}$, $P_{3,1}$, $P_{2,2}$			
$W_{1,3}$		$true$	$true$
$W_{1,3}$, $P_{1,3}$		$true$	$true$
$W_{1,3}$, $P_{2,2}$			$true$
$W_{1,3}$, $P_{3,1}$	$true$	$true$	$true$
$W_{1,3}$, $P_{1,3}$, $P_{2,2}$			$true$
$W_{1,3}$, $P_{2,2}$, $P_{3,1}$			$true$
$W_{1,3}$, $P_{3,1}$, $P_{1,3}$		$true$	$true$
$W_{1,3}$, $P_{1,3}$, $P_{3,1}$, $P_{2,2}$			$true$
$W_{3,1}$,		$true$	
$W_{3,1}$, $P_{1,3}$		$true$	
$W_{3,1}$, $P_{2,2}$			
$W_{3,1}$, $P_{3,1}$		$true$	
$W_{3,1}$, $P_{1,3}$, $P_{2,2}$			
$W_{3,1}$, $P_{2,2}$, $P_{3,1}$			
$W_{3,1}$, $P_{3,1}$, $P_{1,3}$		$true$	
$W_{3,1}$, $P_{1,3}$, $P_{3,1}$, $P_{2,2}$			
$W_{2,2}$		$true$	
$W_{2,2}$, $P_{1,3}$		$true$	
$W_{2,2}$, $P_{2,2}$			
$W_{2,2}$, $P_{3,1}$		$true$	
$W_{2,2}$, $P_{1,3}$, $P_{2,2}$			
$W_{2,2}$, $P_{2,2}$, $P_{3,1}$			
$W_{2,2}$, $P_{3,1}$, $P_{1,3}$		$true$	
$W_{2,2}$, $P_{1,3}$, $P_{3,1}$, $P_{2,2}$			

Figure 7.1 A truth table constructed for Ex. 7.2. Propositions not listed as true on a given line are assumed false, and only $true$ entries are shown in the table.

in every model), and if the entailment holds then α must be valid, because it must be true in all models, because it must be true in all models in which $True$ holds.

b. $False$ doesn't hold in any model, so α trivially holds in every model that $False$ holds in.

c. $\alpha \Rightarrow \beta$ holds in those models where β holds or where $\neg\alpha$ holds. That is precisely the case if $\alpha \Rightarrow \beta$ is valid.

d. This follows from applying **c** in both directions.

e. This reduces to **c**, because $\alpha \land \neg\beta$ is unsatisfiable just when $\alpha \Rightarrow \beta$ is valid.

7.5 These can be computed by counting the rows in a truth table that come out true. Remember to count the propositions that are not mentioned; if a sentence mentions only A and B, then we multiply the number of models for $\{A, B\}$ by 2^2 to account for C and D.

a. 6
b. 12
c. 4

7.6 A binary logical connective is defined by a truth table with 4 rows. Each of the four rows may be true or false, so there are $2^4 = 16$ possible truth tables, and thus 16 possible connectives. Six of these are trivial ones that ignore one or both inputs; they correspond to $True$, $False$, P, Q, $\neg P$ and $\neg Q$. Four of them we have already studied: $\land, \lor, \Rightarrow, \Leftrightarrow$. The remaining six are potentially useful. One of them is reverse implication ($\Leftarrow$ instead of $\Rightarrow$), and the other five are the negations of $\land, \lor, \Rightarrow, \Leftrightarrow$ and $\Leftarrow$. (The first two of these are sometimes called *nand* and *nor*.)

7.7 We use the truth table code in Lisp in the directory `logic/prop.lisp` to show each sentence is valid. We substitute `P, Q, R` for α, β, γ because of the lack of Greek letters in ASCII. To save space in this manual, we only show the first four truth tables:

```
> (truth-table "P ^ Q <=> Q ^ P")
----------------------------------------
 P  Q  P ^ Q  Q ^ P  (P ^ Q) <=> (Q ^ P)
----------------------------------------
 F  F    F      F            \(true\)
 T  F    F      F             T
 F  T    F      F             T
 T  T    T      T             T
----------------------------------------
NIL

> (truth-table "P | Q <=> Q | P")
----------------------------------------
 P  Q  P | Q  Q | P  (P | Q) <=> (Q | P)
----------------------------------------
 F  F    F      F             T
 T  F    T      T             T
 F  T    T      T             T
 T  T    T      T             T
----------------------------------------
NIL

> (truth-table "P ^ (Q ^ R) <=> (P ^ Q) ^ R")
---------------------------------------------------------------------
 P  Q  R  Q ^ R  P ^ (Q ^ R)  P ^ Q ^ R  (P ^ (Q ^ R)) <=> (P ^ Q ^ R)
---------------------------------------------------------------------
 F  F  F    F         F           F                   T
 T  F  F    F         F           F                   T
 F  T  F    F         F           F                   T
 T  T  F    F         F           F                   T
 F  F  T    F         F           F                   T
 T  F  T    F         F           F                   T
 F  T  T    T         F           F                   T
 T  T  T    T         T           T                   T
----------------------------------------------------------------------
NIL

> (truth-table "P | (Q | R) <=> (P | Q) | R")
----------------------------------------------------------------------
 P  Q  R  Q | R  P | (Q | R)  P | Q | R  (P | (Q | R)) <=> (P | Q | R)
----------------------------------------------------------------------
 F  F  F    F         F           F                   T
 T  F  F    F         T           T                   T
 F  T  F    T         T           T                   T
 T  T  F    T         T           T                   T
 F  F  T    T         T           T                   T
 T  F  T    T         T           T                   T
 F  T  T    T         T           T                   T
 T  T  T    T         T           T                   T
----------------------------------------------------------------------
NIL
```

For the remaining sentences, we just show that they are valid according to the `validity` function:

```
> (validity "~~P <=> P")
VALID
> (validity "P => Q <=> ~Q => ~P")
VALID
> (validity "P => Q <=> ~P | Q")
VALID
> (validity "(P <=> Q) <=> (P => Q) ^ (Q => P)")
VALID
> (validity "~(P ^ Q) <=> ~P | ~Q")
VALID
> (validity "~(P | Q) <=> ~P ^ ~Q")
VALID
> (validity "P ^ (Q | R) <=> (P ^ Q) | (P ^ R)")
VALID
> (validity "P | (Q ^ R) <=> (P | Q) ^ (P | R)")
VALID
```

7.8 We use the `validity` function from `logic/prop.lisp` to determine the validity of each sentence:

```
> (validity "Smoke => Smoke")
VALID
> (validity "Smoke => Fire")
SATISFIABLE
> (validity "(Smoke => Fire) => (~Smoke => ~Fire)")
SATISFIABLE
> (validity "Smoke | Fire | ~Fire")
VALID
```

```
> (validity "((Smoke ^ Heat) => Fire) <=> ((Smoke => Fire) | (Heat => Fire))")
VALID
> (validity "(Smoke => Fire) => ((Smoke ^ Heat) => Fire)")
VALID
> (validity "Big | Dumb | (Big => Dumb)")
VALID
> (validity "(Big ^ Dumb) | ~Dumb")
SATISFIABLE
```

Many people are fooled by (e) and (g) because they think of implication as being causation, or something close to it. Thus, in (e), they feel that it is the combination of Smoke and Heat that leads to Fire, and thus there is no reason why one or the other alone should lead to fire. Similarly, in (g), they feel that there is no necessary causal relation between Big and Dumb, so the sentence should be satisfiable, but not valid. However, this reasoning is incorrect, because implication is *not* causation—implication is just a kind of disjunction (in the sense that $P \Rightarrow Q$ is the same as $\neg P \lor Q$). So $Big \lor Dumb \lor (Big \Rightarrow Dumb)$ is equivalent to $Big \lor Dumb \lor \neg Big \lor Dumb$, which is equivalent to $Big \lor \neg Big \lor Dumb$, which is true whether Big is true or false, and is therefore valid.

7.9 From the first two statements, we see that if it is mythical, then it is immortal; otherwise it is a mammal. So it must be either immortal or a mammal, and thus horned. That means it is also magical. However, we can't deduce anything about whether it is mythical. Using the propositional reasoning code:

```
> (setf kb (make-prop-kb))
#S(PROP-KB SENTENCE (AND))
> (tell kb "Mythical => Immortal")
T
> (tell kb "~Mythical => ~Immortal ^ Mammal")
T
> (tell kb "Immortal | Mammal => Horned")
T
> (tell kb "Horned => Magical")
T
> (ask kb "Mythical")
NIL
> (ask kb "~Mythical")
NIL
> (ask kb "Magical")
T
> (ask kb "Horned")
T
```

7.10 Each possible world can be written as a conjunction of symbols, e.g. $(A \land C \land E)$. Asserting that a possible world is not the case can be written by negating that, e.g. $\neg(A \land C \land E)$, which can be rewritten as $(\neg A \lor \neg C \lor \neg E)$. This is the form of a clause; a conjunction of these clauses is a CNF sentence, and can list all the possible worlds for the sentence.

7.11

a. This is a disjunction with 28 disjuncts, each one saying that two of the neighbors are true and the others are false. The first disjunct is

$$X_{2,2} \wedge X_{1,2} \wedge \neg X_{0,2} \wedge \neg X_{0,1} \wedge \neg X_{2,1} \wedge \neg X_{0,0} \wedge \neg X_{1,0} \wedge \neg X_{2,0}$$

The other 27 disjuncts each select two different $X_{i,j}$ to be true.

b. There will be $\binom{n}{k}$ disjuncts, each saying that k of the n symbols are true and the others false.

c. For each of the cells that have been probed, take the resulting number n revealed by the game and construct a sentence with $\binom{n}{8}$ disjuncts. Conjoin all the sentences together. Then use DPLL to answer the question of whether this sentence entails $X_{i,j}$ for the particular i, j pair you are interested in.

d. To encode the global constraint that there are M mines altogether, we can construct a disjunct with $\binom{M}{N}$ disjuncts, each of size N. Remember, $\binom{M}{N=M!/(M-N)!}$. So for a Minesweeper game with 100 cells and 20 mines, this will be morre than 10^{39}, and thus cannot be represented in any computer. However, we can represent the global constraint within the DPLL algorithm itself. We add the parameter *min* and *max* to the DPLL function; these indicate the minimum and maximum number of unassigned symbols that must be true in the model. For an unconstrained problem the values 0 and N will be used for these parameters. For a mineseeper problem the value M will be used for both *min* and *max*. Within DPLL, we fail (return false) immediately if *min* is less than the number of remaining symbols, or if *max* is less than 0. For each recursive call to DPLL, we update *min* and *max* by subtracting one when we assign a true value to a symbol.

e. No conclusions are invalidated by adding this capability to DPLL and encoding the global constraint using it.

f. Consider this string of alternating 1's and unprobed cells (indicated by a dash):

```
|-|1|-|1|-|1|-|1|-|1|-|1|-|1|-|
```

There are two possible models: either there are mines under every even-numbered dash, or under every odd-numbered dash. Making a probe at either end will determine whether cells at the far end are empty or contain mines.

7.12

a. $P \Rightarrow Q$ is equivalent to $\neg P \vee Q$ by implication elimination (Figure 7.11), and $\neg(P_1 \wedge \cdots \wedge P_m)$ is equivalent to $(\neg P_1 \vee \cdots \vee \neg P_m)$ by de Morgan's rule, so $(\neg P_1 \vee \cdots \vee \neg P_m \vee Q)$ is equivalent to $(P_1 \wedge \cdots \wedge P_m) \Rightarrow Q$.

b. A clause can have positive and negative literals; arrange them in the form $(\neg P_1 \vee \cdots \vee P_m \vee Q_1 \vee \cdots \vee Q_n)$. Then, setting $Q = Q_1 \vee \cdots \vee Q_n$, we have

$$(\neg P_1 \vee \cdots \vee P_m \vee Q_1 \vee \cdots \vee Q_n)$$

is equivalent to

$$(P_1 \wedge \cdots \wedge P_m) \Rightarrow Q_1 \vee \cdots \vee Q_n$$

c. For atoms p_i, q_i, r_i, s_i where $\textsc{Unify}(p_j, q_k) = \theta$:

$$\frac{\begin{array}{c} p_1 \wedge \ldots\ p_j\ \ldots \wedge p_{n_1} \ \Rightarrow\ r_1 \vee \ldots r_{n_2} \\ s_1 \wedge \ldots \wedge s_{n_3} \ \Rightarrow\ q_1 \vee \ldots\ q_k\ \ldots \vee q_{n_4} \end{array}}{\text{SUBST}(\theta, (p_1 \wedge \ldots p_{j-1} \wedge p_{j+1} \wedge p_{n_1} \wedge s_1 \wedge \ldots s_{n_3} \ \Rightarrow\ r_1 \vee \ldots r_{n_2} \vee q_1 \vee \ldots\ q_{k-1} \vee q_{k+1} \vee \ldots \vee q_{n_4}))}$$

7.13

a. $Arrow^t \ \Leftrightarrow\ Arrow^{t-1} \wedge \neg Shoot^t$

b. $FacingRight^t \ \Leftrightarrow\ (FacingRight^{t-1} \wedge \neg TurnRight^t \wedge \neg TurnLeft^t)$
$\vee(FacingUp^{t-1} \wedge TurnRight^t$
$\vee(FacingDown^{t-1} \wedge TurnLeft^t$

c. These formulae are the same as (7.7) and (7.8), except that the P for pit is replaced by W for wumpus, and B for breezy is replaced by S for smelly.

$$K(\neg W_{4,4})^t \ \Leftrightarrow\ K(\neg S_{3,4})^t \vee K(\neg S4, 4)^t$$
$$K(W_{4,4})^t \ \Leftrightarrow\ K(S_{3,4})^t \wedge K(\neg W_{2,4})^t \wedge K(\neg W_{3,3})^t$$
$$\vee(K(S_{4,3})^t \wedge K(\neg W_{4,2})^t \wedge K(\neg W_{3,3})^t$$

7.14 Optimal behavior means achieving an expected utility that is as good as any other agent program. The PL-WUMPUS-AGENT is clearly non-optimal when it chooses a random move (and may be non-optimal in other braqnches of its logic). One example: in some cases when there are many dangers (breezes and smells) but no safe move, the agent chooses at random. A more thorough analysis should show when it is better to do that, and when it is better to go home and exit the wumpus world, giving up on any chance of finding the gold. Even when it is best to gamble on an unsafe location, our agent does not distinguish degrees of safety – it should choose the unsafe square which contains a danger in the fewest number of possible models. These refinements are hard to state using a logical agent, but we will see in subsequent chapters that a probabilistic agent can handle them. does

7.15 PL-WUMPUS-AGENT keeps track of 6 static state variables besides *KB*. The difficulty is that these variables change—we don't just add new information about them (as we do with pits and breezy locations), we modify exisitng information. This does not sit well with logic, which is designed for eternal truths. So there are two alternatives. The first is to superscript each proposition with the time (as we did with the circuit agents), and then we could, for example, do $\textsc{Tell}(KB, A^3_{1,1})$ to say that the agent is at location $1, 1$ at time 3. Then at time 4, we would have to copy over many of the existing propositions, and add new ones. The second possibility is to treat every proposition as a timeless one, but to remove outdated propositions from the *KB*. That is, we could do $\textsc{Retract}(KB, A_{1,1})$ and then progTell$(KB, A_{1,2})$ to indicate that the agent has moved from $1, 1$ to $1, 2$. Chapter 10 describes the semantics and implementation of RETRACT.

NOTE: Avoid assigning this problem if you don't feel comfortable requiring students to think ahead about the possibility of retraction.

7.16 It will take time proportional to the number of pure symbols plus the number of unit clauses. We assume that $KB \ \Rightarrow\ \alpha$ is false, and prove a contradiction. $\neg(KB \ \Rightarrow\ \alpha)$ is equivalent to $KB \wedge \neg\alpha$. From this sentence the algorithm will first eliminate all the pure symbols, then it will work on unit clauses until it chooses α (which is a unit clause); at that point it will immediately recognize that either choice (true or false) for α leads to failure, which means that the original non-negated assertion is true.

7.17 Code not shown.

Solutions for Chapter 8 First-Order Logic

8.1 This question will generate a wide variety of possible solutions. The key distinction between analogical and sentential representations is that the analogical representation automatically generates consequences that can be "read off" whenever suitable premises are encoded. When you get into the details, this distinction turns out to be quite hard to pin down—for example, what does "read off" mean?—but it can be justified by examining the time complexity of various inferences on the "virtual inference machine" provided by the representation system.

a. Depending on the scale and type of the map, symbols in the map language typically include city and town markers, road symbols (various types), lighthouses, historic monuments, river courses, freeway intersections, etc.

b. Explicit and implicit sentences: this distinction is a little tricky, but the basic idea is that when the map-drawer plunks a symbol down in a particular place, he says one explicit thing (e.g. that Coit Tower is here), but the analogical structure of the map representation means that many implicit sentences can now be derived. Explicit sentences: there is a monument called Coit Tower at this location; Lombard Street runs (approximately) east-west; San Francisco Bay exists and has this shape. Implicit sentences: Van Ness is longer than North Willard; Fisherman's Wharf is north of the Mission District; the shortest drivable route from Coit Tower to Twin Peaks is the following . . ..

c. Sentences unrepresentable in the map language: Telegraph Hill is approximately conical and about 430 feet high (assuming the map has no topographical notation); in 1890 there was no bridge connecting San Francisco to Marin County (map does not represent changing information); Interstate 680 runs either east or west of Walnut Creek (no disjunctive information).

d. Sentences that are easier to express in the map language: any sentence that can be written easily in English is not going to be a good candidate for this question. Any *linguistic* abstraction from the physical structure of San Francisco (e.g. San Francisco is on the end of a peninsula at the mouth of a bay) can probably be expressed equally easily in the predicate calculus, since that's what it was designed for. Facts such as the shape of the coastline, or the path taken by a road, are best expressed in the map language. Even then, one can argue that the coastline drawn on the map actually consists of lots of individual sentences, one for each dot of ink, especially if the map is drawn using a digital plotter. In this case, the advantage of the map is really in the ease of inference combined with suitability for human "visual computing" apparatus.

e. Examples of other analogical representations:

- Analog audio tape recording. Advantages: simple circuits can record and reproduce sounds. Disadvantages: subject to errors, noise; hard to process in order to separate sounds or remove noise etc.
- Traditional clock face. Advantages: easier to read quickly, determination of how much time is available requires no additional computation. Disadvantages: hard to read precisely, cannot represent small units of time (ms) easily.
- All kinds of graphs, bar charts, pie charts. Advantages: enormous data compression, easy trend analysis, communicate information in a way which we can interpret easily. Disadvantages: imprecise, cannot represent disjunctive or negated information.

8.2 The knowledge base does not entail $\forall\, x\;\; P(x)$. To show this, we must give a model where $P(a)$ and $P(b)$ but $\forall\, x\;\; P(x)$ is false. Consider any model with three domain elements, where a and b refer to the first two elements and the relation referred to by P holds only for those two elements.

8.3 The sentence $\exists\, x, y\;\;\; x = y$ is valid. A sentence is valid if it is true in every model. An existentially quantified sentence is true in a model if it holds under any extended interpretation in which its variables are assigned to domain elements. According to the standard semantics of FOL as given in the chapter, every model contains at least one domain element, hence, for any model, there is an extended interpretation in which x and y are assigned to the first domain element. In such an interpretation, $x = y$ is true.

8.4 $\forall\, x, y\;\;\; x = y$ stipulates that there is exactly one object. If there are two objects, then there is an extended interpretation in which x and y are assigned to different objects, so the sentence would be false. Some students may also notice that any unsatisfiable sentence also meets the criterion, since there are no worlds in which the sentence is true.

8.5 We will use the simplest counting method, ignoring redundant combinations. For the constant symbols, there are D^c assignments. Each predicate of arity k is mapped onto a k-ary relation, i.e., a subset of the D^k possible k-element tuples; there are 2^{D^k} such mappings. Each function symbol of arity k is mapped onto a k-ary function, which specifies a value for each of the D^k possible k-element tuples. Including the invisible element, there are $D+1$ choices for each value, so there are $(D+1)^{D^k}$ functions. The total number of possible combinations is therefore

$$D^c \cdot \left(\sum_{k=1}^{A} 2^{D^k}\right) \cdot \left(\sum_{k=1}^{A} (D+1)^{D^k}\right) .$$

Two things to note: first, the number is finite; second, the maximum arity A is the most crucial complexity parameter.

8.6 In this exercise, it is best not to worry about details of tense and larger concerns with consistent ontologies and so on. The main point is to make sure students understand con-

nectives and quantifiers and the use of predicates, functions, constants, and equality. Let the basic vocabulary be as follows:
$Takes(x, c, s)$: student x takes course c in semester s;
$Passes(x, c, s)$: student x passes course c in semester s;
$Score(x, c, s)$: the score obtained by student x in course c in semester s;
$x > y$: x is greater than y;
F and G: specific French and Greek courses (one could also interpret these sentences as referring to *any* such course, in which case one could use a predicate $Subject(c, f)$ meaning that the subject of course c is field f;
$Buys(x, y, z)$: x buys y from z (using a binary predicate with unspecified seller is OK but less felicitous);
$Sells(x, y, z)$: x sells y to z;
$Shaves(x, y)$: person x shaves person y
$Born(x, c)$: person x is born in country c;
$Parent(x, y)$: x is a parent of y;
$Citizen(x, c, r)$: x is a citizen of country c for reason r;
$Resident(x, c)$: x is a resident of country c;
$Fools(x, y, t)$: person x fools person y at time t;
$Student(x)$, $Person(x)$, $Man(x)$, $Barber(x)$, $Expensive(x)$, $Agent(x)$, $Insured(x)$, $Smart(x)$, $Politician(x)$: predicates satisfied by members of the corresponding categories.

a. Some students took French in spring 2001.
$\exists\, x\;\; Student(x) \land Takes(x, F, Spring2001)$.

b. Every student who takes French passes it.
$\forall\, x, s\;\; Student(x) \land Takes(x, F, s) \;\Rightarrow\; Passes(x, F, s)$.

c. Only one student took Greek in spring 2001.
$\exists\, x\;\; Student(x) \land Takes(x, G, Spring2001) \land \forall\, y\;\; y \neq x \Rightarrow \neg Takes(y, G, Spring2001)$.

d. The best score in Greek is always higher than the best score in French.
$\forall\, s\;\; \exists\, x\;\; \forall\, y\;\; Score(x, G, s) > Score(y, F, s)$.

e. Every person who buys a policy is smart.
$\forall\, x\;\; Person(x) \land (\exists\, y, z\;\; Policy(y) \land Buys(x, y, z)) \;\Rightarrow\; Smart(x)$.

f. No person buys an expensive policy.
$\forall\, x, y, z\;\; Person(x) \land Policy(y) \land Expensive(y) \;\Rightarrow\; \neg Buys(x, y, z)$.

g. There is an agent who sells policies only to people who are not insured.
$\exists\, x\;\; Agent(x) \land \forall\, y, z\;\; Policy(y) \land Sells(x, y, z) \;\Rightarrow\; (Person(z) \land \neg Insured(z))$.

h. There is a barber who shaves all men in town who do not shave themselves.
$\exists\, x\;\; Barber(x) \land \forall\, y\;\; Man(y) \land \neg Shaves(y, y) \;\Rightarrow\; Shaves(x, y)$.

i. A person born in the UK, each of whose parents is a UK citizen or a UK resident, is a UK citizen by birth.
$\forall\, x\;\; Person(x) \land Born(x, UK) \land (\forall\, y\;\; Parent(y, x) \Rightarrow ((\exists\, r\;\; Citizen(y, UK, r)) \lor Resident(y, UK))) \;\Rightarrow\; Citizen(x, UK, Birth)$.

j. A person born outside the UK, one of whose parents is a UK citizen by birth, is a UK citizen by descent.

$$\begin{aligned}&\forall\, x\;\; Person(x) \land \neg Born(x, UK) \land (\exists\, y\;\; Parent(y, x) \land Citizen(y, UK, Birth))\\ &\qquad \Rightarrow\; Citizen(x, UK, Descent).\end{aligned}$$

k. Politicians can fool some of the people all of the time, and they can fool all of the people some of the time, but they can't fool all of the people all of the time.

$$\begin{aligned}&\forall\, x\;\; Politician(x) \;\Rightarrow\\ &\qquad (\exists\, y\;\; \forall\, t\;\; Person(y) \land Fools(x, y, t)) \land\\ &\qquad (\exists\, t\;\; \forall\, y\;\; Person(y) \;\Rightarrow\; Fools(x, y, t)) \land\\ &\qquad \neg(\forall\, t\;\; \forall\, y\;\; Person(y) \;\Rightarrow\; Fools(x, y, t))\end{aligned}$$

8.7 The key idea is to see that the word "same" is referring to every *pair* of Germans. There are several logically equivalent forms for this sentence. The simplest is the Horn clause:

$$\forall\, x, y, l\;\; German(x) \land German(y) \land Speaks(x, l) \;\Rightarrow\; Speaks(y, l)\;.$$

8.8 $\forall\, x, y\;\;\; Spouse(x, y) \land Male(x) \;\;\Rightarrow\;\; Female(y)$. This axiom is no longer true in certain states and countries.

8.9 This is a very educational exercise but also highly nontrivial. Once students have learned about resolution, ask them to do the proof too. In most cases, they will discover missing axioms. Our basic predicates are $Heard(x, e, t)$ (x heard about event e at time t); $Occurred(e, t)$ (event e occurred at time t); $Alive(x, t)$ (x is alive at time t).

$$\begin{aligned}&\exists\, t\;\; Heard(W, DeathOf(N), t)\\ &\forall\, x, e, t\;\; Heard(x, e, t) \;\Rightarrow\; Alive(x, t)\\ &\forall\, x, e, t_2\;\; Heard(x, e, t_2) \;\Rightarrow\; \exists\, t_1\;\; Occurred(e, t_1) \land t_1 < t_2\\ &\forall\, t_1\;\; Occurred(DeathOf(x), t_1) \;\Rightarrow\; \forall\, t_2\;\; t_1 < t_2 \;\Rightarrow\; \neg Alive(x, t_2)\\ &\forall\, t_1, t_2\;\; \neg(t_2 < t_1) \;\Rightarrow\; ((t_1 < t_2) \lor (t_1 = t_2))\\ &\forall\, t_1, t_2, t_3\;\; (t_1 < t_2) \land ((t_2 < t_3) \lor (t_2 = t_3)) \;\Rightarrow\; (t_1 < t_3)\\ &\forall\, t_1, t_2, t_3\;\; ((t_1 < t_2) \lor (t_1 = t_2)) \land (t_2 < t_3) \;\Rightarrow\; (t_1 < t_3)\end{aligned}$$

8.10 It is not entirely clear which sentences need to be written, but this is one of them:

$$\forall\, s_1\;\; Breezy(s_1) \;\Leftrightarrow\; \exists\, s_2\;\; Adjacent(s_1, s_2) \land Pit(s_2)\;.$$

That is, a square is breezy if and only if there is a pit in a neighboring square. Generally speaking, the size of the axiom set is independent of the size of the wumpus world being described.

8.11 Make sure you write definitions with $\Leftrightarrow$. If you use $\Rightarrow$, you are only imposing constraints, not writing a real definition.

$GrandChild(c,a) \;\Leftrightarrow\; \exists\, b \;\; Child(c,b) \land Child(b,a)$
$GreatGrandParent(a,d) \;\Leftrightarrow\; \exists\, b,c \;\; Child(d,c) \land Child(c,b) \land Child(b,a)$
$Brother(x,y) \;\Leftrightarrow\; Male(x) \land Sibling(x,y)$
$Sister(x,y) \;\Leftrightarrow\; Female(x) \land Sibling(x,y)$
$Daughter(d,p) \;\Leftrightarrow\; Female(d) \land Child(d,p)$
$Son(s,p) \;\Leftrightarrow\; Male(s) \land Child(s,p)$
$AuntOrUncle(a,c) \;\Leftrightarrow\; \exists\, p \;\; Child(c,p) \land Sibling(a,p)$
$Aunt(a,c) \;\Leftrightarrow\; Female(a) \land AuntOrUncle(a,c)$
$Uncle(u,c) \;\Leftrightarrow\; Male(u) \land AuntOrUncle(a,c)$
$BrotherInLaw(b,x) \;\Leftrightarrow\; \exists\, m \;\; Spouse(x,m) \land Brother(b,m)$
$SisterInLaw(s,x) \;\Leftrightarrow\; \exists\, m \;\; Spouse(x,m) \land Sister(s,m)$
$FirstCousin(c,k) \;\Leftrightarrow\; \exists\,(\;\; p) AuntOrUncle(p,c) \land Parent(p,k)$

A second cousin is a a child of one's parent's first cousin, and in general an nth cousin is defined as:

$NthCousin(n,c,k) \;\Leftrightarrow\; \exists\, p,f \;\; Parent(p,c) \land NthCousin(n-1,f,p) \land Child(k,f)$

The facts in the family tree are simple: each arrow represents two instances of $Child$ (e.g., $Child(William, Diana)$ and $Child(William, Charles)$), each name represents a sex proposition (e.g., $Male(William)$ or $Female(Diana)$), each double line indicates a $Spouse$ proposition (e.g. $Spouse(Charles, Diana)$). Making the queries of the logical reasoning system is just a way of debugging the definitions.

8.12 $\forall\, x,y \;\; (x+y) = (y+x)$. This does follow from the Peano axioms (although we should write the first axiom for + as $\forall\, m \;\; NatNum(m) \;\Rightarrow\; +(0,m) = m$). Roughly speaking, the definition of + says that $x+y = S^x(y) = S^{x+y}(0)$, where S^x is shorthand for the S function applied x times. Similarly, $y+x = S^y(x) = S^{y+x}(0)$. Hence, the axioms imply that $x+y$ and $y+x$ are equal to syntactically identical expressions. This argument can be turned into a formal proof by induction.

8.13 Although these axioms are sufficient to prove set membership when x is in fact a member of a given set, they have nothing to say about cases where x is not a member. For example, it is not possible to prove that x is not a member of the empty set. These axioms may therefore be suitable for a logical system, such as Prolog, that uses negation-as-failure.

8.14 Here we translate $List?$ to mean "proper list" in Lisp terminology, i.e., a cons structure with Nil as the "rightmost" atom.

$List?(Nil)$
$\forall\, x,l \;\; List?(l) \;\Leftrightarrow\; List?(Cons(x,l))$
$\forall\, x,y \;\; First(Cons(x,y)) = x$
$\forall\, x,y \;\; Rest(Cons(x,y)) = y$
$\forall\, x \;\; Append(Nil,x) = x$
$\forall\, v,x,y,z \;\; List?(x) \;\Rightarrow\; (Append(x,y) = z \;\Leftrightarrow\; Append(Cons(v,x),y) = Cons(v,z))$
$\forall\, x \;\; \neg Find(x,Nil)$
$\forall\, x \;\; List?(z) \;\Rightarrow\; (Find(x,Cons(y,z)) \;\Leftrightarrow\; (x = y \lor Find(x,z))$

8.15 There are several problems with the proposed definition. It allows one to prove, say, $Adjacent([1,1],[1,2])$ but not $Adjacent([1,2],[1,1])$; so we need an additional symmetry axiom. It does not allow one to prove that $Adjacent([1,1],[1,3])$ is false, so it needs to be written as

$$\forall\, s_1, s_2 \quad \Leftrightarrow \quad \ldots$$

Finally, it does not work as the boundaries of the world, so some extra conditions must be added.

8.16 We need the following sentences:

$$\begin{array}{l}\forall\, s_1\;\; Smelly(s_1) \;\Leftrightarrow\; \exists\, s_2\;\; Adjacent(s_1, s_2) \wedge In(Wumpus, s_2) \\ \exists\, s_1\;\; In(Wumpus, s_1) \wedge \forall\, s_2\;\; (s_1 \neq s_2) \;\Rightarrow\; \neg In(Wumpus, s_2)\;.\end{array}$$

8.17 There are three stages to go through. In the first stage, we define the concepts of one-bit and n-bit addition. Then, we specify one-bit and n-bit adder circuits. Finally, we verify that the n-bit adder circuit does n-bit addition.

- One-bit addition is easy. Let Add_1 be a function of three one-bit arguments (the third is the carry bit). The result of the addition is a list of bits representing a 2-bit binary number, least significant digit first:

$$\begin{array}{l} Add_1(0,0,0) = [0,0] \\ Add_1(0,0,1) = [0,1] \\ Add_1(0,1,0) = [0,1] \\ Add_1(0,1,1) = [1,0] \\ Add_1(1,0,0) = [0,1] \\ Add_1(1,0,1) = [1,0] \\ Add_1(1,1,0) = [1,0] \\ Add_1(1,1,1) = [1,1] \end{array}$$

- n-bit addition builds on one-bit addition. Let $Add_n(x_1, x_2, b)$ be a function that takes two lists of binary digits of length n (least significant digit first) and a carry bit (initially 0), and constructs a list of length $n+1$ that represents their sum. (It will always be exactly $n+1$ bits long, even when the leading bit is 0—the leading bit is the overflow bit.)

$$\begin{array}{l} Add_n([],[],b) = [b] \\ Add_1(b_1, b_2, b) = [b_3, b_4] \;\Rightarrow\; Add_n([b_1|x_1],[b_2|x_2],b) = [b_3|Add_n(x_1, x_2, b_4)] \end{array}$$

- The next step is to define the structure of a one-bit adder circuit, as given in Section **??**. Let $Add_1Circuit(c)$ be true of any circuit that has the appropriate components and connections:

$$\begin{array}{l}
\forall\, c \;\; Add_1Circuit(c) \;\Leftrightarrow \\
\quad \exists\, x_1, x_2, a_1, a_2, o_1 \;\; Type(x_1) = Type(x_2) = XOR \\
\qquad \wedge\, Type(a_1) = Type(a_2) = AND \wedge Type(o_1) = OR \\
\qquad \wedge\, Connected(Out(1, x_1), In(1, x_2)) \wedge Connected(In(1, c), In(1, x_1)) \\
\qquad \wedge\, Connected(Out(1, x_1), In(2, a_2)) \wedge Connected(In(1, c), In(1, a_1)) \\
\qquad \wedge\, Connected(Out(1, a_2), In(1, o_1)) \wedge Connected(In(2, c), In(2, x_1)) \\
\qquad \wedge\, Connected(Out(1, a_1), In(2, o_1)) \wedge Connected(In(2, c), In(2, a_1)) \\
\qquad \wedge\, Connected(Out(1, x_2), Out(1, c)) \wedge Connected(In(3, c), In(2, x_2)) \\
\qquad \wedge\, Connected(Out(1, o_1), Out(2, c)) \wedge Connected(In(3, c), In(1, a_2))
\end{array}$$

Notice that this allows the circuit to have additional gates and connections, but they won't stop it from doing addition.

- Now we define what we mean by an n-bit adder circuit, following the design of Figure 8.6. We will need to be careful, because an n-bit adder is not just an $n-1$-bit adder plus a one-bit adder; we have to connect the overflow bit of the $n-1$-bit adder to the carry-bit input of the one-bit adder. We begin with the base case, where $n = 0$:

$$\begin{array}{l}
\forall\, c \;\; Add_nCircuit(c, 0) \;\Leftrightarrow \\
Signal(Out(1, c)) = 0
\end{array}$$

Now, for the recursive case we specify that the first connect the "overflow" output of the $n-1$-bit circuit as the carry bit for the last bit:

$$\begin{array}{l}
\forall\, c, n \;\; n > 0 \;\Rightarrow\; [Add_nCircuit(c, n) \;\Leftrightarrow \\
\quad \exists\, c_2, d \;\; Add_nCircuit(c_2, n-1) \wedge Add_1Circuit(d) \\
\qquad \wedge\, \forall\, m \;\; (m > 0) \wedge (m < 2n-1) \;\Rightarrow\; In(m, c) = In(m, c_2) \\
\qquad \wedge\, \forall\, m \;\; (m > 0) \wedge (m < n) \;\Rightarrow\; \wedge\, Out(m, c) = Out(m, c_2) \\
\qquad \wedge\, Connected(Out(n, c_2), In(3, d)) \\
\qquad \wedge\, Connected(In(2n-1, c), In(1, d)) \wedge Connected(In(2n, c), In(2, d)) \\
\qquad \wedge\, Connected(Out(1, d), Out(n, c)) \wedge Connected(Out(2, d), Out(n+1, c))
\end{array}$$

- Now, to verify that a one-bit adder *circuit* actually adds correctly, we ask whether, given any setting of the inputs, the outputs equal the sum of the inputs:

$$\begin{array}{l}
\forall\, c \;\; Add_1Circuit(c) \;\Rightarrow \\
\quad \forall\, i_1, i_2, i_3 \;\; Signal(In(1, c)) = i_1 \wedge Signal(In(2, c)) = i_2 \wedge Signal(In(3, c)) = i_3 \\
\qquad \Rightarrow\; Add_1(i_1, i_2, i_3) = [Out(1, c), Out(2, c)]
\end{array}$$

If this sentence is entailed by the KB, then every circuit with the $Add_1Circuit$ design is in fact an adder. The query for the n-bit can be written as

$$\begin{array}{l}
\forall\, c, n \;\; Add_nCircuit(c, n) \;\Rightarrow \\
\quad \forall\, x_1, x_2, y \;\; InterleavedInputBits(x_1, x_2, c) \wedge OutputBits(y, c) \\
\qquad \Rightarrow\; Add_n(x_1, x_2, y)
\end{array}$$

where $InterleavedInputBits$ and $OutputBits$ are defined appropriately to map bit sequences to the actual terminals of the circuit. [*Note*: this logical formulation has not been tested in a theorem prover and we hesitate to vouch for its correctness.]

8.18 Strictly speaking, the primitive gates must be defined using logical equivalences to exclude those combinations not listed as correct. If we are using a logic programming system, we can simply list the cases. For example,

$$AND(0,0,0) \qquad AND(0,1,0) \qquad AND(1,0,0) \qquad AND(1,1,1)\ .$$

For the one-bit adder, we have

$$\begin{aligned}
&\forall\, i_1, i_2, i_3, o_1, o_2 \;\; \mathit{Add_1\,Circuit}(i_1, i_2, i_3, o_1, o_2) \;\Leftrightarrow \\
&\quad \exists\, o_{x1}, o_{a1}, o_{x2} \;\; XOR(i_1, i_2, o_{x1}) \land XOR(o_{x1}, i_3, o_1) \\
&\qquad \land AND(i_1, i_2, o_{a1}) \land AND(i_3, o_{x1}, o_{a2}) \\
&\qquad \land OR(o_{a2}, o_{a1}, o_2)
\end{aligned}$$

The verification query is

$$\forall\, i_1, i_2, i_3, o_1, o_2 \;\; Add_1(i_1, i_2, i_3) = [, o_1, o_2] \;\Rightarrow\; \mathit{Add_1\,Circuit}(i_1, i_2, i_3, o_1, o_2)$$

It is not possible to ask whether particular terminals are connected in a given circuit, since the terminals is not reified (nor is the circuit itself).

8.19 The answers here will vary by country. The two key rules for UK passports are given in the answer to Exercsie 8.6.

Solutions for Chapter 9
Inference in First-Order Logic

9.1 We want to show that any sentence of the form $\forall\, v \;\; \alpha$ entails any universal instantiation of the sentence. The sentence $\forall\, v \;\; \alpha$ is true if α is true in all possible extended interpretations. But replacing v with any ground term g must count as one of the interpretations, so if the original sentence is true, then the instantiated sentence must also be true.

9.2 For any sentence α containing a ground term g and for any variable v not occuring in α, we have

$$\frac{\alpha}{\exists\, v \;\; \textsc{Subst}_1(\{g/v\}, \alpha)}$$

where $\textsc{Subst}_1$ is a function that substitutes for a single occurrence of g with v.

9.3 Both b and c are valid; a is invalid because it introduces the previously-used symbol *Everest*. Note that c does not imply that there are two mountains as high as Everest, because nowhere is it stated that *BenNevis* is different from *Kilimanjaro* (or *Everest*, for that matter).

9.4 This is an easy exercise to check that the student understands unification.

a. $\{x/A, y/B, z/B\}$ (or some permutation of this).

b. No unifier (x cannot bind to both A and B).

c. $\{y/John, x/John\}$.

d. No unifier (because the occurs-check prevents unification of y with $Father(y)$).

9.5 Employs(Mother(John), Father(Richard)) This page isn't wide enough to draw the diagram as in Figure 9.2, so we will draw it with indentation denoting children nodes:

```
[1] Employs(x, y)
  [2] Employs(x, Father(z))
    [3] Employs(x, Father(Richard))
      [4] Employs(Mother(w), Father(Richard))
        [5] Employs(Mother(John), Father(Richard))
    [6] Employs(Mother(w), Father(z))
      [4] ...
      [7] Employs(Mother(John), Father(z))
        [5] ...
```

```
[8] Employs(Mother(w), y)
  [9] Employs(Mother(John), y)
    [10] Employs(Mother(John), Father(z)
      [5] ...
  [6] ...
```

9.6 We will give the average-case time complexity for each query/scheme combination in the following table. (An entry of the form "1; n" means that it is $O(1)$ to find the first solution to the query, but $O(n)$ to find them all.) We make the following assumptions: hash tables give $O(1)$ access; there are n people in the data base; there are $O(n)$ people of any specified age; every person has one mother; there are H people in Houston and T people in Tiny Town; T is much less than n; in Q4, the second conjunct is evaluated first.

	Q1	Q2	Q3	Q4
S1	1	1; H	1; n	T; T
S2	1	n; n	1; n	n; n
S3	n	n; n	1; n	n^2; n^2
S4	1	n; n	1; n	n; n
S5	1	1; H	1; n	T; T

Anything that is $O(1)$ can be considered "efficient," as perhaps can anything $O(T)$. Note that S1 and S5 dominate the other schemes for this set of queries. Also note that indexing on predicates plays no role in this table (except in combination with an argument), because there are only 3 predicates (which is $O(1)$). It would make a difference in terms of the constant factor.

9.7 This would work if there were no recursive rules in the knowledge base. But suppose the knowledge base contains the sentences:

$$Member(x, [x|r])$$
$$Member(x, r) \Rightarrow Member(x, [y|r])$$

Now take the query $Member(3, [1, 2, 3])$, with a backward chaining system. We unify the query with the consequent of the implication to get the substitution $\theta = \{x/3, y/1, r/[2, 3]\}$. We then substitute this in to the left-hand side to get $Member(3, [2, 3])$ and try to back chain on that with the substitution θ. When we then try to apply the implication again, we get a failure because y cannot be both 1 and 2. In other words, the failure to standardize apart causes failure in some cases where recursive rules would result in a solution if we did standardize apart.

9.8 Consider a 3-SAT problem of the form

$$(x_{1,1} \vee x_{2,1} \vee x_{3,1}) \wedge (x_{1,2} \vee x_{2,2} \vee x_{3,2}) \vee \ldots$$

We want to rewrite this as a single define clause of the form

$$A \wedge B \wedge C \wedge \ldots \Rightarrow Z,$$

along with a few ground clauses. We can do that with the definite clause

$$OneOf(x_{1,1}, x_{2,1}, x_{3,1}) \wedge OneOf(x_{1,2}, x_{2,2}, x_{3,2}) \wedge \ldots \Rightarrow Solved$$

along with the ground clauses

$OneOf(True, x, y)$
$OneOf(x, True, y)$
$OneOf(x, y, True)$

9.9 We use a very simple ontology to make the examples easier:

a. $Horse(x) \Rightarrow Mammal(x)$
$Cow(x) \Rightarrow Mammal(x)$
$Pig(x) \Rightarrow Mammal(x)$

b. $\mathit{Offspring}(x, y) \wedge Horse(y) \Rightarrow Horse(x)$

c. $Horse(Bluebeard)$

d. $Parent(Bluebeard, Charlie)$

e. $\mathit{Offspring}(x, y) \Rightarrow Parent(y, x)$
$Parent(x, y) \Rightarrow \mathit{Offspring}(y, x)$
(Note we couldn't do $\mathit{Offspring}(x, y) \Leftrightarrow Parent(y, x)$ because that is not in the form expected by Generalized Modus Ponens.)

f. $Mammal(x) \Rightarrow Parent(G(x), x)$ (here G is a Skolem function).

9.10 This questions deals with the subject of looping in backward-chaining proofs. A loop is bound to occur whenever a subgoal arises that is a substitution instance of one of the goals on the stack. Not all loops can be caught this way, of course, otherwise we would have a way to solve the halting problem.

a. The proof tree is shown in Figure S9.1. The branch with $\mathit{Offspring}(Bluebeard, y)$ and $Parent(y, Bluebeard)$ repeats indefinitely, so the rest of the proof is never reached.

b. We get an infinite loop because of rule **b**, $\mathit{Offspring}(x, y) \wedge Horse(y) \Rightarrow Horse(x)$. The specific loop appearing in the figure arises because of the ordering of the clauses—it would be better to order $Horse(Bluebeard)$ before the rule from **b**. However, a loop will occur no matter which way the rules are ordered if the theorem-prover is asked for all solutions.

c. One should be able to prove that both Bluebeard and Charlie are horses.

d. Smith *et al.* (1986) recommend the following method. Whenever a "looping" goal occurs (one that is a substitution instance of a supergoal higher up the stack), suspend the attempt to prove that subgoal. Continue with all other branches of the proof for the supergoal, gathering up the solutions. Then use those solutions (suitably instantiated if necessary) as solutions for the suspended subgoal, continuing that branch of the proof to find additional solutions if any. In the proof shown in the figure, the $\mathit{Offspring}(Bluebeard, y)$ is a repeated goal and would be suspended. Since no other way to prove it exists, that branch will terminate with failure. In this case, Smith's method is sufficient to allow the theorem-prover to find both solutions.

9.11 Surprisingly, the hard part to represent is "who is that man." We want to ask "what relationship does that man have to some known person," but if we represent relations with

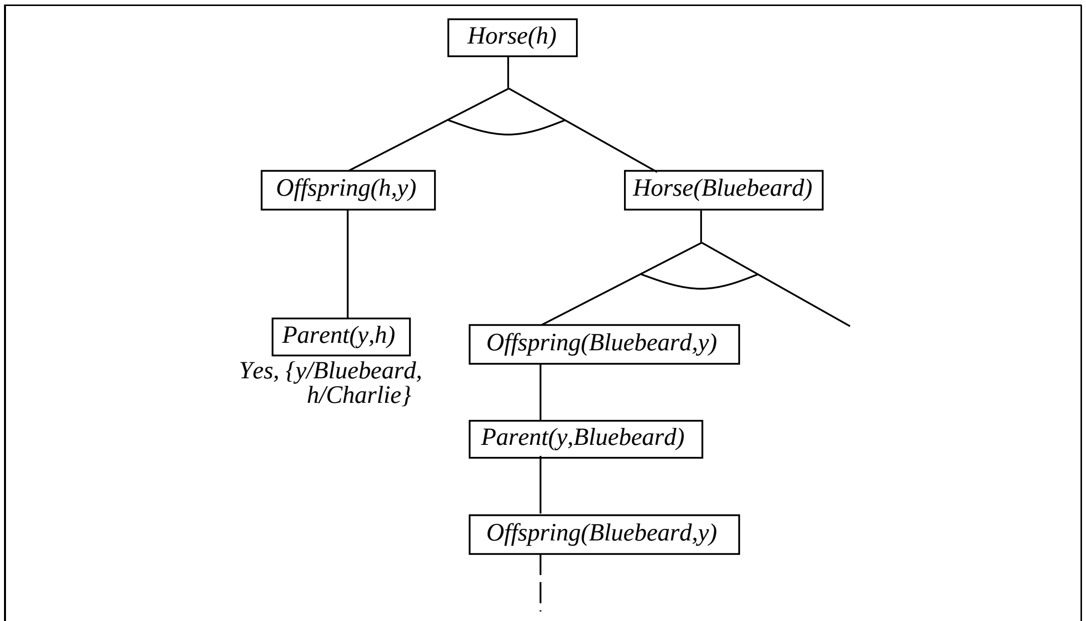

Figure S9.1 Partial proof tree for finding horses.

predicates (e.g., $Parent(x, y)$) then we cannot make the relationship be a variable in first-order logic. So instead we need to reify relationships. We will use $Rel(r, x, y)$ to say that the family relationship r holds between people x and y. Let Me denote me and MrX denote "that man." We will also need the Skolem constants FM for the father of Me and FX for the father of MrX. The facts of the case (put into implicative normal form) are:

$(1)\ Rel(Sibling, Me, x)\ \Rightarrow\ False$
$(2)\ Male(MrX)$
$(3)\ Rel(Father, FX, MrX)$
$(4)\ Rel(Father, FM, Me)$
$(5)\ Rel(Son, FX, FM)$

We want to be able to show that Me is the only son of my father, and therefore that Me is father of MrX, who is male, and therefore that "that man" is my son. The relevant definitions from the family domain are:

$(6)\ Rel(Parent, x, y) \land Male(x)\ \Leftrightarrow\ Rel(Father, x, y)$
$(7)\ Rel(Son, x, y)\ \Leftrightarrow\ Rel(Parent, y, x) \land Male(x)$
$(8)\ Rel(Sibling, x, y)\ \Leftrightarrow\ x \neq y \land \exists p\ \ Rel(Parent, p, x) \land Rel(Parent, p, y)$
$(9)\ Rel(Father, x_1, y) \land Rel(Father, x_2, y)\ \Rightarrow\ x_1 = x_2$

and the query we want is:

$(Q)\ Rel(r, MrX, y)$

We want to be able to get back the answer $\{r/Son, y/Me\}$. Translating 1-9 and Q into INF (and negating Q and including the definition of $\neq$) we get:

$(6a)\ Rel(Parent,x,y) \wedge Male(x) \ \Rightarrow\ Rel(Father,x,y)$
$(6b)\ Rel(Father,x,y) \ \Rightarrow\ Male(x)$
$(6c)\ Rel(Father,x,y) \ \Rightarrow\ Rel(Parent,x,y)$
$(7a)\ Rel(Son,x,y) \ \Rightarrow\ Rel(Parent,y,x)$
$(7b)\ Rel(Son,x,y) \ \Rightarrow\ Male(x))$
$(7c)\ Rel(Parent,y,x) \wedge Male(x) \ \Rightarrow\ Rel(Son,x,y)$
$(8a)\ Rel(Sibling,x,y) \ \Rightarrow\ x \neq y$
$(8b)\ Rel(Sibling,x,y) \ \Rightarrow\ Rel(Parent,P(x,y),x)$
$(8c)\ Rel(Sibling,x,y) \ \Rightarrow\ Rel(Parent,P(x,y),y)$
$(8d)\ Rel(Parent,P(x,y),x) \wedge Rel(Parent,P(x,y),y) \wedge x \neq y \ \Rightarrow\ Rel(Sibling,x,y)$
$(9)\ Rel(Father,x_1,y) \wedge Rel(Father,x_2,y) \ \Rightarrow\ x_1 = x_2$
$(N)\ True \ \Rightarrow\ x = y \vee x \neq y$
$(N')\ x = y \wedge x \neq y \ \Rightarrow\ False$
$(Q')\ Rel(r,MrX,y) \ \Rightarrow\ False$

Note that (1) is non-Horn, so we will need resolution to be be sure of getting a solution. It turns out we also need demodulation (page 284) to deal with equality. The following lists the steps of the proof, with the resolvents of each step in parentheses:

$(10)\ Rel(Parent,FM,Me)$	$(4,6c)$
$(11)\ Rel(Parent,FM,FX)$	$(5,7a)$
$(12)\ Rel(Parent,FM,y) \wedge Me \neq y \ \Rightarrow\ Rel(Sibling,Me,y)$	$(10,8d)$
$(13)\ Rel(Parent,FM,y) \wedge Me \neq y \ \Rightarrow\ False$	$(12,1)$
$(14)\ Me \neq FX \ \Rightarrow\ False$	$(13,11)$
$(15)\ Me = FX$	$(14,N)$
$(16)\ Rel(Father,Me,MrX)$	$(15,3,demodulation)$
$(17)\ Rel(Parent,Me,MrX)$	$(16,6c)$
$(18)\ Rel(Son,MrX,Me)$	$(17,2,7c)$
$(19)\ False\ \{r/Son,y/Me\}$	$(18,Q')$

9.12 Here is a goal tree:

```
goals = [Criminal(West)]
  goals = [American(West), Weapon(y), Sells(West, y, z), Hostile(z)]
  goals = [Weapon(y), Sells(West, y, z), Hostile(z)]
    goals = [Missle(y), Sells(West, y, z), Hostile(z)]
    goals = [Sells(West, M1, z), Hostile(z)]
      goals = [Missle(M1), Owns(Nono, M1), Hostile(Nono)]
      goals = [Owns(Nono, M1), Hostile(Nono)]
    goals = [Hostile(Nono)]
goals = []
```

9.13

a. In the following, an indented line is a step deeper in the proof tree, while two lines at the same indentation represent two alternative ways to prove the goal that is unindented above it. The `P1` and `P2` annotation on a line mean that the first or second clause of `P` was used to derive the line.

```
P(A, [1,2,3])                   goal
  P(1, [1|2,3])                 P1 => solution, with A = 1
  P(1, [1|2,3])                 P2
    P(2, [2,3])                 P1 => solution, with A = 2
    P(2, [2,3])                 P2
      P(3, [3])                 P1 => solution, with A = 3
      P(3, [3])                 P2

P(2, [1, A, 3])                 goal
  P(2, [1|2, 3])                P1
  P(2, [1|2, 3])                P2
    P(2, [2|3])                 P1 => solution, with A = 2
    P(2, [2|3])                 P2
      P(2, [3])                 P1
      P(2, [3])                 P2
```

b. `P` could better be called `Member`; it succeeds when the first argument is an element of the list that is the second argument.

9.14 The different versions of `sort` illustrate the distinction between logical and procedural semantics in Prolog.

a.
```
sorted([]).
sorted([X]).
sorted([X,Y|L]) :- X<Y, sorted([Y|L]).
```
b.
```
perm([],[]).
perm([X|L],M) :-
  delete(X,M,M1),
  perm(L,M1).

delete(X,[X|L],L).       %% deleting an X from [X|L] yields L
delete(X,[Y|L],[Y|M]) :- delete(X,L,M).

member(X,[X|L]).
member(X,[_|L]) :- member(X,L).
```
c.
```
sort(L,M) :- perm(L,M), sorted(M).
```
This is about as close to an executable formal specification of sorting as you can get—it says the absolute minimum about what sort means: in order for `M` to be a sort of `L`, it must have the same elements as `L`, and they must be in order.

d. Unfortunately, this doesn't fare as well as a program as it does as a specification. It is a generate-and-test sort: `perm` generates candidate permutations one at a time, and `sorted` tests them. In the worst case (when there is only one sorted permutation, and it is the last one generated), this will take $O(n!)$ generations. Since each `perm` is $O(n^2)$ and each `sorted` is $O(n)$, the whole `sort` is $O(n!n^2)$ in the worst case.

e. Here's a simple insertion sort, which is $O(n^2)$:

```
isort([],[]).
isort([X|L],M) :- isort(L,M1), insert(X,M1,M).
```

```
insert(X,[],[X]).
insert(X,[Y|L],[X,Y|L]) :- X=<Y.
insert(X,[Y|L],[Y|M]) :- Y<X, insert(X,L,M).
```

9.15 This exercise illustrates the power of pattern-matching, which is built into Prolog.

a. The code for simplification looks straightforward, but students may have trouble finding the middle way between undersimplifying and looping infinitely.

```
simplify(X,X) :- primitive(X).
simplify(X,Y) :- evaluable(X), Y is X.
simplify(Op(X)) :- simplify(X,X1), simplify_exp(Op(X1)).
simplify(Op(X,Y)) :- simplify(X,X1), simplify(Y,Y1), simplify_exp(Op(X1,Y1)).

simplify_exp(X,Y) :- rewrite(X,X1), simplify(X1,Y).
simplify_exp(X,X).

primitive(X) :- atom(X).
```

b. Here are a few representative rewrite rules drawn from the extensive list in Norvig (1992).

```
Rewrite(X+0,X).
Rewrite(0+X,X).
Rewrite(X+X,2*X).
Rewrite(X*X,X^2).
Rewrite(X^0,1).
Rewrite(0^X,0).
Rewrite(X*N,N*X) :- number(N).
Rewrite(ln(e^X),X).
Rewrite(X^Y*X^Z,X^(Y+Z)).
Rewrite(sin(X)^2+cos(X)^2,1).
```

c. Here are the rules for differentiation, using `d(Y,X)` to represent the derivative of expression `Y` with respect to variable `X`.

```
Rewrite(d(X,X),1).
Rewrite(d(U,X),0) :- atom(U), U /= X.
Rewrite(d(U+V,X),d(U,X)+d(V,X)).
Rewrite(d(U-V,X),d(U,X)-d(V,X)).
Rewrite(d(U*V,X),V*d(U,X)+U*d(V,X)).
Rewrite(d(U/V,X),(V*d(U,X)-U*d(V,X))/(V^2)).
Rewrite(d(U^N,X),N*U^(N-1)*d(U,X)) :- number(N).
Rewrite(d(log(U),X),d(U,X)/U).
Rewrite(d(sin(U),X),cos(U)*d(U,X)).
Rewrite(d(cos(U),X),-sin(U)*d(U,X)).
Rewrite(d(e^U,X),d(U,X)*e^U).
```

9.16 Once you understand how Prolog works, the answer is easy:

```
solve(X,[X]) :- goal(X).
solve(X,[X|P]) :- successor(X,Y), solve(Y,P).
```

We could render this in English as "Given a start state, if it is a goal state, then the path consisting of just the start state is a solution. Otherwise, find some successor state such that there is a path from the successor to the goal; then a solution is the start state followed by that path."

Notice that `solve` can not only be used to find a path `P` that is a solution, it can also be used to verify that a given path is a solution.

If you want to add heuristics (or even breadth-first search), you need an explicit queue. The algorithms become quite similar to the versions written in Lisp or Python or Java or in pseudo-code in the book.

9.17 This question tests both the student's understanding of resolution and their ability to think at a high level about relations among sets of sentences. Recall that resolution allows one to show that $KB \models \alpha$ by proving that $KB \land \neg\alpha$ is inconsistent. Suppose that in general the resolution system is called using $\textsc{Ask}(KB, \alpha)$. Now we want to show that a given sentence, say β is valid or unsatisfiable.

A sentence β is valid if it can be shown to be true without additional information. We check this by calling $\textsc{Ask}(KB_0, \beta)$ where KB_0 is the empty knowledge base.

A sentence β that is unsatisfiable is inconsistent by itself. So if we the empty knowledge base again and call $\textsc{Ask}(KB_0, \neg\beta)$ the resolution system will attempt to derive a contradiction starting from $\neg\neg\beta$. If it can do so, then it must be that $\neg\neg\beta$, and hence β, is inconsistent.

9.18 This is a form of inference used to show that Aristotle's syllogisms could not capture all sound inferences.

a. $\forall\, x \;\; Horse(x) \;\Rightarrow\; Animal(x)$
$\forall\, x, h \;\; Horse(x) \land HeadOf(h, x) \;\Rightarrow\; \exists\, y \;\; Animal(y) \land HeadOf(h, y)$

b. $A.\ \neg Horse(x) \lor Animal(x)$
$B.\ Horse(G)$
$C.\ HeadOf(H, G)$
$D.\ \neg Animal(y) \lor \neg HeadOf(H, y)$
(Here A. comes from the first sentence in **a.** while the others come from the second. H and G are Skolem constants.)

c. Resolve D and C to yield $\neg Animal(G)$. Resolve this with A to give $\neg Horse(G)$. Resolve this with B to obtain a contradiction.

9.19 This exercise tests the students understanding of models and implication.

a. (A) translates to "For every natural number there is some other natural number that is smaller than or equal to it." (B) translates to "There is a particular natural number that is smaller than or equal to any natural number."

b. Yes, (A) is true under this interpretation. You can always pick the number itself for the "some other" number.

c. Yes, (B) is true under this interpretation. You can pick 0 for the "particular natural number."

d. No, (A) does not logically entail (B).

e. Yes, (B) logically entails (A).

f. We want to try to prove via resolution that (A) entails (B). To do this, we set our knowledge base to consist of (A) and the negation of (B), which we will call (-B), and try to derive a contradiction. First we have to convert (A) and (-B) to canonical form. For (-B), this involves moving the $\neg$ in past the two quantifiers. For both sentences, it involves introducing a Skolem function:

(A) $x \geq F_1(x)$
(-B) $\neg F_2(y) \geq y$

Now we can try to resolve these two together, but the occurs check rules out the unification. It looks like the substitution should be $\{x/F_2(y),\ y/F_1(x)\}$, but that is equivalent to $\{x/F_2(y),\ y/F_1(F_2(y))\}$, which fails because y is bound to an expression containing y. So the resolution fails, there are no other resolution steps to try, and therefore (B) does not follow from (A).

g. To prove that (B) entails (A), we start with a knowledge base containing (B) and the negation of (A), which we will call (-A):

(-A) $\neg F_1 \geq y$
(B) $x \geq F_2(x)$

This time the resolution goes through, with the substitution $\{x/F_1,\ y/F_2(F_1)\}$, thereby yielding $False$, and proving that (B) entails (A).

9.20 One way of seeing this is that resolution allows reasoning by cases, by which we can prove C by proving that either A or B is true, without knowing which one. With definite clauses, we always have a single chain of inference, for which we can follow the chain and instantiate variables.

9.21 No. Part of the definition of algorithm is that it must terminate. Since there can be an infinite number of consequences of a set of sentences, no algorithm can generate them all. Another way to see thatthe answer is no is to remember that entailment for FOL is semidecidable. If there were an algorithm that generates the set of consequences of a set of sentences S, then when given the task of deciding if B is entailed by S, one could just check if B is in the generated set. But we know that this is not possible, therefore generating the set of sentences is impossible.

Solutions for Chapter 10 Knowledge Representation

10.1 Shooting the wumpus makes it dead, but there are no actions that cause it to come alive. Hence the successor-state axiom for $Alive$ just has the second clause:

$$\begin{aligned}\forall\, a, s \;\; Alive(Wumpus, Result(a, s)) \;\;\Leftrightarrow\;\; & [Alive(x, y, s) \\ & \wedge\neg(a = Shoot \wedge Has(Agent, Arrow, s) \\ & \wedge Facing(Agent, Wumpus, s))]\end{aligned}$$

where $Facing(a, b, s)$ is defined appropriately in terms of the locations of a and b and the orientation of a. Possession of the arrow is lost by shooting, and again there is no way to make it true:

$$\begin{aligned}&\forall\, a, s \;\; Has(Agent, Arrow, Result(a, s)) \;\Leftrightarrow \\ &\quad [Has(Agent, Arrow, s) \wedge (a \neq Shoot)]\end{aligned}$$

10.2

$$Time(S_0, 0)$$
$$Time(Result(seq, s), t) \;\Rightarrow\; Time(Result([a|seq], s), t + 1)$$

Notice that the recursion needs no base case because we already have the axiom

$$Result([\,], s) = s\ .$$

10.3 This question takes the student through the initial stages of developing a logical representation for actions that incorporates more and more realism. Implementing the reasoning tasks in a theorem-prover is also a good idea. Although the use of logical reasoning for the initial task—finding a route on a graph—may seem like overkill, the student should be impressed that we can keep making the situation more complicated simply by describing those added complications, with no additions to the reasoning system.

a. $At(Robot, Arad, S_0)$.

b. $\exists\, s \;\; At(Robot, Bucharest, s)$.

c. The successor-state axiom should be mechanical by now. $\forall\, a, x, y, s$:

$$\begin{aligned}At(Robot, y, Result(a, s)) \;\;\Leftrightarrow\;\; & [(a = Go(x, y) \\ & \wedge DirectRoute(x, y) \wedge At(Robot, x, s)) \\ \vee\;\; & (At(Robot, y, s) \\ & \wedge\neg(\exists\, z \;\; a = Go(y, z) \wedge z \neq y))]\end{aligned}$$

d. To represent the amount of fuel the robot has in a given situation, use the function $Fuel(Robot, s)$. Let $Distance(x, y)$ denote the distance between cities x and y, measured in units of fuel consumption. Let $Full$ be a constant denoting the fuel capacity of the tank.

e. The initial situation is described by $At(Robot, Arad, S_0) \land Fuel(Robot, s) = Full$. The above axiom for location is extended as follows (note that we do not say what happens if the robot runs out of gas). $\forall\, a, x, y, s$:

$$\begin{aligned}
At(Robot, y, Result(a, s)) \quad &\Leftrightarrow \quad [(a = Go(x, y) \\
&\qquad \land DirectRoute(x, y) \land At(Robot, x, s) \\
&\qquad \land Distance(x, y) < Fuel(Robot, s)) \\
&\quad \lor \quad (At(Robot, y, s) \\
&\qquad \land \neg(\exists\, z \;\; a = Go(y, z) \land z \neq y))] \\
Fuel(Robot, Result(a, s)) = f \quad &\Leftrightarrow \quad [(a = Go(x, y) \\
&\qquad \land DirectRoute(x, y) \land At(Robot, x, s) \\
&\qquad \land Distance(x, y) < Fuel(Robot, s) \\
&\qquad \land f = Fuel(Robot, s) - Distance(x, y)) \\
&\quad \lor \quad (f = Fuel(Robot, s) \\
&\qquad \land \neg(\exists\, v, w \;\; a = Go(v, w) \land v \neq w))]
\end{aligned}$$

f. The simplest way to extend the representation is to add the predicate $GasStation(x)$, which is true of cities with gas stations. The $Fillup$ action is described by adding another clause to the above axiom for $Fuel$, saying that $f = Full$ when $a = FillUp$.

10.4 This question was inadvertently left in the exercises after the corresponding material was excised from the chapter. Future printings may omit or replace this exercise.

10.5 Remember that we defined substances so that $Water$ is a category whose elements are all those things of which one might say "it's water." One tricky part is that the English language is ambiguous. One sense of the word "water" includes ice ("that's frozen water"), while another sense excludes it: ("that's not water—it's ice"). The sentences here seem to use the first sense, so we will stick with that. It is the sense that is roughly synonymous with H_2O.

The other tricky part is that we are dealing with objects that change (freeze and melt) over time. Thus, it won't do to say $w \in Liquid$, because w (a mass of water) might be a liquid at one time and a solid at another. For simplicity, we will use a situation calculus representation, with sentences such as $T(w \in Liquid, s)$. There are many possible correct answers to each of these. The key thing is to be *consistent* in the way that information is represented. For example, do not use $Liquid$ as a predicate on objects if $Water$ is used as a substance category.

a. "Water is a liquid between 0 and 100 degrees." We will translate this as "For any water and any situation, the water is liquid iff and only if the water's temperature in the situation is between 0 and 100 centigrade."

$$\begin{array}{l}\forall\, w,s\;\; w \in Water \;\Rightarrow\\ \quad (Centigrade(0) < Temperature(w,s) < Centigrade(100)) \;\Leftrightarrow\\ \qquad T(w \in Liquid, s)\end{array}$$

b. "Water boils at 100 degrees." It is a good idea here to do some tool-building. On page 243 we used $MeltingPoint$ as a predicate applying to individual instances of a substance. Here, we will define $SBoilingPoint$ to denote the boiling point of all instances of a substance. The basic meaning of boiling is that instances of the substance becomes gaseous above the boiling point:

$$\begin{array}{l}SBoilingPoint(c,bp) \;\Leftrightarrow\\ \quad \forall\, x,s\;\; x \in c \;\Rightarrow\\ \qquad (\forall\, t\;\; T(Temperature(x,t),s) \wedge t > bp \;\Rightarrow\; T(x \in Gas, s))\end{array}$$

Then we need only say $SBoilingPoint(Water, Centigrade(100))$.

c. "The water in John's water bottle is frozen."

We will use the constant Now to represent the situation in which this sentence holds. Note that it is easy to make mistakes in which one asserts that only some of the water in the bottle is frozen.

$$\begin{array}{l}\exists\, b\;\; \forall\, w\;\; w \in Water \wedge b \in WaterBottles \wedge Has(John, b, Now)\\ \qquad \wedge\; Inside(w,b,Now) \;\Rightarrow\; (w \in Solid, Now)\end{array}$$

d. "Perrier is a kind of water."

$$Perrier \subset Water$$

e. "John has Perrier in his water bottle."

$$\begin{array}{l}\exists\, b\;\; \forall\, w\;\; w \in Water \wedge b \in WaterBottles \wedge Has(John, b, Now)\\ \qquad \wedge\; Inside(w,b,Now) \;\Rightarrow\; w \in Perrier\end{array}$$

f. "All liquids have a freezing point."

Presumably what this means is that all substances that are liquid at room temperature have a freezing point. If we use $RTLiquidSubstance$ to denote this class of substances, then we have

$$\forall\, c\;\; RTLiquidSubstance(c) \;\Rightarrow\; \exists\, t\;\; SFreezingPoint(c,t)$$

where $SFreezingPoint$ is defined similarly to $SBoilingPoint$. Note that this statement is false in the real world: we can invent categories such as "blue liquid" which do not have a unique freezing point. An interesting exercise would be to define a "pure" substance as one all of whose instances have the same chemical composition.

g. "A liter of water weighs more than a liter of alcohol."

$$\begin{array}{l}\forall\, w,a\;\; w \in Water \wedge a \in Alcohol \wedge Volume(w) = Liters(1)\\ \quad \wedge Volume(a) = Liters(1) \;\Rightarrow\; Mass(w) > Mass(a)\end{array}$$

10.6 This is a fairly straightforward exercise that can be done in direct analogy to the corresponding definitions for sets.

a. $ExhaustivePartDecomposition$ holds between a set of parts and a whole, saying that anything that is a part of the whole must be a part of one of the set of parts.

$$\begin{array}{l}\forall\, s,w \;\; ExhaustivePartDecomposition(s,w) \;\Leftrightarrow \\ \qquad (\forall\, p \;\; PartOf(p,w) \;\Rightarrow\; \exists\, p_2 \;\; p_2 \in s \land PartOf(p,p_2))\end{array}$$

b. $PartPartition$ holds between a set of parts and a whole when the set is disjoint and is an exhaustive decomposition.

$$\begin{array}{l}\forall\, s,w \;\; PartPartition(s,w) \;\Leftrightarrow \\ \qquad PartwiseDisjoint(s) \land ExhaustivePartDecomposition(s,w)\end{array}$$

c. A set of parts is $PartwiseDisjoint$ if when you take any two parts from the set, there is nothing that is a part of both parts.

$$\begin{array}{l}\forall\, s \;\; PartwiseDisjoint(s) \;\Leftrightarrow \\ \;\forall\, p_1,p_2 \;\; p_1 \in s \land p_2 \in s \land p_1 \neq p_2 \;\Rightarrow\; \neg\exists p_3 \; PartOf(p_3,p_1) \land PartOf(p_3,p_2)\end{array}$$

It is *not* the case that $PartPartition(s, BunchOf(s))$ for any s. A set s may consist of physically overlapping objects, such as a hand and the fingers of the hand. In that case, $BunchOf(s)$ is equal to the hand, but s is not a partition of it. We need to ensure that the elements of s are partwise disjoint:

$$\forall\, s \;\; PartwiseDisjoint(s) \;\Rightarrow\; PartPartition(s, BunchOf(s))\;.$$

10.7 For an instance i of a substance s with price per pound c and weight n pounds, the price of i will be $n \times c$, or in other words:

$$\begin{array}{l}\forall\, i,s,n,c \;\; i \in s \land PricePer(s, Pounds(1)) = \$(c) \land Weight(i) = Pounds(n) \\ \qquad \Rightarrow\; Price(i) = \$(n \times c)\end{array}$$

If b is the set of tomatoes in a bag, then $BunchOf(b)$ is the composite object consisting of all the tomatoes in the bag. Then we have

$$\begin{array}{l}\forall\, i,s,n,c \;\; b \subset s \land PricePer(s, Pounds(1)) = \$(c) \\ \qquad\qquad \land\, Weight(BunchOf(b)) = Pounds(n) \\ \qquad \Rightarrow\; Price(BunchOf(b)) = \$(n \times c)\end{array}$$

10.8 In the scheme in the chapter, a conversion axiom looks like this:

$$\forall\, x \;\; Centimeters(2.54 \times x) = Inches(x)\;.$$

"50 dollars" is just $\$(50)$, the name of an abstract monetary quantity. For any measure function such as \$, we can extend the use of $>$ as follows:

$$\forall\, x,y \;\; x > y \;\Rightarrow\; \$(x) > \$(y)\;.$$

Since the conversion axiom for dollars and cents has

$$\forall\, x \;\; Cents(100 \times x) = \$(x)$$

it follows immediately that $\$(50) > Cents(50)$.

In the new scheme, we must introduce objects whose lengths are converted:

$$\forall\, x \;\; Centimeters(Length(x)) = 2.54 \times Inches(Length(x))\;.$$

There is no obvious way to refer directly to “50 dollars” or its relation to “50 cents”. Again, we must introduce objects whose monetary value is 50 dollars or 50 cents:

$$\forall\, x, y \;\; \$(Value(x)) = 50 \land Cents(Value(y)) = 50 \;\Rightarrow\; \$(Value(x)) > \$(Value(y))$$

10.9 We will define a function $ExchangeRate$ that takes three arguments: a source currency, a time interval, and a target currency. It returns a number representing the exchange rate. For example,

$$ExchangeRate(USDollar, 17Feb1995, DanishKrone) = 5.8677$$

means that you can get 5.8677 Krone for a dollar on February 17th. This was the Federal Reserve bank’s Spot exchange rate as of 10:00 AM. It is the mid-point between the buying and selling rates. A more complete analysis might want to include buying and selling rates, and the possibility for many different exchanges, as well as the commissions and fees involved. Note also the distinction between a currency such as $USDollar$ and a unit of measurement, such as is used in the expression $USDollars(1.99)$.

10.10 Another fun problem for clear thinkers:

a. Remember that $T(c, i)$ means that some event of type c occurs throughout the interval i:

$$\forall\, c, i \;\; T(c, i) \;\Leftrightarrow\; (\exists\, e \;\; e \in c \land During(i, e))$$

Using $SubEvent$ as the question requests is not so easy, because the interval subsumes all events within it.

b. A $Both(p, q)$ event is one in which both p and q occur throughout the duration of the event. There is only one way this can happen: both p and q have to persist for the whole interval. Another way to say it:

$$\forall\, i, j \;\; During(j, i) \;\Rightarrow\; [T(p, j) \land T(q, j)]$$

is logically equivalent to

$$[\forall\, i, j \;\; During(j, i) \;\Rightarrow\; T(p, j)] \land [\forall\, i, j \;\; During(j, i) \;\Rightarrow\; T(q, j)]$$

whereas the same equivalence fails to hold for disjunction (see the next part).

c. $T(OneOf(p, q), i)$ means that a p event occurs throughout i or a q event does:

$$\begin{aligned}&\forall\, p, q, i \;\; T(OneOf(p, q), i) \;\Leftrightarrow\\ &\qquad [(\forall\, j \;\; During(j, i) \;\Rightarrow\; T(p, j)) \lor (\forall\, j \;\; During(j, i) \;\Rightarrow\; T(q, j))]\;.\end{aligned}$$

On the other hand, $T(Either(p, q), i)$ holds if, at every point in i, either a p or a q is happening:

$$\forall\, p, q, i \;\; T(OneOf(p, q), i) \;\Leftrightarrow\; (\forall\, j \;\; During(j, i) \;\Rightarrow\; (T(p, j) \lor T(q, j)))\;.$$

d. $T(Never(p), i)$ should mean that there is never an event of type p going on in any subinterval of i, while $T(Not(p), i)$ should mean that there is no single event of type p that spans all of i, even though there may be one or more events of type p for subintervals of i:

$$\forall\, p, i \;\; T(Never(p), i) \;\Leftrightarrow\; \neg\exists\, j \;\; During(j, i) \land T(p, j)$$
$$\forall\, p, i \;\; T(Not(p), i) \;\Leftrightarrow\; \neg T(p, i)$$

One could also ask students to prove two versions of de Morgan's laws using the two types of negation, each with its corresponding type of disjunction.

10.11 Any object x is an event, and $Location(x)$ is the event that for every subinterval of time, refers to the place where x is. For example, $Location(Peter)$ is the complex event consisting of his home from midnight to about 9:00 today, then various parts of the road, then his office from 10:00 to 1:30, and so on. To say that an event is fixed is to say that any two moments of the event have the same spatial extent:

$$\begin{aligned}&\forall\, e\;\; Fixed(e) \;\Leftrightarrow\\ &\quad(\forall\, a,b\;\; a \in Moments \wedge b \in Moments \wedge Subevent(a,e) \wedge Subevent(b,e)\\ &\qquad\Rightarrow\; SpatialExtent(a) = SpatialExtent(b))\end{aligned}$$

10.12 We will omit universally quantified variables:

$$\begin{array}{lcl} Before(i,j) & \Leftrightarrow & \exists\, k\;\; Meet(i,k) \wedge Meet(k,j)\\ After(i,j) & \Leftrightarrow & Before(j,i)\\ During(i,j) & \Leftrightarrow & \exists\, k,m\;\; Meet(Start(j),k) \wedge Meet(k,i)\\ & & \qquad \wedge Meet(i,m) \wedge Meet(m,End(j))\\ Overlap(i,j) & \Leftrightarrow & \exists\, k\;\; During(k,i) \wedge During(k,j)\end{array}$$

10.13

$$\begin{aligned}&Link(url_1, url_2) \;\Leftrightarrow\\ &\quad InTag(\text{“}a\text{”}, str, GetPage(utl_1)) \wedge In(\text{“}href = \text{“”} + url_2 + \text{“””}, str)\\ &LinkText(url_1, url_2, text) \;\Leftrightarrow\\ &\quad InTag(\text{“}a\text{”}, str, GetPage(utl_1)) \wedge In(\text{“}href = \text{“”} + url_2 + \text{“””} + text, str)\end{aligned}$$

10.14 Here is an initial sketch of one approach. (Others are possible.) A given object to be purchased may *require* some additional parts (e.g., batteries) to be functional, and there may also be *optional* extras. We can represent requirements as a relation between an individual object and a class of objects, qualified by the number of objects required:

$$\forall\, x\;\; x \in Coolpix995DigitalCamera \;\Rightarrow\; Requires(x, AABattery, 4)\;.$$

We also need to know that a particular object is compatible, i.e., fills a given role appropriately. For example,

$$\begin{aligned}&\forall\, x,y\;\; x \in Coolpix995DigitalCamera \wedge y \in DuracellAABattery\\ &\qquad\Rightarrow\; Compatible(y, x, AABattery)\end{aligned}$$

Then it is relatively easy to test whether the set of ordered objects contains compatible required objects for each object.

10.15 Plurals can be handled by a $Plural$ relation between strings, e.g.,

$$Plural(\text{“}computer\text{”}, \text{“}computers\text{”})$$

plus an assertion that the plural (or singular) of a name is also a name for the same category:

$$\forall\, c, s_1, s_2\;\; Name(s_1, c) \wedge (Plural(s_1, s_2) \vee Plural(s_2, s_1)) \;\Rightarrow\; Name(s_2, c)$$

Conjunctions can be handled by saying that any conjunction string is a name for a category if one of the conjuncts is a name for the category:

$$\forall\, c, s, s_2 \;\; \mathit{Conjunct}(s_2, s) \land \mathit{Name}(s_2, c) \;\Rightarrow\; \mathit{Name}(s, c)$$

where $\mathit{Conjunct}$ is defined appropriately in terms of concatenation. Probably it would be better to redefine $\mathit{RelevantCategoryName}$ instead.

10.16 Chapter 22 explains how to use logic to parse text strings and extract semantic information. The outcome of this process is a definition of what objects are acceptable to the user for a specific shopping request; this allows the agent to go out and find offers matching the user's requirements. We omit the full definition of the agent, although a skeleton may appear on the AIMA project web pages.

10.17 Here is a simple version of the answer; it can be elaborated *ad infinitum.* Let the term $\mathit{Buy}(b, x, s, p)$ denote the event category of buyer b buying object x from seller s for price p. We want to say about it that b transfers the money to s, and s transfers ownership of x to b.

$$\begin{aligned}&T(\mathit{Buy}(b, x, s, p), i) \;\Leftrightarrow\\ &\quad T(\mathit{Owns}(s, x), \mathit{Start}(i)) \land\\ &\quad \exists\, m \;\; \mathit{Money}(m) \land p = \mathit{Value}(m) \land T(\mathit{Owns}(b, m), \mathit{Start}(i)) \land\\ &\quad T(\mathit{Owns}(b, x), \mathit{End}(i)) \land T(\mathit{Owns}(s, m), \mathit{End}(i))\end{aligned}$$

10.18 Let $\mathit{Trade}(b, x, a, y)$ denote the class of events where person b trades object y to person a for object x:

$$\begin{aligned}&T(\mathit{Trade}(b, x, a, y), i) \;\Leftrightarrow\\ &\quad T(\mathit{Owns}(b, y), \mathit{Start}(i)) \land T(\mathit{Owns}(a, x), \mathit{Start}(i)) \land\\ &\quad T(\mathit{Owns}(b, x), \mathit{End}(i)) \land T(\mathit{Owns}(a, y), \mathit{End}(i))\end{aligned}$$

Now the only tricky part about defining buying in terms of trading is in distinguishing a price (a measurement) from an actual collection of money.

$$T(\mathit{Buy}(b, x, a, p), i) \;\Leftrightarrow\; \exists\, m \;\; \mathit{Money}(m) \land \mathit{Trade}(b, x, a, m) \land \mathit{Value}(m) = p$$

10.19 There are many possible approaches to this exercise. The idea is for the students to think about doing knowledge representation for real; to consider a host of complications and find some way to represent the facts about them. Some of the key points are:

- Ownership occurs over time, so we need either a situation-calculus or interval-calculus approach.
- There can be joint ownership and corporate ownership. This suggests the owner is a group of some kind, which in the simple case is a group of one person.
- Ownership provides certain rights: to use, to resell, to give away, etc. Much of this is outside the definition of ownership *per se*, but a good answer would at least consider how much of this to represent.
- Own can own abstract obligations as well as concrete objects. This is the idea behind the futures market, and also behind banks: when you deposit a dollar in a bank, you are giving up ownership of that particular dollar in exchange for ownership of the right

to withdraw another dollar later. (Or it could coincidentally turn out to be the exact same dollar.) Leases and the like work this way as well. This is tricky in terms of representation, because it means we have to reify transactions of this kind. That is, $Withdraw(person, money, bank, time)$ must be an object, not a predicate.

10.20 Most schools distinguish between required courses and elected courses, and between courses inside the department and outside the department. For each of these, there may be requirements for the number of courses, the number of units (since different courses may carry different numbers of units), and on grade point averages. We show our chosen vocabulary by example:

- Student Jones' complete course of study for the whole college career consists of Math1, CS1, CS2, CS3, CS21, CS33 and CS34, and some other courses outside the major.

$$\begin{array}{l} Take(Jones, \\ \;\{Math1, EE1, Bio24, CS1, CS2, CS3, CS21, CS33, CS34|others\}) \end{array}$$

- Jones meets the requirements for a major in Computer Science

$$Major(Jones, CS)$$

- Courses Math1, CS1, CS2, and CS3 are required for a Computer Science major.

$$\begin{array}{l} Required(\{Math1, CS1, CS2, CS3\}, CS) \\ \forall\, s, d \;\; Required(s, d) \;\Leftrightarrow \\ \qquad (\forall\, p \;\; \exists\, others \;\; Major(p, d) \;\Rightarrow\; Take(p, Union(s, others))) \end{array}$$

- A student must take at least 18 units in the CS department to get a degree in CS.

$$\begin{array}{l} Department(CS1) = CS \wedge Department(Math1) = Math \wedge \ldots \\ Units(CS1) = 3 \wedge Units(CS2) = 4 \wedge \ldots \\ RequiredUnitsIn(18, CS, CS) \\ \forall\, u, d \;\; RequiredUnitsIn(u, d) \;\Leftrightarrow \\ \qquad (\forall\, p \;\; \exists\, s, others \;\; Major(p, d) \;\Rightarrow\; Take(p, Union(s, others)) \\ \qquad\quad \wedge\, AllInDepartment(s, d) \wedge TotalUnits(s) \geq u \\ \forall\, s, d \;\; AllInDepartment(s, d) \;\Leftrightarrow\; (\forall\, c \;\; c \in s \;\Rightarrow\; Department(c) = d) \\ \forall\, c \;\; TotalUnits(\{\}) = 0 \\ \forall\, c, s \;\; TotalUnits(\{c|s\}) = Units(c) + TotalUnits(s) \end{array}$$

One can easily imagine other kinds of requirements; these just give you a flavor.

In this solution we took "over an extended period" to mean that we should recommend a set of courses to take, without scheduling them on a semester-by-semester basis. If you wanted to do that, you would need additional information such as when courses are taught, what is a reasonable course load in a semester, and what courses are prerequisites for what others. For example:

$$\begin{array}{l} Taught(CS1, Fall) \\ Prerequisites(\{CS1, CS2\}, CS3) \\ TakeInSemester(Jones, Fall95, \{Math1, CS1, English1, History1\}) \\ MaxCoursesPerSemester(5) \end{array}$$

The problem with finding the *best* program of study is in defining what *best* means to the student. It is easy enough to say that all other things being equal, one prefers a good teacher to a bad one, or an interesting course to a boring one. But how do you decide which is best when one course has a better teacher and is expected to be easier, while an alternative is more interesting and provides one more credit? Chapter 16 uses utility theory to address this. If you can provide a way of weighing these elements against each other, then you can choose a best program of study; otherwise you can only eliminate some programs as being worse than others, but can't pick an absolute best one. Complexity is a further problem: witha general-purpose theorem-prover it's hard to do much more than enumerate legal programs and pick the best.

10.21 This exercise and the following two are rather complex, perhaps suitable for term projects. At this point, we want to strongly urge that you do assign some of these exercises (or ones like them) to give your students a feeling of what it is really like to do knowledge representation. In general, students find classification hierarchies easier than other representation tasks. A recent twist is to compare one's hierarchy with online ones such as `yahoo.com`.

10.22 This is the most involved representation problem. It is suitable for a group project of 2 or 3 students over the course of at least 2 weeks.

10.23 Normally one would assign 10.22 in one assignment, and then when it is done, add this exercise (posibly varying the questions). That way, the students see whether they have made sufficient generalizations in their initial answer, and get experience with debugging and modifying a knowledge base.

10.24 In many AI and Prolog textbooks, you will find it stated plainly that implications suffice for the implementation of inheritance. This is true in the logical but not the practical sense.

a. Here are three rules, written in Prolog. We actually would need many more clauses on the right hand side to distinguish between different models, different options, etc.

```
worth(X,575) :- year(X,1973), make(X,dodge), style(X,van).
worth(X,27000) :- year(X,1994), make(X,lexus), style(X,sedan).
worth(X,5000) :- year(X,1987), make(X,toyota), style(X,sedan).
```

To find the value of JB, given a data base with `year(jb,1973)`,`make(jb,dodge)` and `style(jb,van)` we would call the backward chainer with the goal `worth(jb,D)`, and read the value for `D`.

b. The time efficiency of this query is $O(n)$, where n in this case is the 11,000 entries in the Blue Book. A semantic network with inheritance would allow us to follow a link from `JB` to `1973-dodge-van`, and from there to follow the `worth` slot to find the dollar value in $O(1)$ time.

c. With forward chaining, as soon as we are told the three facts about JB, we add the new fact `worth(jb,575)`. Then when we get the query `worth(jb,D)`, it is $O(1)$ to find the answer, assuming indexing on the predicate and first argument. This makes logical inference seem just like semantic networks except for two things: the logical inference does a hash table lookup instead of pointer following, and logical inference explicitly stores `worth` statements for each individual car, thus wasting space if there are a lot of individual cars. (For this kind of application, however, we will probably want to consider only a few individual cars, as opposed to the 11,000 different models.)

d. If each category has many properties—for example, the specifications of all the replacement parts for the vehicle—then forward-chaining on the implications will also be an impractical way to figure out the price of a vehicle.

e. If we have a rule of the following kind:

```
worth(X,D) :- year-make-style(X,Yr,Mk,St),
              year-make-style(Y,Yr,Mk,St), worth(Y,D).
```

together with facts in the database about some other specific vehicle of the same type as JB, then the query `worth(jb,D)` will be solved in $O(1)$ time with appropriate indexing, regardless of how many other facts are known about that type of vehicle and regardless of the number of types of vehicle.

10.25 When categories are reified, they can have properties as individual objects (such as $Cardinality$ and $Supersets$) that do not apply to their elements. Without the distinction between boxed and unboxed links, the sentence $Cardinality(SingletonSets, 1)$ might mean that every singleton set has one element, or that there si only one singleton set.

Solutions for Chapter 11 Planning

11.1 Both problem solver and planner are concerned with getting from a start state to a goal using a set of defined operations or actions. But in planning we open up the representation of states, goals, and plans, whcihc allows for a wider variety of algorithms that decompose the search space.

11.2 This is an easy exercise, the point of which is to understand that "applicable" means satisfying the preconditions, and that a concrete action instance is one with the variables replaced by constants. The applicable actions are:

$$\begin{aligned}
&Fly(P_1, JFK, SFO)\\
&Fly(P_1, JFK, JFK)\\
&Fly(P_2, SFO, JFK)\\
&Fly(P_2, SFO, SFO)
\end{aligned}$$

A minor point of this is that the action of flying nowhere—from one airport to itself—is allowable by the definition of Fly, and is applicable (if not useful).

11.3 For the regular schema we have:

$$\begin{aligned}
&FlyPrecond(p, f, to, s) \;\Leftrightarrow\\
&\qquad At(p, f, s) \land Plane(p) \land Airport(f) \land Airport(to)\\
&At(p, x, Result(a, s)) \;\Leftrightarrow\\
&\qquad (At(p, x, s) \land (a \neq Fly(p, f, x) \lor \neg FlyPrecond(p, f, x, s)))\\
&\qquad \lor At(p, f, s) \land a = Fly(p, f, x) \land FlyPrecond(p, f, x, s)
\end{aligned}$$

When we add $Warped$ we get:

$$\begin{aligned}
&At(p, x, Result(a, s)) \;\Leftrightarrow\\
&\qquad (At(p, x, s) \land (a \neq Fly(p, f, x) \land a \neq Teleport(p, f, x))\\
&\qquad\qquad \lor a = Fly(p, f, x) \land \neg FlyPrecond(p, f, x, s)\\
&\qquad\qquad \lor a = Teleport(p, f, x) \land \neg TeleportPrecond(p, f, x, s)\\
&\qquad \lor At(p, f, s) \land a = Fly(p, f, x) \land FlyPrecond(p, f, x, s)\\
&\qquad \lor At(p, f, s) \land a = Teleport(p, f, x) \land TeleportPrecond(p, f, x, s)
\end{aligned}$$

In general, we (1) created a $Precond$ predicate for each action, and then, for each fluent such as At, we create a predicate that says the fluent keeps its old value if either an irrelevant action is taken, or an action whose precondition is not satisfied, and it takes on a new value according to the effects of a relevant action if that action's preconditions are satisfied.

11.4 This exercise is intended as a fairly easy exercise in describing a domain. It is similar to the Shakey problem (11.13), so you should probably assign only one of these two.

a. The initial state is:

$$\begin{array}{l} At(Monkey, A) \land At(Bananas, B) \land At(Box, C) \land \\ Height(Monkey, Low) \land Height(Box, Low) \land Height(Bananas, High) \land \\ Pushable(Box) \land Climbable(Box) \end{array}$$

b. The actions are:

$$\begin{array}{l} Action(\textsc{Action}{:}Go(x,y), \textsc{Precond}{:}At(Monkey, x), \\ \quad \textsc{Effect}{:}At(Monkey, y) \land \neg(At(Monkey, x))) \\ Action(\textsc{Action}{:}Push(b,x,y), \textsc{Precond}{:}At(Monkey, x) \land Pushable(b), \\ \quad \textsc{Effect}{:}At(b, y) \land At(Monkey, y) \land \neg At(b, x) \land \neg At(Monkey, x)) \\ Action(\textsc{Action}{:}ClimbUp(b), \\ \quad \textsc{Precond}{:}At(Monkey, x) \land At(b, x) \land Climbable(b), \\ \quad \textsc{Effect}{:}On(Monkey, b) \land \neg Height(Monkey, High)) \\ Action(\textsc{Action}{:}Grasp(b), \\ \quad \textsc{Precond}{:}Height(Monkey, h) \land Height(b, h) \\ \quad\quad \land At(Monkey, x) \land At(b, x), \\ \quad \textsc{Effect}{:}Have(Monkey, b)) \\ Action(\textsc{Action}{:}ClimbDown(b), \\ \quad \textsc{Precond}{:}On(Monkey, b) \land Height(Monkey, High), \\ \quad \textsc{Effect}{:}\neg On(Monkey, b) \land \neg Height(Monkey, High) \\ \quad\quad \land Height(Monkey, Low) \\ Action(\textsc{Action}{:}UnGrasp(b), \textsc{Precond}{:}Have(Monkey, b), \\ \quad \textsc{Effect}{:}\neg Have(Monkey, b)) \end{array}$$

c. In situation calculus, the goal is a state s such that:

$$Have(Monkey, Bananas, s) \land (\exists\, x \;\; At(Box, x, s_0) \land At(Box, x, s))$$

In STRIPS, we can only talk about the goal state; there is no way of representing the fact that there must be some relation (such as equality of location of an object) between two states within the plan. So there is no way to represent this goal.

d. Actually, we did include the $Pushable$ precondition. This is an example of the qualification problem.

11.5 Only positive literals are represented in a state. So not mentioning a literal is the same as having it be negative.

11.6 Goals and preconditions can only be positive literals. So a negative effect can only make it harder to achieve a goal (or a precondition to an action that achieves the goal). Therefore, eliminating all negative effects only makes a problem easier.

11.7

a. It is feasible to use bidirectional search, because it is possible to invert the actions. However, most of those who have tried have concluded that biderectional search is generally not efficient, because the forward and backward searches tend to miss each other. This is due to the large state space. A few planners, such as PRODIGY (Fink and Blythe, 1998) have used bidirectional search.

b. Again, this is feasible but not popular. PRODIGY is in fact (in part) a partial-order planner: in the forward direction it keeps a total-order plan (equivalent to a state-based planner), and in the backward direction it maintains a tree-structured partial-order plan.

c. An action A can be added if all the preconditions of A have been achieved by other steps in the plan. When A is added, ordering constraints and causal links are also added to make sure that A appears after all the actions that enabled it and that a precondition is not disestablished before A can be executed. The algorithm does search forward, but it is not the same as forward state-space search because it can explore actions in parallel when they don't conflict. For example, if A has three preconditions that can be satisfied by the non-conflicting actions B, C, and D, then the solution plan can be represented as a single partial-order plan, while a state-space planner would have to consider all $3!$ permutations of B, C, and D.

d. Yes, this is one possible way of implementing a bidirectional search in the space of partial-order plans.

11.8 The drawing is actually rather complex, and doesn't fit well on this page. Some key things to watch out for: (1) Both Fly and $Load$ actions are possible at level A_0; the planes can still fly when empty. (2) Negative effects appear in S_1, and are mutex with their positive counterparts.

11.9

a. Literals are persistent, so if it does not appear in the final level, it never will and never did, and thus cannot be achieved.

b. In a serial planning graph, only one action can occur per time step. The level cost (the level at which a literal first appears) thus represents the minimum number of actions in a plan that might possibly achieve the literal.

11.10 A forward state-space planner maintains a partial plan that is a strict linear sequence of actions; the plan refinement operator is to add an applicable action to the end of the sequence, updating literals according to the action's effects.

A backward state-space planner maintains a partial plan that is a reversed sequence of actions; the refinement operator is to add an action to the beginning of the sequence as long as the action's effects are compatible with the state at the beginning of the sequence.

11.11 The initial state is:

$$On(B, Table) \land On(C, A) \land On(A, Table) \land Clear(B) \land Clear(C)$$

The goal is:

$$On(A, B) \land On(B, C)$$

First we'll explain why it is an anomaly for a noninterleaved planner. There are two subgoals; suppose we decide to work on $On(A, B)$ first. We can clear C off of A and then move A

on to B. But then there is no way to achieve $On(B, C)$ without undoing the work we have done. Similarly, if we work on the subgoal $On(B, C)$ first we can immediately achieve it in one step, but then we have to undo it to get A on B.

Now we'll show how things work out with an interleaved planner such as POP. Since $On(A, B)$ isn't true in the initial state, there is only one way to achieve it: $Move(A, x, B)$, for some x. Similarly, we also need a $Move(B, x', C)$ step, for some x'. Now let's look at the $Move(A, x, B)$ step. We need to achieve its precondition $Clear(A)$. We could do that either with $Move(b, A, y)$ or with $MoveToTable(b, A)$. Let's assume we choose the latter. Now if we bind b to C, then all of the preconditions for the step $MoveToTable(C, A)$ are true in the initial state, and we can add causal links to them. We then notice that there is a threat: the $Move(B, x', C)$ step threatens the $Clear(C)$ condition that is required by the $MoveToTable$ step. We can resolve the threat by ordering $Move(B, x', C)$ after the $MoveToTable$ step. Finally, notice that all the preconditions for $Move(B, x', C)$ are true in the initial state. Thus, we have a complete plan with all the preconditions satisfied. It turns out there is a well-ordering of the three steps:

$MoveToTable(C, A)$
$Move(B, Table, C)$
$Move(A, Table, B)$

11.12 The actions we need are the four from page 346:

$Action(\textsc{Action}{:}RightShoe, \textsc{Precond}{:}RightSockOn, \textsc{Effect}{:}RightShoeOn)$
$Action(\textsc{Action}{:}RightSock, \textsc{Effect}{:}RightSockOn)$
$Action(\textsc{Action}{:}LeftShoe, \textsc{Precond}{:}LeftSockOn, \textsc{Effect}{:}LeftShoeOn)$
$Action(\textsc{Action}{:}LeftSock, \textsc{Effect}{:}LeftSockOn)$

One solution found by GRAPHPLAN is to execute *RightSock* and *LeftSock* in the first time step, and then *RightShoe* and *LeftShoe* in the second.

Now we add the following two actions (neither of which has preconditions):

$Action(\textsc{Action}{:}Hat, \textsc{Effect}{:}HatOn)$
$Action(\textsc{Action}{:}Coat, \textsc{Effect}{:}CoatOn)$

The partial-order plan is shown in Figure S11.1. We saw on page 348 that there are 6 total-order plans for the shoes/socks problem. Each of these plans has four steps, and thus five arrow links. The next step, *Hat* could go at any one of these five locations, giving us $6 \times 5 = 30$ total-order plans, each with five steps and six links. Then the final step, *Coat*, can go in any one of these 6 positions, giving us $30 \times 6 = 180$ total-order plans.

11.13 The actions are quite similar to the monkey and banannas problem—you should prob-

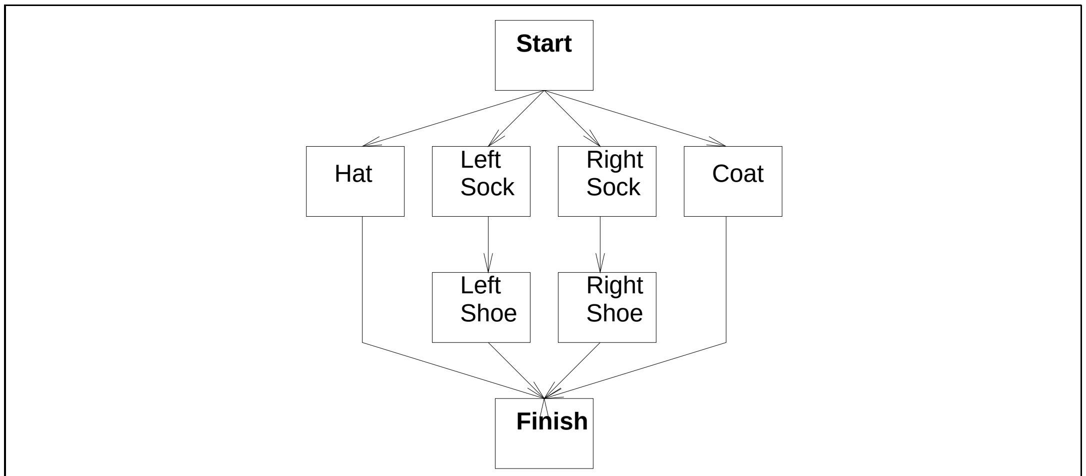

Figure S11.1 Partial-order plan including a hat and coat, for Exercise 11.1.

ably assign only one of these two problems. The actions are:

$$
\begin{aligned}
&\mathit{Action}(\textsc{Action}{:}\mathit{Go}(x,y), \textsc{Precond}{:}\mathit{At}(\mathit{Shakey},x) \wedge \mathit{In}(x,r) \wedge \mathit{In}(y,r),\\
&\quad \textsc{Effect}{:}\mathit{At}(\mathit{Shakey},y) \wedge \neg(\mathit{At}(\mathit{Shakey},x)))\\
&\mathit{Action}(\textsc{Action}{:}\mathit{Push}(b,x,y), \textsc{Precond}{:}\mathit{At}(\mathit{Shakey},x) \wedge \mathit{Pushable}(b),\\
&\quad \textsc{Effect}{:}\mathit{At}(b,y) \wedge \mathit{At}(\mathit{Shakey},y) \wedge \neg \mathit{At}(b,x) \wedge \neg \mathit{At}(\mathit{Shakey},x))\\
&\mathit{Action}(\textsc{Action}{:}\mathit{ClimbUp}(b), \textsc{Precond}{:}\mathit{At}(\mathit{Shakey},x) \wedge \mathit{At}(b,x) \wedge \mathit{Climbable}(b),\\
&\quad \textsc{Effect}{:}\mathit{On}(\mathit{Shakey},b) \wedge \neg \mathit{On}(\mathit{Shakey},\mathit{Floor}))\\
&\mathit{Action}(\textsc{Action}{:}\mathit{ClimbDown}(b), \textsc{Precond}{:}\mathit{On}(\mathit{Shakey},b),\\
&\quad \textsc{Effect}{:}\mathit{On}(\mathit{Shakey},\mathit{Floor}) \wedge \neg \mathit{On}(\mathit{Shakey},b))\\
&\mathit{Action}(\textsc{Action}{:}\mathit{TurnOn}(l), \textsc{Precond}{:}\mathit{On}(\mathit{Shakey},b) \wedge \mathit{At}(\mathit{Shakey},x) \wedge \mathit{At}(l,x),\\
&\quad \textsc{Effect}{:}\mathit{TurnedOn}(l))\\
&\mathit{Action}(\textsc{Action}{:}\mathit{TurnOff}(l), \textsc{Precond}{:}\mathit{On}(\mathit{Shakey},b) \wedge \mathit{At}(\mathit{Shakey},x) \wedge \mathit{At}(l,x),\\
&\quad \textsc{Effect}{:}\neg \mathit{TurnedOn}(l))
\end{aligned}
$$

The initial state is:

$$
\begin{aligned}
&\mathit{In}(\mathit{Switch}_1,\mathit{Room}_1) \wedge \mathit{In}(\mathit{Door}_1,\mathit{Room}_1) \wedge \mathit{In}(\mathit{Door}_1,\mathit{Corridor})\\
&\mathit{In}(\mathit{Switch}_1,\mathit{Room}_2) \wedge \mathit{In}(\mathit{Door}_2,\mathit{Room}_2) \wedge \mathit{In}(\mathit{Door}_2,\mathit{Corridor})\\
&\mathit{In}(\mathit{Switch}_1,\mathit{Room}_3) \wedge \mathit{In}(\mathit{Door}_3,\mathit{Room}_3) \wedge \mathit{In}(\mathit{Door}_3,\mathit{Corridor})\\
&\mathit{In}(\mathit{Switch}_1,\mathit{Room}_4) \wedge \mathit{In}(\mathit{Door}_4,\mathit{Room}_4) \wedge \mathit{In}(\mathit{Door}_4,\mathit{Corridor})\\
&\mathit{In}(\mathit{Shakey},\mathit{Room}_3) \wedge \mathit{At}(\mathit{Shakey},X_S)\\
&\mathit{In}(\mathit{Box}_1,\mathit{Room}_1) \wedge \mathit{In}(\mathit{Box}_2,\mathit{Room}_1) \wedge \mathit{In}(\mathit{Box}_3,\mathit{Room}_1) \wedge \mathit{In}(\mathit{Box}_4,\mathit{Room}_1)\\
&\mathit{Climbable}(\mathit{Box}_1) \wedge \mathit{Climbable}(\mathit{Box}_2) \wedge \mathit{Climbable}(\mathit{Box}_3) \wedge \mathit{Climbable}(\mathit{Box}_4)\\
&\mathit{Pushable}(\mathit{Box}_1) \wedge \mathit{Pushable}(\mathit{Box}_2) \wedge \mathit{Pushable}(\mathit{Box}_3) \wedge \mathit{Pushable}(\mathit{Box}_4)\\
&\mathit{At}(\mathit{Box}_1,X_1) \wedge \mathit{At}(\mathit{Box}_2,X_2) \wedge \mathit{At}(\mathit{Box}_3,X_3) \wedge \mathit{At}(\mathit{Box}_4,X_4)\\
&\mathit{TurnwdOn}(\mathit{Switch}_1) \wedge \mathit{TurnedOn}(\mathit{Switch}_4)
\end{aligned}
$$

A plan to achieve the goal is:

$Go(X_S, Door_3)$
$Go(Door_3, Door_1)$
$Go(Door_1, X_2)$
$Push(Box_2, X_2, Door_1)$
$Push(Box_2, Door_1, Door_2)$
$Push(Box_2, Door_2, Switch_2)$

11.14 GRAPHPLAN is a propositional algorithm, so, just as we could solve certain FOL by translating them into propositional logic, we can solve certain situation calculus problems by translating into propositional form. The trick is how exactly to do that.

The *Finish* action in POP planning has as its preconditions the goal state. We can create a *Finish* action for GRAPHPLAN, and give it the effect *Done*. In this case there would be a finite number of instantiations of the *Finish* action, and we would reason with them.

11.15 (Figure (11.1) is a little hard to find—it is on page 403.)

a. The point of this exercise is to consider what happens when a planner comes up with an impossible action, such as flying a plane from someplace where it is not. For example, suppose P_1 is at JFK and we give it the action of flying from Bangalore to Brisbane. By (11.1), P_1 was at JFK and did not fly away, so it is still there.

b. Yes, the plan will still work, because the fluents hold from the situation before an inapplicable action to the state afterward, so the plan can continue from that state.

c. It depends on the details of how the axioms are written. Our axioms were of the form *Action is possible* $\Rightarrow$ *Rule*. This tells us nothing about the case where the action is not possible. We would need to reformulate the axioms, or add additional ones to say what happens when the action is not possible.

11.16 A **precondition axiom** is of the form

$$Fly(P_1, JFK, SFO)^0 \Rightarrow At(P_1, JFK)^0.$$

There are $O(T \times |P| \times |A|^2)$ of these axioms, where T is the number of time steps, $|P|$ is the number of planes and $|A|$ is the number of airports. More generally, if there are n action schemata of maximum arity k, with $|O|$ objects, then there are $O(n \times T \times |O|^k)$ axioms.

With symbol-splitting, we don't have to describe each specific flight, we need only say that for a plane to fly *anywhere*, it must be at the start airport. That is,

$$Fly_1(P_1)^0 \wedge Fly_2(JFK)^0 \Rightarrow At(P_1, JFK)^0$$

More generally, if there are n action schemata of maximum arity k, with $|O|$ objects, and each precondition axiom depends on just two of the arguments, then there are $O(n \times T \times |O|^2)$ axioms, for a speedup of $O(|O|^{k-2})$.

An **action exclusion axiom** is of the form

$$\neg(Fly(P_2, JFK, SFO)^0 \wedge Fly(P_2, JFK, LAX)^0).$$

With the notation used above, there are $O(T \times |P| \times |A|^3)$ axioms for Fly. More generally, there can be up to $O(n \times T \times |O|^{2k})$ axioms.

With symbol-splitting, we wouldn't gain anything for the Fly axioms, but we would gain in cases where there is another variable that is not relevant to the exclusion.

11.17

a. Yes, this will find a plan whenever the normal SATPLAN finds a plan no longer than T_{max}.

b. No.

c. There is no simple and clear way to induce WALKSAT to find short solutions, because it has no notion of the length of a plan—the fact that the problem is a planning problem is part of the encoding, not part of WALKSAT. But if we are willing to do some rather brutal surgery on WALKSAT, we can achieve shorter solutions by identifying the variables that represent actions and (1) tending to randomly initialize the action variables (particularly the later ones) to false, and (1) prefering to randomly flip an earlier action variable rather than a later one.

Solutions for Chapter 12
Planning and Acting in the Real World

12.1

a. $Duration(d)$ is ***eligible*** to be an effect because the action does have the effect of moving the clock by d. It is possible that the duration depends on the action outcome, so if disjunctive or conditional effects are used there must be a way to associate durations with outcomes, which is most easily done by putting the duration into the outcome expression.

b. The STRIPS model assumes that actions are time points characterized only by their preconditions and effects. Even if an action occupies a resource, that has no effect on the outcome state (as explained on page 420). Therefore, we must extend the STRIPS formalism. We could do this by treating a RESOURCE: effect differently from other effects, but the difference is sufficiently large that it makes more sense to treat it separately.

12.2 The basic idea here is to record the initial resource level in the precondition and the change in resource level in the effect of each action.

a. Let $Screws(s)$ denote the fact that there are s screws. We need to add $Screws(100)$ to the initial state, and add a fourth argument to the $Engine$ predicate indicating the number of screws required—i.e., $Engine(E_1, C_1, 30, 40)$ and $Engine(E_2, C_2, 60, 50)$. We add $Screws(s_0)$ to the precondition of $AddEngine$ and add s as a fourth argument of the $Engine$ literal. Then add $Screws(s_0 - s)$ to the effect of $AddEngine$.

b. A simple solution is to say that any action that consumes a resource is potentially in conflict with any causal link protecting the same resource.

c. The planner can keep track of the resource requirements of actions added to the plan and backtrack whenever the total usage exceeds the initial amount.

12.3 There is a wide range of possible answers to this question. The important point is that students understand what constitutes a correct implementation of an action: as mentioned on page 424, it must be a consistent plan where all the preconditions and effects are accounted for. So the first thing we need is to decide on the preconditions and effects of the high-level actions. For *GetPermit*, assume the precondition is owning land, and the effect is having a permit for that piece of land. For *HireBuilder*, the precondition is having the ability to pay, and the effect is having a signed contract in hand.

One possible decomposition for *GetPermit* is the three-step sequence *GetPermitForm*, *FillOutForm*, and *GetFormApproved*. There is a causal link with the condition *HaveForm* between the first two, and one with the condition *HaveCompletedForm* between the last two. Finally, the *GetFormApproved* step has the effect *HavePermit*. This is a valid decomposition.

For *HireBuilder*, suppose we choose the three-step sequence *InterviewBuilders*, *Choose-Builder*, and *SignContract*. This last step has the precondition *AbleToPay* and the effect *Have-ContractInHand*. There are also causal links between the substeps, but they don't affect the correctness of the decomposition.

12.4 Consider the problem of building two adjacent walls of the house. Mostly these sub-plans are independent, but they must share the step of putting up a common post at the corner of the two walls. If that step was not shared, we would end up with an extra post, and two unattached walls.

Note that tasks are often decomposed specifically so as to minimize the amount of step sharing. For example, one could decompose the house building task into subtasks such as "walls" and "floors." However, real contractors don't do it that way. Instead they have "rough walls" and "rough floors" steps, followed by a "finishing" step.

12.5 In the HTN view, the space of possible decompositions may constrain the allowable solutions, eliminating some possible sequences of primitive actions. For example, the de-composition of the *LAToNYRoundTrip* action can stipulate that the agent should go to New York. In a simple STRIPS formulation where the start and goal states are the same, the empty plan is a solution. We can get around this problem by rethinking the goal description. The goal state is not $At(LA)$, but $At(LA) \land Visited(NY)$. We add $Visited(y)$ as an effect of $Fly(x, y)$. Then, the solution must be a trip that includes New York. There remains the prob-lem of preventing the STRIPS plan from including other stops on its itinerary; fixing this is much more difficult because negated goals are not allowed.

12.6 Suppose we have a STRIPS action description for a with precondition p and effect q. The "action" to be decomposed is $Achieve(q)$. The decomposition has two steps: $Achieve(p)$ and a. This can be extended in the obvious way for conjunctive effects and precondi-tions.

12.7 We need one action, $Assign$, which assigns the value in the source register (or variable if you prefer, but the term "register" makes it clearer that we are dealing with a physical location) sr to the destination register dr:

$$\begin{aligned}
&Action(\text{ACTION:}Assign(dr, sr),\\
&\quad \text{PRECOND:}Register(dr) \land Register(sr) \land Value(dr, dv) \land Value(sr, sv),\\
&\quad \text{EFFECT:}Value(dr, sv) \land \neg Value(dr, dv))
\end{aligned}$$

Now suppose we start in an initial state with $Register(R_1) \land Register(R_2) \land Value(R_1, V_1) \land Value(R_2, V_2)$ and we have the goal $Value(R_1, V_2) \land Value(R_2, V_1)$. Unfortunately, there is no way to solve this as is. We either need to add an explicit $Register(R_3)$ condition to the initial state, or we need a way to create new registers. That could be done with an action for allocating a new register:

$Action($ACTION:$Allocate(r),$
  EFFECT:$Register(r))$

Then the following sequence of steps constitues a valid plan:

$Allocate(R_3)$
$Assign(R_3, R_1)$
$Assign(R_1, R_2)$
$Assign(R_2, R_1)$

12.8 For the first case, where one instance of action schema a is in the plan, the reformulation is correct, in the sense that a solution for the original disjunctive formulation is a solution for the new formulation and *vice versa*. For the second case, where more than one instance of the action schema may occur, the reformulation is incorrect. It assumes that the outcomes of the instances are governed by a single hidden variable, so that if, for example, P is the outcome of one instance it must also be the outcome of the other. It is possible that a solution for the reformulated case will fail in the original formulation.

12.9 With unbounded indeterminacy, the set of possible effects for each action is unknown or too large to be enumerated. Hence, the space of possible actions sequences required to handle all these eventualities is far too large to consider.

12.10 Using the second definition of $Clear$ in the chapter—namely, that there is a clear space for a block—the only change is that the destination remains clear if it is the table:

$Action(Move(b, x, y),$
  PRECOND:$On(b, x) \land Clear(b) \land Clear(y),$
  EFFECT:$On(b, y) \land Clear(x) \land \neg On(b, x) \land ($**when** $y \neq Table\colon \neg Clear(y)))$

12.11 Let $CleanH$ be true iff the robot's current square is clean and $CleanO$ be true iff the other square is clean. Then $Suck$ is characterized by

$Action(Suck,$ PRECOND:, EFFECT:$CleanH)$

Unfortunately, moving affects these new literals! For $Left$ we have

$Action(Left,$ PRECOND:$AtR,$
  EFFECT:$AtL \land \neg AtR \land$ **when** $CleanH\colon CleanO \land$ **when** $CleanO\colon CleanH$
    $\land$ **when** $\neg CleanO\colon \neg CleanH \land$ **when** $\neg CleanH\colon \neg CleanO)$

with the dual for $Right$.

12.12 Here we borrow from the last description of the $Left$ on page 433:

$Action(Suck,$ PRECOND:,
  EFFECT:$($**when** $AtL\colon CleanL \lor ($**when** $CleanL\colon \neg CleanL))$
    $\land ($**when** $AtR\colon CleanR \lor ($**when** $CleanR\colon \neg CleanR)))$

12.13 The main thing to notice here is that the vacuum cleaner moves repeatedly over dirty areas—presumably, until they are clean. Also, each forward move is typically short, followed by an immediate reversing over the same area. This is explained in terms of a disjunctive outcome: the area may be fully cleaned or not, the reversing enables the agent to check, and the repetition ensures completion (unless the dirt is ingrained). Thus, we have a strong cyclic plan with sensing actions.

12.14

a. "Lather. Rinse. Repeat."
This is an unconditional plan, if taken literally, involving an infinite loop. If the precondition of $Lather$ is $\neg Clean$, and the goal is $Clean$, then execution monitoring will cause execution to terminate once $Clean$ is achieved because at that point the correct repair is the empty plan.

b. "Apply shampoo to scalp and let it remain for several minutes. Rinse and repeat if necessary."
This is a conditional plan where "if necessary" presumably tests $\neg Clean$.

c. "See a doctor if problems persist."
This is also a conditional step, although it is not specified here what problems are tested.

12.17 First, we need to decide if the precondition is satisfied. There are three cases:

a. If it is known to be unsatisfied, the new belief state is identical to the old (since we assume nothing happens).

b. If it is known to be satisfied, the unconditional effects (which are all knowledge propositions) are added and deleted from the belief state in the usual STRIPS fashion. Each conditional effect whose condition is known to be true is handled in the same way. For each setting of the unknown conditions, we create a belief state with the appropriate additions and deletions.

c. If the status of the precondition is unknown, each new belief state is effectively the disjunction of the unchanged belief state from (a) with one of the belief states obtained from (b). To enforce the "list of knowledge propositions" representation, we keep those propositions that are identical in each of the two belief states being disjoined and discard those that differ. This results in a weaker belief state than if we were to retain the disjunction; on the other hand, retaining the disjunctions over many steps could lead to exponentially large representations.

12.18 For $Right$ we have the obvious dual version of Equation 12.2:

$$\begin{aligned}&Action(Right, \textsc{Precond}{:}AtL,\\ &\qquad \textsc{Effect}{:}K(AtR) \land \neg K(AtL) \land \textbf{when } CleanL{:}\ \neg K(CleanL) \land\\ &\qquad\qquad \textbf{when } CleanR{:}\ K(CleanR) \land \textbf{when } \neg CleanR{:}\ K(\neg CleanR))\end{aligned}$$

With $Suck$, dirt is sometimes deposited when the square is clean. With automatic dirt sensing, this is always detected, so we have a disjunctive conditional effect:

$$\begin{aligned}&Action(Suck, \textsc{Precond}:,\\ &\quad \textsc{Effect}:\textbf{when } AtL \wedge \neg CleanL\colon K(CleanL)\\ &\qquad \wedge \textbf{ when } AtL \wedge CleanL\colon K(CleanL) \vee \neg K(CleanL) \wedge\\ &\qquad \textbf{when } AtR \wedge \neg CleanR\colon K(CleanR)\\ &\qquad \wedge \textbf{ when } AtR \wedge CleanR\colon K(CleanR) \vee \neg K(CleanR)\end{aligned}$$

12.19 The continuous planning agent described in Section 12.6 has at least one of the listed abilities, namely the ability to accept new goals as it goes along. A new goal is simply added as an extra open precondition in the $Finish$ step, and the planner will find a way to satisfy it, if possible, along with the other remaining goals. Because the data structures built by the continuous planning agent as it works on the plan remain largely in place as the plan is executed, the cost of replanning is usually relatively small unless the failure is catastrophic. There is no specific time bound that is guaranteed, and in general no such bound is possible because changing even a single state variable might require completely reconstructing the plan from scratch.

12.20 Let T be the proposition that the patient is dehydrated and S be the side effect. We have

$$\begin{aligned}&Action(Drink, \textsc{Precond}:, \textsc{Effect}:\neg T)\\ &Action(Medicate, \textsc{Precond}:, \textsc{Effect}:\neg D \wedge \textbf{when } T\colon S)\end{aligned}$$

and the initial state is $\neg S \wedge (T \vee D) \wedge (\neg T \vee \neg D)$. The solution plan is $[Drink, Medicate]$. There are two possible worlds, one where T holds and one where D holds. In the first, $Drink$ causes $\neg T$ and $Medicate$ has no effect; in the second, $Drink$ has no effect and $Medicate$ causes $\neg D$. In both cases, the final state is $\neg S \wedge \neg T \wedge \neg D$.

12.21 One solution plan is $[Test, \textbf{if}\, CultureGrowth\, \textbf{then}[Drink, Medicate]]$.