Instructor's Manual:
Exercise Solutions
for

Artificial Intelligence

A Modern Approach

Third Edition (International Version)

Stuart J. Russell and Peter Norvig

with contributions from
Ernest Davis, Nicholas J. Hay, and Mehran Sahami

Prentice Hall
Upper Saddle River Boston Columbus San Francisco New York
Indianapolis London Toronto Sydney Singapore Tokyo Montreal
Dubai Madrid Hong Kong Mexico City Munich Paris Amsterdam Cape Town

Editor-in-Chief: Michael Hirsch
Executive Editor: Tracy Dunkelberger
Assistant Editor: Melinda Haggerty
Editorial Assistant: Allison Michael
Vice President, Production: Vince O'Brien
Senior Managing Editor: Scott Disanno
Production Editor: Jane Bonnell
Interior Designers: Stuart Russell and Peter Norvig

Library of Congress Cataloging-in-Publication Data on File

Prentice Hall
is an imprint of

www.pearsonhighered.com

10 9 8 7 6 5 4 3 2 1
ISBN-13: 978-0-13-606738-2
ISBN-10: 0-13-606738-7

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 (Third 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 making your own slides.
- 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.

d. We define **rationality** as the property of a system which does the "right thing" given what it knows. See Section 2.2 for a more complete discussion. Both describe perfect rationality, however; see Section 27.3.

e. We define **logical reasoning** as the a process of deriving new sentences from old, such that the new sentences are necessarily true if the old ones are true. (Notice that does not refer to any specific syntax oor formal language, but it does require a well-defined notion of truth.)

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 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.

1.4 No. IQ test scores correlate well with certain other measures, such as success in college, ability to make good decisions in complex, real-world situations, ability to learn new skills and subjects quickly, and so on, 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. Consider the following analogy: if a human runs the 100m in 10 seconds, we might describe him or her as *very athletic* and expect competent performance in other areas such as walking, jumping, hurdling, and perhaps throwing balls; but we would not desscribe a Boeing 747 as *very athletic* because it can cover 100m in 0.4 seconds, nor would we expect it to be good at hurdling and throwing balls.

Even for humans, IQ tests are controversial because of their theoretical presuppositions about innate ability (distinct from training effects) adn the generalizability of results. 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."

1.5 In order of magnitude figures, the computational power of the computer is 100 times larger.

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

- Although bar code scanning is in a sense computer vision, these are not AI systems. The problem of reading a bar code is an extremely limited and artificial form of visual interpretation, and it has been carefully designed to be as simple as possible, given the hardware.
- In many respects. The problem of determining the relevance of a web page to a query is a problem in natural language understanding, and the techniques are related to those we will discuss in Chapters 22 and 23. Search engines like Ask.com, which group the retrieved pages into categories, use clustering techniques analogous to those we discuss in Chapter 20. Likewise, other functionalities provided by a search engines use intelligent techniques; for instance, the spelling corrector uses a form of data mining based on observing users' corrections of their own spelling errors. On the other hand, the problem of indexing billions of web pages in a way that allows retrieval in seconds is a problem in database design, not in artificial intelligence.
- To a limited extent. Such menus tends to use vocabularies which are very limited – e.g. the digits, "Yes", and "No" — and within the designers' control, which greatly simplifies the problem. On the other hand, the programs must deal with an uncontrolled space of all kinds of voices and accents.

The voice activated directory assistance programs used by telephone companies, which must deal with a large and changing vocabulary are certainly AI programs.

- This is borderline. There is something to be said for viewing these as intelligent agents working in cyberspace. The task is sophisticated, the information available is partial, the techniques are heuristic (not guaranteed optimal), and the state of the world is dynamic. All of these are characteristic of intelligent activities. On the other hand, the task is very far from those normally carried out in human cognition.

1.8 Presumably the brain has evolved so as to carry out this operations on visual images, but the mechanism is only accessible for one particular purpose in this particular cognitive task of image processing. Until about two centuries ago there was no advantage in people (or animals) being able to compute the convolution of a Gaussian for any other purpose.

The really interesting question here is what we mean by saying that the "actual person" can do something. The person can see, but he cannot compute the convolution of a Gaussian; but computing that convolution is *part* of seeing. This is beyond the scope of this solution manual.

1.9 Evolution tends to perpetuate organisms (and combinations and mutations of organisms) that are successful 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 This question is intended to be about the essential nature of the AI problem and what is required to solve it, but could also be interpreted as a sociological question about the current practice of AI research.

A *science* is a field of study that leads to the acquisition of empirical knowledge by the scientific method, which involves falsifiable hypotheses about what is. A pure *engineering* field can be thought of as taking a fixed base of empirical knowledge and using it to solve problems of interest to society. Of course, engineers do bits of science—e.g., they measure the properties of building materials—and scientists do bits of engineering to create new devices and so on.

As described in Section 1.1, the "human" side of AI is clearly an empirical science—called cognitive science these days—because it involves psychological experiments designed out to find out how human cognition actually works. What about the the "rational" side? If we view it as studying the abstract relationship among an arbitrary task environment, a computing device, and the program for that computing device that yields the best performance in the task environment, then the rational side of AI is really mathematics and engineering; it does not require any empirical knowledge about the *actual* world—and the *actual* task environment—that we inhabit; that a given program will do well in a given environment is a *theorem*. (The same is true of pure decision theory.) In practice, however, we are interested in task environments that do approximate the actual world, so even the rational side of AI involves finding out what the actual world is like. For example, in studying rational agents that communicate, we are interested in task environments that contain humans, so we have

to find out what human language is like. In studying perception, we tend to focus on sensors such as cameras that extract useful information from the actual world. (In a world without light, cameras wouldn't be much use.) Moreover, to design vision algorithms that are good at extracting information from camera images, we need to understand the actual world that generates those images. Obtaining the required understanding of scene characteristics, object types, surface markings, and so on is a quite different kind of science from ordinary physics, chemistry, biology, and so on, but it is still science.

In summary, AI is definitely engineering but it would not be especially useful to us if it were not also an empirical science concerned with those aspects of the real world that affect the design of intelligent systems for that world.

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 1.11, 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.

1.14

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. Some lane changes and turns can be made on clearly marked roads in light to moderate traffic. Driving in downtown Cairo is too unpredictable for any of these to work.

c. (driving in Victorville, California) Yes, to some extent, as demonstrated in DARPA's Urban Challenge. Some of the vehicles managed to negotiate streets, intersections, well-behaved traffic, and well-behaved pedestrians in good visual conditions.

d. (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.

e. (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.

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

g. (theorem proving) Yes. For example, the proof of Robbins algebra described on page 360.

h. (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.

i. (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.

j. (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.

k. (surgery) Yes. Robots are increasingly being used for surgery, although always under the command of a doctor. Robotic skills demonstrated at superhuman levels include drilling holes in bone to insert artificial joints, suturing, and knot-tying. They are not yet capable of planning and carrying out a complex operation autonomously from start to finish.

1.15

The progress made in this contests is a matter of fact, but the impact of that progress is a matter of opinion.

- **DARPA Grand Challenge for Robotic Cars** In 2004 the Grand Challenge was a 240 km race through the Mojave Desert. It clearly stressed the state of the art of autonomous driving, and in fact no competitor finished the race. The best team, CMU, completed only 12 of the 240 km. In 2005 the race featured a 212km course with fewer curves and wider roads than the 2004 race. Five teams finished, with Stanford finishing first, edging out two CMU entries. This was hailed as a great achievement for robotics and for the Challenge format. In 2007 the Urban Challenge put cars in a city setting, where they had to obey traffic laws and avoid other cars. This time CMU edged out Stanford.

The competition appears to have been a good testing ground to put theory into practice, something that the failures of 2004 showed was needed. But it is important that the competition was done at just the right time, when there was theoretical work to consolidate, as demonstrated by the earlier work by Dickmanns (whose VaMP car drove autonomously for 158km in 1995) and by Pomerleau (whose Navlab car drove 5000km across the USA, also in 1995, with the steering controlled autonomously for 98% of the trip, although the brakes and accelerator were controlled by a human driver).

- **International Planning Competition** In 1998, five planners competed: Blackbox, HSP, IPP, SGP, and STAN. The result page (`ftp://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html`) stated "all of these planners performed very well, compared to the state of the art a few years ago." Most plans found were 30 or 40 steps, with some over 100 steps. In 2008, the competition had expanded quite a bit: there were more tracks (satisficing vs. optimizing; sequential vs. temporal; static vs. learning). There were about 25 planners, including submissions from the 1998 groups (or their descendants) and new groups. Solutions found were much longer than in 1998. In sum, the field has progressed quite a bit in participation, in breadth, and in power of the planners. In the 1990s it was possible to publish a Planning paper that discussed only a theoretical approach; now it is necessary to show quantitative evidence of the efficacy of an approach. The field is stronger and more mature now, and it seems that the planning competition deserves some of the credit. However, some researchers feel that too much emphasis is placed on the particular classes of problems that appear in the competitions, and not enough on real-world applications.
- **Robocup Robotics Soccer** This competition has proved extremely popular, attracting 407 teams from 43 countries in 2009 (up from 38 teams from 11 countries in 1997). The robotic platform has advanced to a more capable humanoid form, and the strategy and tactics have advanced as well. Although the competition has spurred innovations in distributed control, the winning teams in recent years have relied more on individual ball-handling skills than on advanced teamwork. The competition has served to increase interest and participation in robotics, although it is not clear how well they are advancing towards the goal of defeating a human team by 2050.
- **TREC Information Retrieval Conference** This is one of the oldest competitions, started in 1992. The competitions have served to bring together a community of researchers, have led to a large literature of publications, and have seen progress in participation and in quality of results over the years. In the early years, TREC served its purpose as a place to do evaluations of retrieval algorithms on text collections that were large for the time. However, starting around 2000 TREC became less relevant as the advent of the World Wide Web created a corpus that was available to anyone and was much larger than anything TREC had created, and the development of commercial search engines surpassed academic research.
- **NIST Open Machine Translation Evaluation** This series of evaluations (explicitly not labelled a "competition") has existed since 2001. Since then we have seen great advances in Machine Translation quality as well as in the number of languages covered.

The dominant approach has switched from one based on grammatical rules to one that relies primarily on statistics. The NIST evaluations seem to track these changes well, but don't appear to be driving the changes.

Overall, we see that whatever you measure is bound to increase over time. For most of these competitions, the measurement was a useful one, and the state of the art has progressed. In the case of ICAPS, some planning researchers worry that too much attention has been lavished on the competition itself. In some cases, progress has left the competition behind, as in TREC, where the resources available to commercial search engines outpaced those available to academic researchers. In this case the TREC competition was useful—it helped train many of the people who ended up in commercial search engines—and in no way drew energy away from new ideas.

Solutions for Chapter 2
Intelligent Agents

2.1 This question tests the student's understanding of environments, rational actions, and performance measures. Any sequential environment in which rewards may take time to arrive will work, because then we can arrange for the reward to be "over the horizon." Suppose that in any state there are two action choices, a and b, and consider two cases: the agent is in state s at time T or at time $T-1$. In state s, action a reaches state s' with reward 0, while action b reaches state s again with reward 1; in s' either action gains reward 10. At time $T-1$, it's rational to do a in s, with expected total reward 10 before time is up; but at time T, it's rational to do b with total expected reward 1 because the reward of 10 cannot be obtained before time is up.

Students may also provide common-sense examples from real life: investments whose payoff occurs after the end of life, exams where it doesn't make sense to start the high-value question with too little time left to get the answer, and so on.

The environment state can include a clock, of course; this doesn't change the gist of the answer—now the action will depend on the clock as well as on the non-clock part of the state—but it does mean that the agent can never be in the same state twice.

2.2 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

EXTERNAL MEMORY

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.

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.

2.3

a. *An agent that senses only partial information about the state cannot be perfectly rational.*
False. Perfect rationality refers to the ability to make good decisions given the sensor information received.

b. *There exist task environments in which no pure reflex agent can behave rationally.*
True. A pure reflex agent ignores previous percepts, so cannot obtain an optimal state estimate in a partially observable environment. For example, correspondence chess is played by sending moves; if the other player's move is the current percept, a reflex agent could not keep track of the board state and would have to respond to, say, "a4" in the same way regardless of the position in which it was played.

c. *There exists a task environment in which every agent is rational.*
True. For example, in an environment with a single state, such that all actions have the same reward, it doesn't matter which action is taken. More generally, any environment that is reward-invariant under permutation of the actions will satisfy this property.

d. *The input to an agent program is the same as the input to the agent function.*
False. The agent function, notionally speaking, takes as input the entire percept sequence up to that point, whereas the agent program takes the current percept only.

e. *Every agent function is implementable by some program/machine combination.*
False. For example, the environment may contain Turing machines and input tapes and the agent's job is to solve the halting problem; there is an agent *function* that specifies the right answers, but no agent program can implement it. Another example would be an agent function that requires solving intractable problem instances of arbitrary size in constant time.

f. *Suppose an agent selects its action uniformly at random from the set of possible actions. There exists a deterministic task environment in which this agent is rational.*
True. This is a special case of (c); if it doesn't matter which action you take, selecting randomly is rational.

g. *It is possible for a given agent to be perfectly rational in two distinct task environments.*
True. For example, we can arbitrarily modify the parts of the environment that are unreachable by any optimal policy as long as they stay unreachable.

h. *Every agent is rational in an unobservable environment.*
False. Some actions are stupid—and the agent may know this if it has a model of the environment—even if one cannot perceive the environment state.

i. *A perfectly rational poker-playing agent never loses.*
False. Unless it draws the perfect hand, the agent can always lose if an opponent has better cards. This can happen for game after game. The correct statement is that the agent's expected winnings are nonnegative.

2.4 Many of these can actually be argued either way, depending on the level of detail and abstraction.

A. Partially observable, stochastic, sequential, dynamic, continuous, multi-agent.

B. Partially observable, stochastic, sequential, dynamic, continuous, single agent (unless there are alien life forms that are usefully modeled as agents).

C. Partially observable, deterministic, sequential, static, discrete, single agent. This can be multi-agent and dynamic if we buy books via auction, or dynamic if we purchase on a long enough scale that book offers change.

D. Fully observable, stochastic, episodic (every point is separate), dynamic, continuous, multi-agent.

E. Fully observable, stochastic, episodic, dynamic, continuous, single agent.

F. Fully observable, stochastic, sequential, static, continuous, single agent.

G. Fully observable, deterministic, sequential, static, continuous, single agent.

H. Fully observable, strategic, sequential, static, discrete, multi-agent.

2.5 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 outcome state.
- *Learning agent*: an agent whose behavior improves over time based on its experience.

2.6 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.)

e. It depends on the program and the environment. If the environment is dynamic, speeding up the machine may mean choosing different (perhaps better) actions and/or acting sooner. If the environment is static and the program pays no attention to the passage of elapsed time, the agent function is unchanged.

2.7

The design of goal- and utility-based agents depends on the structure of the task environment. The simplest such agents, for example those in chapters 3 and 10, compute the agent's entire future sequence of actions in advance before acting at all. This strategy works for static and deterministic environments which are either fully-known or unobservable

For fully-observable and fully-known static environments a policy can be computed in advance which gives the action to by taken in any given state.

```
function GOAL-BASED-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              model, a description of how the next state depends on current state and action
              goal, a description of the desired goal state
              plan, a sequence of actions to take, initially empty
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, model)
  if GOAL-ACHIEVED(state,goal) then return a null action
  if plan is empty then
plan ← PLAN(state,goal,model)
  action ← FIRST(plan)
  plan ← REST(plan)
  return action
```

Figure S2.1 A goal-based agent.

For partially-observable environments the agent can compute a conditional plan, which specifies the sequence of actions to take as a function of the agent's perception. In the extreme, a conditional plan gives the agent's response to every contingency, and so it is a representation of the entire agent function.

In all cases it may be either intractable or too expensive to compute everything out in advance. Instead of a conditional plan, it may be better to compute a single sequence of actions which is likely to reach the goal, then monitor the environment to check whether the plan is succeeding, repairing or replanning if it is not. It may be even better to compute only the start of this plan before taking the first action, continuing to plan at later time steps.

Pseudocode for simple goal-based agent is given in Figure S2.1. GOAL-ACHIEVED tests to see whether the current state satisfies the goal or not, doing nothing if it does. PLAN computes a sequence of actions to take to achieve the goal. This might return only a prefix of the full plan, the rest will be computed after the prefix is executed. This agent will act to maintain the goal: if at any point the goal is not satisfied it will (eventually) replan to achieve the goal again.

At this level of abstraction the utility-based agent is not much different than the goal-based agent, except that action may be continuously required (there is not necessarily a point where the utility function is "satisfied"). Pseudocode is given in Figure S2.2.

2.8 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.

2.9 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
```

function UTILITY-BASED-AGENT($percept$) **returns** an action
  persistent: $state$, the agent's current conception of the world state
    $model$, a description of how the next state depends on current state and action
    $utility - function$, a description of the agent's utility function
    $plan$, a sequence of actions to take, initially empty
    $action$, the most recent action, initially none

$state \leftarrow$ UPDATE-STATE($state$, $action$, $percept$, $model$)
if $plan$ is empty **then**
$plan \leftarrow$ PLAN($state$,$utility - function$,$model$)
$action \leftarrow$ FIRST($plan$)
$plan \leftarrow$ REST($plan$)
return $action$

Figure S2.2 A utility-based agent.

```
(cond ((eq status 'Dirty) 'Suck)
      ((eq location 'A) 'Right)
      (t 'Left))))
```

For states 1, 3, 5, 7 in Figure 4.9, the performance measures are 1996, 1999, 1998, 2000 respectively.

2.10

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.

2.11

a. Because the agent does not know the geography and perceives only location and local dirt, and cannot 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))))))
```

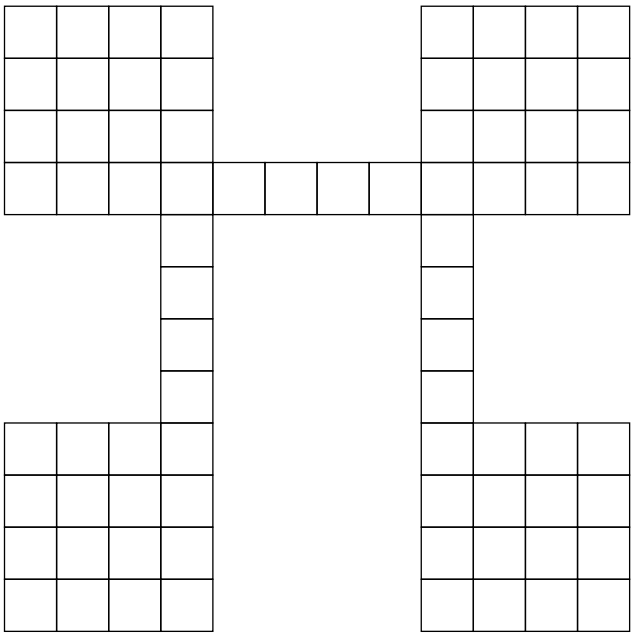

Figure S2.3 An environment in which random motion will take a long time to cover all the squares.

This is fairly close to what the Roomba^{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 probability 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 7.22). For very small Minesweeper environments, optimal exploration is feasible although the belief state

update is nontrivial to explain.

2.12 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.

2.13

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 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.

3.2

a. We'll define the coordinate system so that the center of the maze is at $(0, 0)$, and the maze itself is a square from $(-1, -1)$ to $(1, 1)$.

Initial state: robot at coordinate $(0, 0)$, facing North.
Goal test: either $|x| > 1$ or $|y| > 1$ where (x, y) is the current location.
Successor function: move forwards any distance d; change direction robot it facing.
Cost function: total distance moved.

The state space is infinitely large, since the robot's position is continuous.

b. The state will record the intersection the robot is currently at, along with the direction it's facing. At the end of each corridor leaving the maze we will have an exit node. We'll assume some node corresponds to the center of the maze.

Initial state: at the center of the maze facing North.
Goal test: at an exit node.
Successor function: move to the next intersection in front of us, if there is one; turn to face a new direction.
Cost function: total distance moved.

There are $4n$ states, where n is the number of intersections.

c. Initial state: at the center of the maze.
Goal test: at an exit node.
Successor function: move to next intersection to the North, South, East, or West.
Cost function: total distance moved.

We no longer need to keep track of the robot's orientation since it is irrelevant to

predicting the outcome of our actions, and not part of the goal test. The motor system that executes this plan will need to keep track of the robot's current orientation, to know when to rotate the robot.

d. State abstractions:

(i) Ignoring the height of the robot off the ground, whether it is tilted off the vertical.
(ii) The robot can face in only four directions.
(iii) Other parts of the world ignored: possibility of other robots in the maze, the weather in the Caribbean.

Action abstractions:

(i) We assumed all positions we safely accessible: the robot couldn't get stuck or damaged.
(ii) The robot can move as far as it wants, without having to recharge its batteries.
(iii) Simplified movement system: moving forwards a certain distance, rather than controlled each individual motor and watching the sensors to detect collisions.

3.3

a. State space: States are all possible city pairs (i, j). The map is *not* the state space.

Successor function: The successors of (i, j) are all pairs (x, y) such that $Adjacent(x, i)$ and $Adjacent(y, j)$.

Goal: Be at (i, i) for some i.

Step cost function: The cost to go from (i, j) to (x, y) is $max(d(i, x), d(j, y))$.

b. In the best case, the friends head straight for each other in steps of equal size, reducing their separation by twice the time cost on each step. Hence (iii) is admissible.

c. Yes: e.g., a map with two nodes connected by one link. The two friends will swap places forever. The same will happen on any chain if they start an odd number of steps apart. (One can see this best on the graph that represents the state space, which has two disjoint sets of nodes.) The same even holds for a grid of any size or shape, because every move changes the Manhattan distance between the two friends by 0 or 2.

d. Yes: take any of the unsolvable maps from part (c) and add a self-loop to any one of the nodes. If the friends start an odd number of steps apart, a move in which one of the friends takes the self-loop changes the distance by 1, rendering the problem solvable. If the self-loop is not taken, the argument from (c) applies and no solution is possible.

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 (n choices), the second in any square in column 2 except the same row that as the rook in column 1 ($n-1$ choices), and so on. This gives $n!$ elements of the search space.

For n queens, notice that a queen attacks at most three squares in any given column, so in column 2 there are at least $(n-3)$ choices, in column at least $(n-6)$ choices, and so on. Thus the state space size $S \geq n \cdot (n-3) \cdot (n-6) \cdots$. Hence we have

$$\begin{aligned} S^3 &\geq n \cdot n \cdot n \cdot (n-3) \cdot (n-3) \cdot (n-3) \cdot (n-6) \cdot (n-6) \cdot (n-6) \cdot \cdots \\ &\geq n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot (n-4) \cdot (n-5) \cdot (n-6) \cdot (n-7) \cdot (n-8) \cdot \cdots \\ &= n! \end{aligned}$$

or $S \geq \sqrt[3]{n!}$.

3.6

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.

3.7

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 3.31.

c. Code not shown.

d. Implementations and analysis not shown.

3.8

a. Any path, no matter how bad it appears, might lead to an arbitrarily 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 reward 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 accommodate 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.

3.9

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 second 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 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.

3.11 A world state is how reality is or could be. In one world state we're in Arad, in another we're in Bucharest. The world state also includes which street we're on, what's currently on the radio, and the price of tea in China. A state description is an agent's internal description of a world state. Examples are $In(Arad)$ and $In(Bucharest)$. These descriptions are necessarily approximate, recording only some aspect of the state.

We need to distinguish between world states and state descriptions because state description are lossy abstractions of the world state, because the agent could be mistaken about

how the world is, because the agent might want to imagine things that aren't true but it could make true, and because the agent cares about the world not its internal representation of it.

Search nodes are generated during search, representing a state the search process knows how to reach. They contain additional information aside from the state description, such as the sequence of actions used to reach this state. This distinction is useful because we may generate different search nodes which have the same state, and because search nodes contain more information than a state representation.

3.12 The state space is a tree of depth one, with all states successors of the initial state. There is no distinction between depth-first search and breadth-first search on such a tree. If the sequence length is unbounded the root node will have infinitely many successors, so only algorithms which test for goal nodes as we generate successors can work.

What happens next depends on how the composite actions are sorted. If there is no particular ordering, then a random but systematic search of potential solutions occurs. If they are sorted by dictionary order, then this implements depth-first search. If they are sorted by length first, then dictionary ordering, this implements breadth-first search.

A significant disadvantage of collapsing the search space like this is if we discover that a plan starting with the action "unplug your battery" can't be a solution, there is no easy way to ignore all other composite actions that start with this action. This is a problem in particular for informed search algorithms.

Discarding sequence structure is not a particularly practical approach to search.

3.13

The graph separation property states that "every path from the initial state to an unexplored state has to pass through a state in the frontier."

At the start of the search, the frontier holds the initial state; hence, trivially, every path from the initial state to an unexplored state includes a node in the frontier (the initial state itself).

Now, we assume that the property holds at the beginning of an arbitrary iteration of the GRAPH-SEARCH algorithm in Figure 3.7. We assume that the iteration completes, i.e., the frontier is not empty and the selected leaf node n is not a goal state. At the end of the iteration, n has been removed from the frontier and its successors (if not already explored or in the frontier) placed in the frontier. Consider any path from the initial state to an unexplored state; by the induction hypothesis such a path (at the beginning of the iteration) includes at least one frontier node; except when n is the only such node, the separation property automatically holds. Hence, we focus on paths passing through n (and no other frontier node). By definition, the next node n' along the path from n must be a successor of n that (by the preceding sentence) is already not in the frontier. Furthermore, n' cannot be in the explored set, since by assumption there is a path from n' to an unexplored node not passing through the frontier, which would violate the separation property as every explored node is connected to the initial state by explored nodes (see lemma below for proof this is always possible). Hence, n' is not in the explored set, hence it will be added to the frontier; then the path will include a frontier node and the separation property is restored.

The property is violated by algorithms that move nodes from the frontier into the ex-

plored set before all of their successors have been generated, as well as by those that fail to add some of the successors to the frontier. Note that it is not necessary to generate *all* successors of a node at once before expanding another node, as long as partially expanded nodes remain in the frontier.

Lemma: Every explored node is connected to the initial state by a path of explored nodes.

Proof: This is true initially, since the initial state is connected to itself. Since we never remove nodes from the explored region, we only need to check new nodes we add to the explored list on an expansion. Let n be such a new explored node. This is previously on the frontier, so it is a neighbor of a node n' previously explored (i.e., its parent). n' is, by hypothesis is connected to the initial state by a path of explored nodes. This path with n appended is a path of explored nodes connecting n' to the initial state.

3.14

a. *False*: a lucky DFS might expand exactly d nodes to reach the goal. A* largely dominates any graph-search algorithm that is *guaranteed to find optimal solutions*.

b. *True*: $h(n) = 0$ is always an admissible heuristic, since costs are nonnegative.

c. *True*: A* search is often used in robotics; the space can be discretized or skeletonized.

d. *True*: depth of the solution matters for breadth-first search, not cost.

e. *False*: a rook can move across the board in move one, although the Manhattan distance from start to finish is 8.

3.15

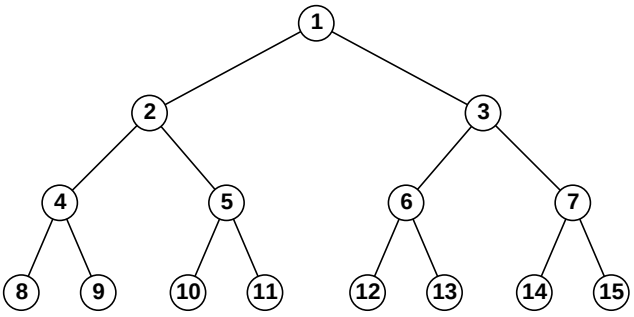

Figure S3.1 The state space for the problem defined in Ex. 3.15.

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. The branching factor is 2 in the forward direction; 1 in the reverse direction.

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

e. The solution can be read off the binary numeral for the goal number. Write the goal number in binary. Since we can only reach positive integers, this binary expansion beings with a 1. From most- to least- significant bit, skipping the initial 1, go Left to the node $2n$ if this bit is 0 and go Right to node $2n + 1$ if it is 1. For example, suppose the goal is 11, which is 1011 in binary. The solution is therefore Left, Right, Right.

3.16

a. **Initial state**: one arbitrarily selected piece (say a straight piece).
Successor function: for any open peg, add any piece type from remaining types. (You can add to open holes as well, but that isn't necessary as all complete tracks can be made by adding to pegs.) For a curved piece, add *in either orientation*; for a fork, add *in either orientation* and (if there are two holes) connecting *at either hole*. It's a good idea to disallow any overlapping configuration, as this terminates hopeless configurations early. (Note: there is no need to consider open holes, because in any solution these will be filled by pieces added to open pegs.)
Goal test: all pieces used in a single connected track, no open pegs or holes, no overlapping tracks.
Step cost: one per piece (actually, doesn't really matter).

b. All solutions are at the same depth, so depth-first search would be appropriate. (One could also use depth-limited search with limit $n - 1$, but strictly speaking it's not necessary to do the work of checking the limit because states at depth $n - 1$ have no successors.) The space is very large, so uniform-cost and breadth-first would fail, and iterative deepening simply does unnecessary extra work. There are many repeated states, so it might be good to use a closed list.

c. A solution has no open pegs or holes, so every peg is in a hole, so there must be equal numbers of pegs and holes. Removing a fork violates this property. There are two other "proofs" that are acceptable: 1) a similar argument to the effect that there must be an even number of "ends"; 2) each fork creates two tracks, and only a fork can rejoin those tracks into one, so if a fork is missing it won't work. The argument using pegs and holes is actually more general, because it also applies to the case of a three-way fork that has one hole and three pegs or one peg and three holes. The "ends" argument fails here, as does the fork/rejoin argument (which is a bit handwavy anyway).

d. The maximum possible number of open pegs is 3 (starts at 1, adding a two-peg fork increases it by one). Pretending each piece is unique, any piece can be added to a peg, giving at most $12 + (2 \cdot 16) + (2 \cdot 2) + (2 \cdot 2 \cdot 2) = 56$ choices per peg. The total depth is 32 (there are 32 pieces), so an upper bound is $168^{32}/(12! \cdot 16! \cdot 2! \cdot 2!)$ where the factorials deal with permutations of identical pieces. One could do a more refined analysis to handle the fact that the branching factor shrinks as we go down the tree, but it is not pretty.

3.17 **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.

3.18 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.

3.19 As an ordinary person (or agent) browsing the web, we can only generate 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.

3.20 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.

3.21

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 $h(n) = 0$.

3.22 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.

3.23 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]

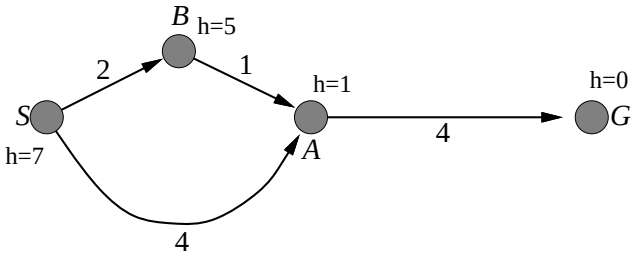

Figure S3.2 A graph with an inconsistent heuristic on which GRAPH-SEARCH fails to return the optimal solution. The successors of S are A with $f = 5$ and B with $f = 7$. A is expanded first, so the path via B will be discarded because A will already be in the closed list.

3.24 See Figure S3.2.

3.25 It is complete whenever $0 \leq w < 2$. $w = 0$ 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.

3.26

a. The branching factor is 4 (number of neighbors of each location).

b. The states at depth k form a square rotated at 45 degrees to the grid. Obviously there are a linear number of states along the boundary of the square, so the answer is $4k$.

c. Without repeated state checking, BFS expends exponentially many nodes: counting precisely, we get $((4^{x+y+1} - 1)/3) - 1$.

d. There are quadratically many states within the square for depth $x + y$, so the answer is $2(x + y)(x + y + 1) - 1$.

e. True; this is the Manhattan distance metric.

f. False; all nodes in the rectangle defined by $(0, 0)$ and (x, y) are candidates for the optimal path, and there are quadratically many of them, all of which may be expended in the worst case.

g. True; removing links may induce detours, which require more steps, so h is an underestimate.

h. False; nonlocal links can reduce the actual path length below the Manhattan distance.

3.27

a. n^{2n}. There are n vehicles in n^2 locations, so roughly (ignoring the one-per-square constraint) $(n^2)^n = n^{2n}$ states.

b. 5^n.

c. Manhattan distance, i.e., $|(n - i + 1) - x_i| + |n - y_i|$. This is exact for a lone vehicle.

d. Only (iii) $\min\{h_1, \ldots, h_n\}$. The explanation is nontrivial as it requires two observations. First, let the *work* W in a given solution be the total *distance* moved by all vehicles over their joint trajectories; that is, for each vehicle, add the lengths of all the steps taken. We have $W \geq \sum_i h_i \geq\geq n \cdot min\{h_1, ..., h_n\}$. Second, the total work we can get done per step is $\leq n$. (Note that for every car that jumps 2, another car has to stay put (move 0), so the total work per step is bounded by n.) Hence, completing all the work requires at least $n \cdot min\{h_1, ..., h_n\}/n = min\{h_1, ..., h_n\}$ steps.

3.28 The heuristic $h = h_1 + h_2$ (adding misplaced tiles and Manhattan distance) sometimes overestimates. Now, suppose $h(n) \leq h^*(n) + c$ (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

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

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

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

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

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 $h(n) \leq c(n, a, n')$. 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

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

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

3.30 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.

3.31 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 cannot 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.

3.32 Students should provide results in the form of graphs and/or tables showing both run-time and number of nodes generated. (Different heuristics have different computation costs.) Runtimes may be very small for 8-puzzles, so you may want to assign the 15-puzzle or 24-puzzle instead. The use of pattern databases is also worth exploring experimentally.

Solutions for Chapter 4
Beyond Classical Search

4.1

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

b. Local beam search with one initial state and no limit on the number of states retained, 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. Simulated annealing with $T = \infty$ at all times is a random-walk search: it always accepts a new state.

e. 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.

4.2 Despite its humble origins, this question raises many of the same issues as the scientifically important problem of protein design. There is a discrete assembly space in which pieces are chosen to be added to the track and a continuous configuration space determined by the "joint angles" at every place where two pieces are linked. Thus we can define a state as a set of oriented, linked pieces and the associated joint angles in the range $[-10, 10]$, plus a set of unlinked pieces. The linkage and joint angles exactly determine the physical layout of the track; we can allow for (and penalize) layouts in which tracks lie on top of one another, or we can disallow them. The evaluation function would include terms for how many pieces are used, how many loose ends there are, and (if allowed) the degree of overlap. We might include a penalty for the amount of deviation from 0-degree joint angles. (We could also include terms for "interestingness" and "traversability"—for example, it is nice to be able to drive a train starting from any track segment to any other, ending up in either direction without having to lift up the train.) The tricky part is the set of allowed moves. Obviously we can unlink any piece or link an unlinked piece to an open peg with either orientation at any allowed angle (possibly excluding moves that create overlap). More problematic are moves to join a peg

and hole on already-linked pieces and moves to change the angle of a joint. Changing one angle may force changes in others, and the changes will vary depending on whether the other pieces are at their joint-angle limit. In general there will be no unique "minimal" solution for a given angle change in terms of the consequent changes to other angles, and some changes may be impossible.

4.3 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.

4.4 Code not shown.

4.5 See Figure S4.1 for the adapted algorithm. For states that OR-SEARCH finds a solution for it records the solution found. If it later visits that state again it immediately returns that solution.

When OR-SEARCH fails to find a solution it has to be careful. Whether a state can be solved depends on the path taken to that solution, as we do not allow cycles. So on failure OR-SEARCH records the value of $path$. If a state is which has previously failed when $path$ contained any subset of its present value, OR-SEARCH returns failure.

To avoid repeating sub-solutions we can label all new solutions found, record these labels, then return the label if these states are visited again. Post-processing can prune off unused labels. Alternatively, we can output a direct acyclic graph structure rather than a tree.

See (Bertoli *et al.*, 2001) for further details.

4.6

The question statement describes the required changes in detail, see Figure S4.2 for the modified algorithm. When OR-SEARCH cycles back to a state on $path$ it returns a token $loop$ which means to loop back to the most recent time this state was reached along the path to it. Since $path$ is implicitly stored in the returned plan, there is sufficient information for later processing, or a modified implementation, to replace these with labels.

The plan representation is implicitly augmented to keep track of whether the plan is cyclic (i.e., contains a $loop$) so that OR-SEARCH can prefer acyclic solutions.

AND-SEARCH returns failure if all branches lead directly to a $loop$, as in this case the plan will always loop forever. This is the only case it needs to check as if all branches in a finite plan loop there must be some And-node whose children all immediately loop.

4.7 A sequence of actions is a solution to a belief state problem if it takes every initial physical state to a goal state. We can relax this problem by requiring it take only *some* initial physical state to a goal state. To make this well defined, we'll require that it finds a solution

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state has previously been solved then return RECALL-SUCCESS(state)
  if state has previously failed for a subset of path then return failure
  if state is on path then
      RECORD-FAILURE(state, path)
      return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then
          RECORD-SUCCESS(state, [action | plan])
          return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure S4.1 AND-OR search with repeated state checking.

for the physical state with the most costly solution. If $h^*(s)$ is the optimal cost of solution starting from the physical state s, then

$$h(S) = \max_{s \in S} h^*(s)$$

is the heuristic estimate given by this relaxed problem. This heuristic assumes any solution to the most difficult state the agent things possible will solve all states.

On the sensorless vacuum cleaner problem in Figure 4.14, h correctly determines the optimal cost for all states except the central three states (those reached by $[suck]$, $[suck, left]$ and $[suck, right]$) and the root, for which h estimates to be 1 unit cheaper than they really are. This means A* will expand these three central nodes, before marching towards the solution.

4.8

a. An action sequence is a solution for belief state b if performing it starting in any state $s \in b$ reaches a goal state. Since any state in a subset of b is in b, the result is immediate.

Any action sequence which is *n*ot a solution for belief state b is also not a solution for any superset; this is the contrapositive of what we've just proved. One cannot, in general, say anything about arbitrary supersets, as the action sequence need not lead to a goal on the states outside of b. One can say, for example, that if an action sequence

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return loop
  cyclic − plan ← None
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then
          if plan is acyclic then return [action | plan]
          cyclic − plan ← [action | plan]
  if cyclic − plan ≠ None then return cyclic − plan
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  loopy ← True
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
      if plan_i ≠ loop then loopy ← False
  if not loopy then
     return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n−1} then plan_{n−1} else plan_n]
  return failure
```

Figure S4.2 AND-OR search with repeated state checking.

solves a belief state b and a belief state b' then it solves the union belief state $b \cup b'$.

b. On expansion of a node, do not add to the frontier any child belief state which is a superset of a previously explored belief state.

c. If you keep a record of previously solved belief states, add a check to the start of OR-search to check whether the belief state passed in is a subset of a previously solved belief state, returning the previous solution in case it is.

4.9

Consider a very simple example: an initial belief state $\{S_1, S_2\}$, actions a and b both leading to goal state G from either initial state, and

$$c(S_1, a, G) = 3\,; \qquad c(S_2, a, G) = 5\,;$$
$$c(S_1, b, G) = 2\,; \qquad c(S_2, b, G) = 6\,.$$

In this case, the solution $[a]$ costs 3 or 5, the solution $[b]$ costs 2 or 6. Neither is "optimal" in any obvious sense.

In some cases, there *will* be an optimal solution. Let us consider just the deterministic case. For this case, we can think of the cost of a plan as a mapping from each initial physical state to the actual cost of executing the plan. In the example above, the cost for $[a]$ is

$\{S_1{:}3, S_2{:}5\}$ and the cost for $[b]$ is $\{S_1{:}2, S_2{:}6\}$. We can say that plan p_1 *weakly dominates* p_2 if, for each initial state, the cost for p_1 is no higher than the cost for p_2. (Moreover, p_1 *dominates* p_2 if it weakly dominates it *and* has a lower cost for some state.) If a plan p weakly dominates all others, it is optimal. Notice that this definition reduces to ordinary optimality in the observable case where every belief state is a singleton. As the preceding example shows, however, a problem may have no optimal solution in this sense. A perhaps acceptable version of A* would be one that returns any solution that is not dominated by another.

PRINCIPLE OF OPTIMALITY

To understand whether it is possible to apply A* at all, it helps to understand its dependence on Bellman's (1957) **principle of optimality**: *An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.* It is important to understand that this is a restriction on performance measures designed to facilitate efficient algorithms, not a general definition of what it means to be optimal.

In particular, if we define the cost of a plan in belief-state space as the minimum cost of any physical realization, we violate Bellman's principle. Modifying and extending the previous example, suppose that a and b reach S_3 from S_1 and S_4 from S_2, and then reach G from there:

$$c(S_1, a, S_3) = 6\,; \qquad c(S_2, a, S_4) = 2\,;$$
$$c(S_1, b, S_3) = 6\,; \qquad c(S_2, b, S_4) = 1\,. c(S_3, a, G) = 2\,; \qquad c(S_4, a, G) = 2\,;$$
$$c(S_3, b, G) = 1\,; \qquad c(S_4, b, G) = 9\,.$$

In the belief state $\{S_3, S_4\}$, the minimum cost of $[a]$ is $\min\{2, 2\} = 2$ and the minimum cost of $[b]$ is $\min\{1, 9\} = 1$, so the optimal plan is $[b]$. In the initial belief state $\{S_1, S_2\}$, the four possible plans have the following costs:

$$[a, a] : \min\{8, 4\} = 4\,; [a, b] : \min\{7, 11\} = 7\,; [b, a] : \min\{8, 3\} = 3\,; [b, b] : \min\{7, 10\} = 7\,.$$

Hence, the optimal plan in $\{S_1, S_2\}$ is $[b, a]$, which does *not* choose b in $\{S_3, S_4\}$ even though that is the optimal plan at that point. This counterintuitive behavior is a direct consequence of choosing the minimum of the possible path costs as the performance measure.

This example gives just a small taste of what might happen with nonadditive performance measures. Details of how to modify and analyze A* for general path-dependent cost functions are give by Dechter and Pearl (1985). Many aspects of A* carry over; for example, we can still derive lower bounds on the cost of a path through a given node. For a belief state b, the minimum value of $g(s) + h(s)$ for each state s in b is a lower bound on the minimum cost of a plan that goes through b.

4.10 The belief state space is shown in Figure S4.3. 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, 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.

4.11

The student needs to make several design choices in answering this question. First, how will the vertices of objects be represented? The problem states the percept is a list of vertex positions, but that is not precise enough. Here is one good choice: The agent has an

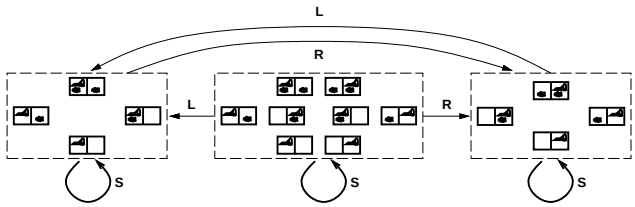

Figure S4.3 The belief state space for the sensorless vacuum world under Murphy's law.

orientation (a heading in degrees). The visible vertexes are listed in clockwise order, starting straight ahead of the agent. Each vertex has a relative angle (0 to 360 degrees) and a distance. We also want to know if a vertex represents the left edge of an obstacle, the right edge, or an interior point. We can use the symbols L, R, or I to indicate this.

The student will need to do some basic computational geometry calculations: intersection of a path and a set of line segments to see if the agent will bump into an obstacle, and visibility calculations to determine the percept. There are efficient algorithms for doing this on a set of line segments, but don't worry about efficiency; an exhaustive algorithm is ok. If this seems too much, the instructor can provide an environment simulator and ask the student only to program the agent.

To answer (c), the student will need some exchange rate for trading off search time with movement time. It is probably too complex to make the simulation asynchronous real-time; easier to impose a penalty in points for computation.

For (d), the agent will need to maintain a set of possible positions. Each time the agent moves, it may be able to eliminate some of the possibilities. The agent may consider moves that serve to reduce uncertainty rather than just get to the goal.

4.12 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. Online search is equivalent to offline search in belief-state space where each action in a belief-state can have multiple successor belief-states: one for each percept the agent could observe after the action. A successor belief-state is constructed by taking the previous belief-state, itself a set of states, replacing each state in this belief-state by the successor state under the action, and removing all successor states which are inconsistent with the percept. This is exactly the construction in Section 4.4.2. AND-OR search can be used to solve this search problem. The initial belief state has $2^1 0 = 1024$ states in it, as we know whether two edges have walls or not (the upper and right edges have no walls) but nothing more. There are $2^{2^{12}}$ possible belief states, one for each set of environment configurations.

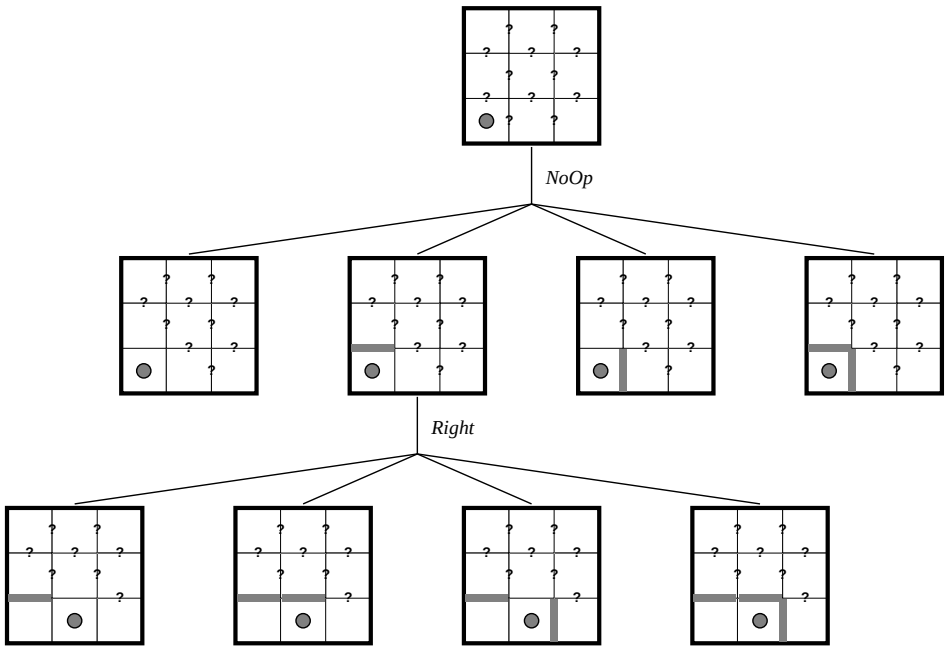

Figure S4.4 The 3×3 maze exploration problem: the initial state, first percept, and one selected action with its perceptual outcomes.

We can view this as a contingency problem in belief state space. 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 represented 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 S4.4. 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} reachable belief states, 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).

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

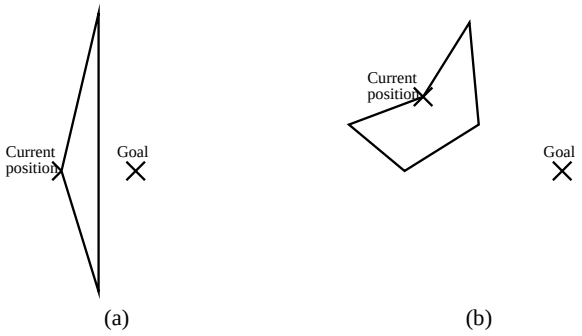

Figure S4.5 (a) Getting stuck with a convex obstacle. (b) Getting stuck with a nonconvex obstacle.

a. It is possible (see Figure S4.5(a)) but very unlikely—the obstacle has to have an unusual shape and be positioned correctly with respect to the goal.

b. With nonconvex obstacles, getting stuck is much more likely to be a problem (see Figure S4.5(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.

e. LRTA* always makes a move, but may move back if the old state looks better than the new state. But then the old state is penalized for the cost of the trip, so eventually the local minimum fills up and the agent escapes.

4.14

Since we can observe successor states, we always know how to backtrack from to a previous state. This means we can adapt iterative deepening search to solve this problem. The only difference is backtracking must be explicit, following the action which the agent can see leads to the previous state.

The algorithm expands the following nodes:

Depth 1: $(0,0)$, $(1,0)$, $(0,0)$, $(-1,0)$, $(0,0)$

Depth 2: $(0,1)$, $(0,0)$, $(0,-1)$, $(0,0)$, $(1,0)$, $(2,0)$, $(1,0)$, $(0,0)$, $(1,0)$, $(1,1)$, $(1,0)$, $(1,-1)$

Solutions for Chapter 5
Adversarial Search

5.1 The translation uses the model of the opponent $OM(s)$ to fill in the opponent's actions, leaving our actions to be determined by the search algorithm. Let $P(s)$ be the state predicted to occur after the opponent has made all their moves according to OM. Note that the opponent may take multiple moves in a row before we get a move, so we need to define this recursively. We have $P(s) = s$ if PLAYERs is us or TERMINAL-TESTs is true, otherwise $P(s) = P(\text{RESULT}(s, OM(s))$.

The search problem is then given by:

a. Initial state: $P(S_0)$ where S_0 is the initial game state. We apply P as the opponent may play first

b. Actions: defined as in the game by ACTIONSs.

c. Successor function: $\text{RESULT}'(s, a) = P(\text{RESULT}(s, a))$

d. Goal test: goals are terminal states

e. Step cost: the cost of an action is zero unless the resulting state s' is terminal, in which case its cost is $M - \text{UTILITY}(s')$ where $M = \max_s \text{UTILITY}(s)$. Notice that all costs are non-negative.

Notice that the state space of the search problem consists of game state where we are to play and terminal states. States where the opponent is to play have been compiled out. One might alternatively leave those states in but just have a single possible action.

Any of the search algorithms of Chapter 3 can be applied. For example, depth-first search can be used to solve this problem, if all games eventually end. This is equivalent to using the minimax algorithm on the original game if $OM(s)$ always returns the minimax move in s.

5.2

a. Initial state: two arbitrary 8-puzzle states. Successor function: one move on an unsolved puzzle. (You could also have actions that change both puzzles at the same time; this is OK but technically you have to say what happens when one is solved but not the other.) Goal test: both puzzles in goal state. Path cost: 1 per move.

b. Each puzzle has $9!/2$ reachable states (remember that half the states are unreachable). The joint state space has $(9!)^2/4$ states.

c. This is like backgammon; expectiminimax works.

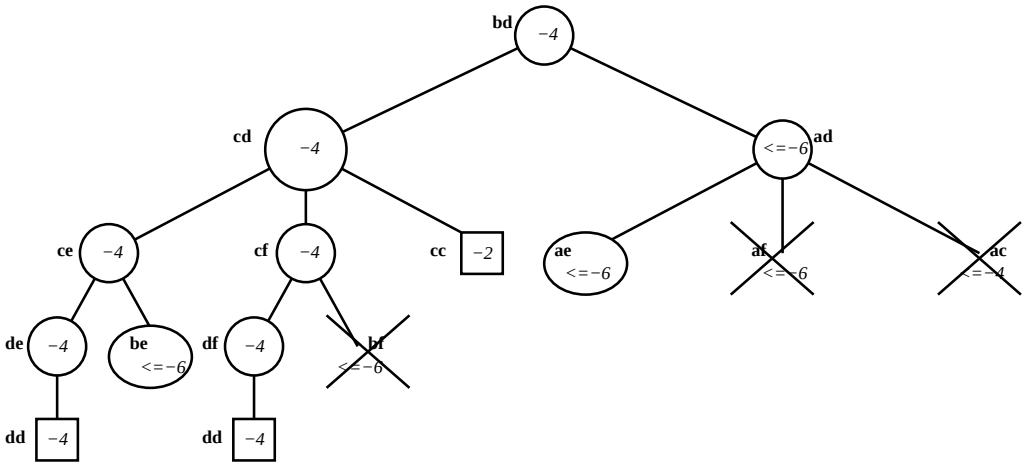

Figure S5.1 Pursuit-evasion solution tree.

d. Actually the statement in the question is not true (it applies to a previous version of part (c) in which the opponent is just trying to prevent you from winning—in that case, the coin tosses will eventually allow you to solve one puzzle without interruptions). For the game described in (c), consider a state in which the coin has come up heads, say, and you get to work on a puzzle that is 2 steps from the goal. Should you move one step closer? If you do, your opponent wins if he tosses heads; or if he tosses tails, you toss tails, and he tosses heads; or any sequence where both toss tails n times and then he tosses heads. So his probability of winning is *at least* $1/2+1/8+1/32+\cdots = 2/3$. So it seems you're better off moving *away* from the goal. (There's no way to stay the same distance from the goal.) This problem unintentionally seems to have the same kind of solution as suicide tictactoe with passing.

5.3

a. See Figure S5.1; the values are just (minus) the number of steps along the path from the root.

b. See Figure S5.1; note that there is both an upper bound and a lower bound for the left child of the root.

c. See figure.

d. The shortest-path length between the two players is a lower bound on the total capture time (here the players take turns, so no need to divide by two), so the "?" leaves have a capture time greater than or equal to the sum of the cost from the root and the shortest-path length. Notice that this bound is derived when the Evader plays very badly. The true value of a node comes from best play by both players, so we can get better bounds by assuming better play. For example, we can get a better bound from the cost when the Evader simply moves backwards and forwards rather than moving towards the Pursuer.

e. See figure (we have used the simple bounds). Notice that once the right child is known

to have a value below –6, the remaining successors need not be considered.

f. The pursuer always wins if the tree is finite. To prove this, let the tree be rooted as the pursuer's current node. (I.e., pick up the tree by that node and dangle all the other branches down.) The evader must either be at the root, in which case the pursuer has won, or in some subtree. The pursuer takes the branch leading to that subtree. This process repeats at most d times, where d is the maximum depth of the original subtree, until the pursuer either catches the evader or reaches a leaf node. Since the leaf has no subtrees, the evader must be at that node.

5.4 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.

5.5 Code not shown.

5.6 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.

5.7 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 S5.2.

5.8

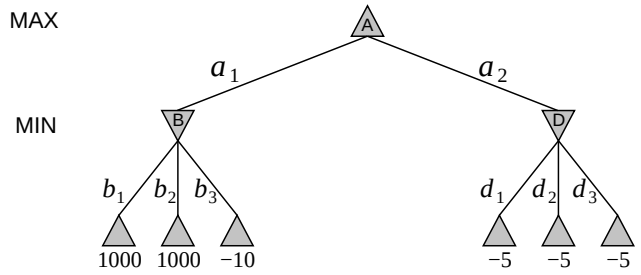

Figure S5.2 A simple game tree showing that setting a trap for MIN by playing a_i is a win if MIN falls for it, but may also be disastrous. The minimax move is of course a_2, with value -5.

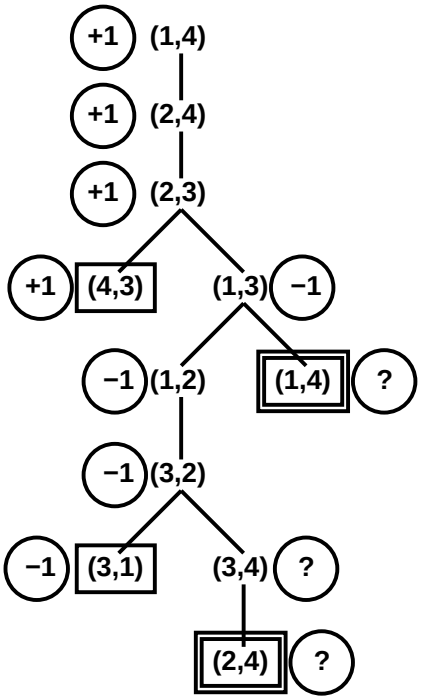

Figure S5.3 The game tree for the four-square game in Exercise 5.8. Terminal states are in single boxes, loop states in double boxes. Each state is annotated with its minimax value in a circle.

a. (5) The game tree, complete with annotations of all minimax values, is shown in Figure S5.3.

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 "?".

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 164. 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 17.7 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 example, 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.

5.9 For **a**, there are at most 9! games. (This is the number of move sequences that fill up the board, but many wins and losses end before the board is full.) For **b–e**, Figure S5.4 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.

5.10

a. An upper bound on the number of terminal nodes is $N!$, one for each ordering of squares, so an upper bound on the total number of nodes is $\sum_{i=1}^{N} i!$. This is not much

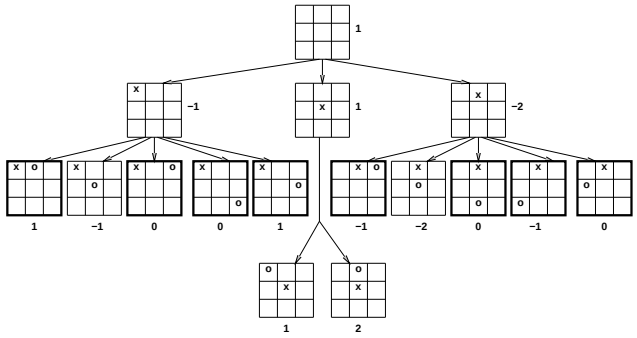

Figure S5.4 Part of the game tree for tic-tac-toe, for Exercise 5.9.

bigger than $N!$ itself as the factorial function grows superexponentially. This is an overestimate because some games will end early when a winning position is filled.

This count doesn't take into account transpositions. An upper bound on the number of distinct game states is 3^N, as each square is either empty or filled by one of the two players. Note that we can determine who is to play just from looking at the board.

b. In this case no games terminate early, and there are $N!$ different games ending in a draw. So ignoring repeated states, we have exactly $\sum_{i=1}^{N} i!$ nodes.

At the end of the game the squares are divided between the two players: $\lceil N/2 \rceil$ to the first player and $\lfloor N/2 \rfloor$ to the second. Thus, a good lower bound on the number of distinct states is $\binom{N}{\lceil N/2 \rceil}$, the number of distinct terminal states.

c. For a state s, let $X(s)$ be the number of winning positions containing no O's and $O(s)$ the number of winning positions containing no X's. One evaluation function is then $Eval(s) = X(s) - O(S)$. Notice that empty winning positions cancel out in the evaluation function.

Alternatively, we might weight potential winning positions by how close they are to completion.

d. Using the upper bound of $N!$ from (a), and observing that it takes $100NN!$ instructions. At 2GHz we have 2 billion instructions per second (roughly speaking), so solve for the largest N using at most this many instructions. For one second we get $N = 9$, for one minute $N = 11$, and for one hour $N = 12$.

5.11 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.

5.12 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 5.2.2 (p.165) 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.

5.13 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 $n_{i1} \ldots n_{i_{b_i}}$ 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 $\max(l_3, l_5, \ldots, l_k)$.

5.14 The result is given in Section 6 of Knuth (1975). The exact statement (Corollary 1 of Theorem 1) is that the algorithms examines $b^{\lfloor m/2 \rfloor} + b^{\lceil m/2 \rceil} - 1$ nodes at level m. These are exactly the nodes reached when Min plays only optimal moves and/or Max plays only optimal moves. The proof is by induction on m.

5.15 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 80 million positions in the table (ignoring pointers for hash table bucket lists). This is about 1/22 of the 1800 million positions generated during a three-minute search.

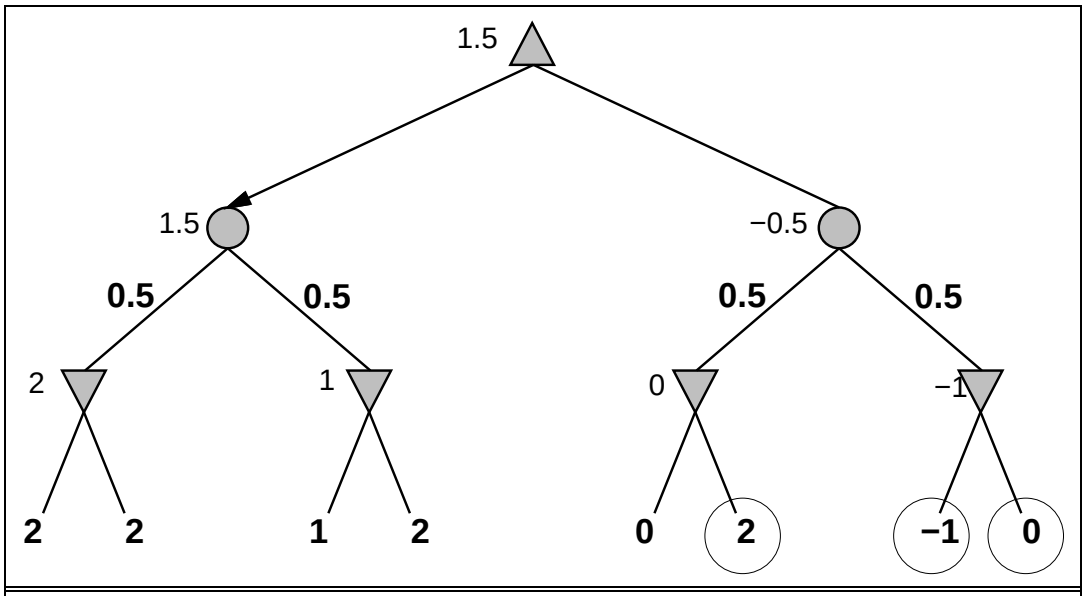

Figure S5.5 Pruning with chance nodes solution.

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.

5.16

a. See Figure S5.5.

b. Given nodes 1–6, we would need to look at 7 and 8: if they were both $+\infty$ then the values of the min node and chance node above would also be $+\infty$ and the best move would change. Given nodes 1–7, we do not need to look at 8. Even if it is $+\infty$, the min node cannot be worth more than -1, so the chance node above cannot be worth more than -0.5, so the best move won't change.

c. The worst case is if either of the third and fourth leaves is -2, in which case the chance node above is 0. The best case is where they are both 2, then the chance node has value 2. So it must lie between 0 and 2.

d. See figure.

5.18 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.

5.19 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 in Section 5.6.2 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.

5.20

a. No pruning. In a max tree, the value of the root is the value of the best leaf. Any unseen leaf might be the best, so we have to see them all.

b. No pruning. An unseen leaf might have a value arbitrarily higher or lower than any other leaf, which (assuming non-zero outcome probabilities) means that there is no bound on the value of any incompletely expanded chance or max node.

c. No pruning. Same argument as in (a).

d. No pruning. Nonnegative values allow *lower* bounds on the values of chance nodes, but a lower bound does not allow any pruning.

e. Yes. If the first successor has value 1, the root has value 1 and all remaining successors can be pruned.

f. Yes. Suppose the first action at the root has value 0.6, and the first outcome of the second action has probability 0.5 and value 0; then all other outcomes of the second action can be pruned.

g. (ii) Highest probability first. This gives the strongest bound on the value of the node, all other things being equal.

5.21

a. *In a fully observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what strategy the second player is using—that is, what move the second player will make, given the first player's move.*
True. The second player will play optimally, and so is perfectly predictable up to ties. Knowing which of two equally good moves the opponent will make does not change the value of the game to the first player.

b. *In a partially observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what move the second player will make, given the first player's move.*
False. In a partially observable game, knowing the second player's move tells the first player additional information about the game state that would otherwise be available only to the second player. For example, in Kriegspiel, knowing the opponent's future move tells the first player where one of the opponent's pieces is; in a card game, it tells the first player one of the opponent's cards.

c. *A perfectly rational backgammon agent never loses.*
False. Backgammon is a game of chance, and the opponent may consistently roll much better dice. The correct statement is that the *expected* winnings are optimal. It is suspected, but not known, that when playing first the expected winnings are positive even against an optimal opponent.

5.22 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.

Solutions for Chapter 6
Constraint Satisfaction Problems

6.1 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.

6.2

a. Solution A: There is a variable corresponding to each of the n^2 positions on the board.
Solution B: There is a variable corresponding to each knight.

b. Solution A: Each variable can take one of two values, {occupied,vacant}
Solution B: Each variable's domain is the set of squares.

c. Solution A: every pair of squares separated by a knight's move is constrained, such that both cannot be occupied. Furthermore, the entire set of squares is constrained, such that the total number of occupied squares should be k.
Solution B: every pair of knights is constrained, such that no two knights can be on the same square or on squares separated by a knight's move. Solution B may be preferable because there is no global constraint, although Solution A has the smaller state space when k is large.

d. Any solution must describe a *complete-state* formulation because we are using a local search algorithm. For simulated annealing, the successor function must completely connect the space; for random-restart, the goal state must be reachable by hillclimbing from some initial state. Two basic classes of solutions are:
Solution C: ensure no attacks at any time. Actions are to remove any knight, add a knight in any unattacked square, or move a knight to any unattacked square.
Solution D: allow attacks but try to get rid of them. Actions are to remove any knight, add a knight in any square, or move a knight to any square.

6.3 **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