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

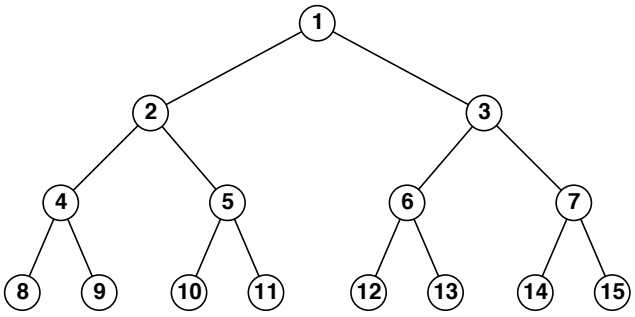

Figure S3.1 The state space for the problem defined in Ex. 3.15.

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, $g(n) \propto \mathit{depth}(n)$, so uniform-cost search reproduces breadth-first search.

b. Breadth-first search is best-first search with $f(n) = \mathit{depth}(n)$; depth-first search is best-first search with $f(n) = -\mathit{depth}(n)$; 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]

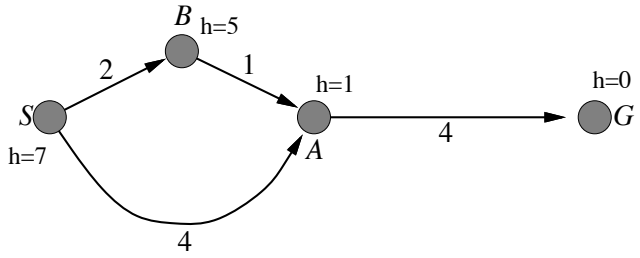

Figure S3.2 A graph with an inconsistent heuristic on which GRAPH-SEARCH fails to return the optimal solution. The successors of S are A with $f = 5$ and B with $f = 7$. A is expanded first, so the path via B will be discarded because A will already be in the closed list.

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 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.

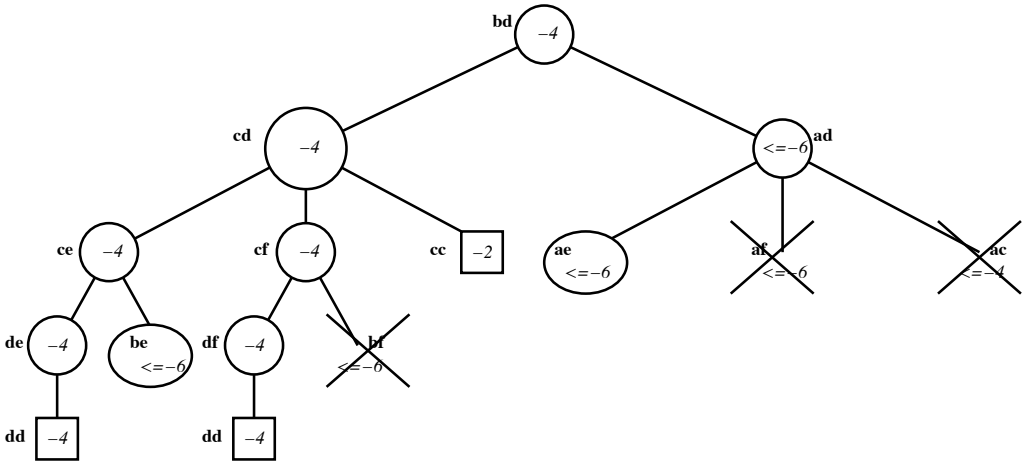

Figure S5.1 Pursuit-evasion solution tree.

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

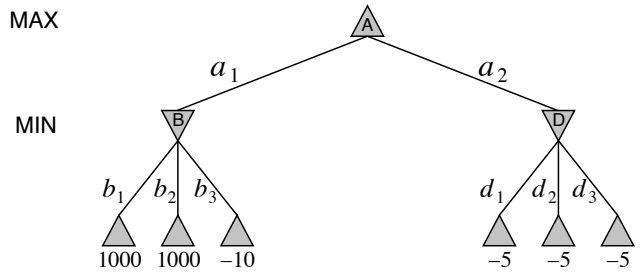

Figure S5.2 A simple game tree showing that setting a trap for MIN by playing a_i is a win if MIN falls for it, but may also be disastrous. The minimax move is of course a_2, with value -5.

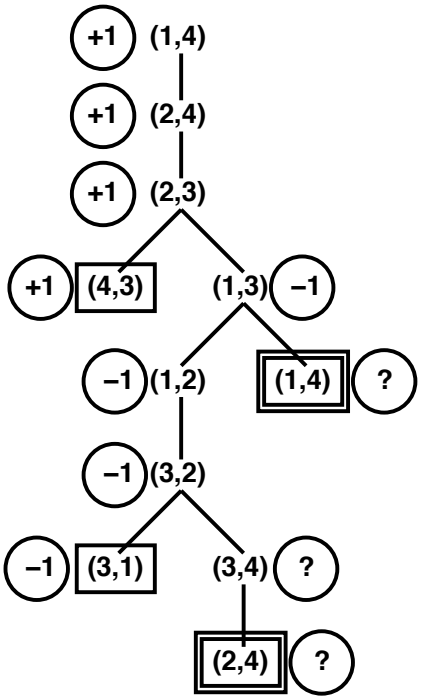

Figure S5.3 The game tree for the four-square game in Exercise 5.8. Terminal states are in single boxes, loop states in double boxes. Each state is annotated with its minimax value in a circle.

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

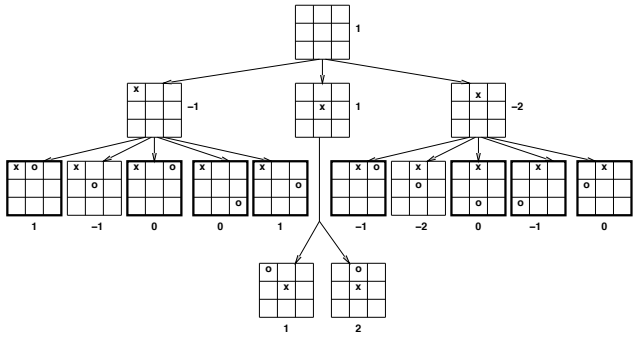

Figure S5.4 Part of the game tree for tic-tac-toe, for Exercise 5.9.

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 $n_{i1} \ldots n_{ib_i}$ are the siblings of node i.

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

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

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

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

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

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

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

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

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.

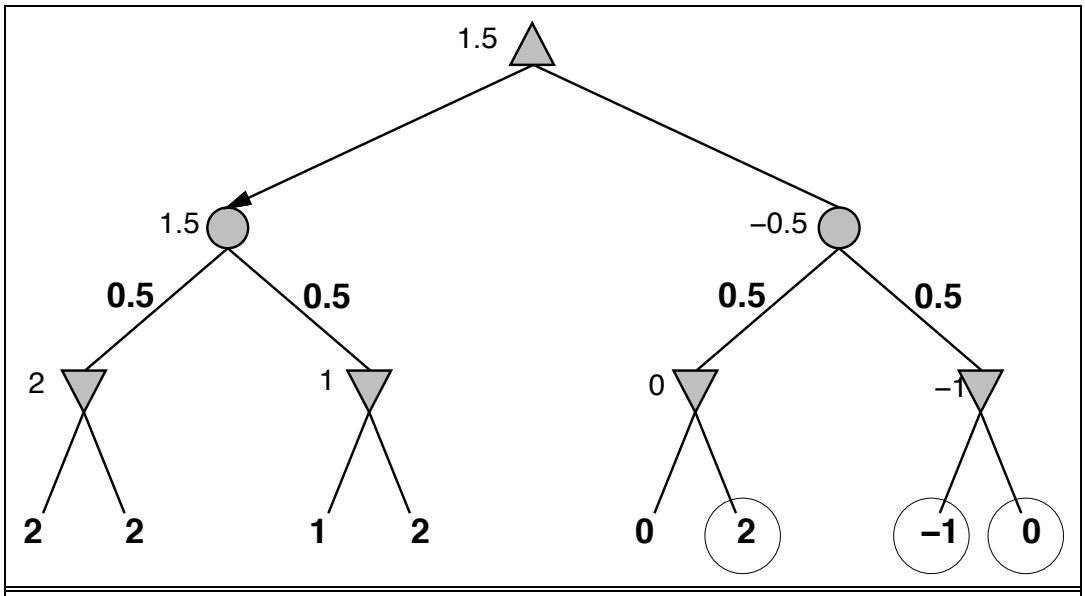

Figure S5.5 Pruning with chance nodes solution.

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 7
Logical Agents

7.1 To save space, we'll show the list of models as a table (Figure S7.1) rather than a collection of diagrams. There are eight possible combinations of pits in the three squares, and four possibilities for the wumpus location (including nowhere).

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

7.2 As human reasoners, we can see from the first two statements, that if it is mythical, then it is immortal; otherwise it is a mammal. So it must be either immortal or a mammal, and thus horned. That means it is also magical. However, we can't deduce anything about whether it is mythical. To proide a formal answer, we can enumerate the possible worlds ($2^5 = 32$ of them with 5 proposition symbols), mark those in which all the assertions are true, and see which conclusions hold in all of those. Or, we can let the machine do the work—in this case, the Lisp code for propositional reasoning:

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

7.3

a. See Figure S7.2. We assume the language has built-in Boolean operators **not**, **and**, **or**, **iff**.

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

Figure S7.1 A truth table constructed for Ex. 7.2. Propositions not listed as true on a given line are assumed false, and only $true$ entries are shown in the table.

b. The question is somewhat ambiguous: we can interpret "in a *partial* model" to mean in *all* such models or *some* such models. For the former interpretation, the sentences $False \wedge P$, $True \vee \neg P$, and $P \wedge \neg P$ can all be determined to be true or false in any partial model. For the latter interpretation, we can in addition have sentences such as $A \wedge P$ which is false in the partial model $\{A = false\}$.

c. A general algorithm for partial models must handle the empty partial model, with no assignments. In that case, the algorithm must determine validity and unsatisfiability,

```
function PL-TRUE?(s, m) returns true or false
  if s = True then return true
  else if s = False then return false
  else if SYMBOL?(s) then return LOOKUP(s, m)
  else branch on the operator of s
      ¬: return not PL-TRUE?(ARG1(s), m)
      ∨: return PL-TRUE?(ARG1(s), m) or PL-TRUE?(ARG2(s), m)
      ∧: return PL-TRUE?(ARG1(s), m) and PL-TRUE?(ARG2(s), m)
      ⇒: (not PL-TRUE?(ARG1(s), m)) or PL-TRUE?(ARG2(s), m)
      ⇔: PL-TRUE?(ARG1(s), m) iff PL-TRUE?(ARG2(s), m)
```

Figure S7.2 Pseudocode for evaluating the truth of a sentence wrt a model.

which are co-NP-complete and NP-complete respectively.

d. It helps if **and** and **or** evaluate their arguments in sequence, terminating on false or true arguments, respectively. In that case, the algorithm already has the desired properties: in the partial model where P is true and Q is unknown, $P \vee Q$ returns true, and $\neg P \wedge Q$ returns false. But the truth values of $Q \vee \neg Q$, $Q \vee True$, and $Q \wedge \neg Q$ are not detected.

e. Early termination in Boolean operators will provide a very substantial speedup. In most languages, the Boolean operators already have the desired property, so you would have to write special "dumb" versions and observe a slow-down.

7.4 In all cases, the question can be resolved easily by referring to the definition of entailment.

a. $False \models True$ is true because $False$ has no models and hence entails every sentence AND because $True$ is true in all models and hence is entailed by every sentence.

b. $True \models False$ is false.

c. $(A \wedge B) \models (A \Leftrightarrow B)$ is true because the left-hand side has exactly one model that is one of the two models of the right-hand side.

d. $A \Leftrightarrow B \models A \vee B$ is false because one of the models of $A \Leftrightarrow B$ has both A and B false, which does not satisfy $A \vee B$.

e. $A \Leftrightarrow B \models \neg A \vee B$ is true because the RHS is $A \Rightarrow B$, one of the conjuncts in the definition of $A \Leftrightarrow B$.

f. $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$ is true because the RHS is false only when both disjuncts are false, i.e., when A and B are true and C is false, in which case the LHS is also false. This may seem counterintuitive, and would not hold if $\Rightarrow$ is interpreted as "causes."

g. $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$ is true; proof by truth table enumeration, or by application of distributivity (Fig 7.11).

h. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B)$ is true; removing a conjunct only allows more models.

i. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$ is false; removing a disjunct allows fewer models.

j. $(A \vee B) \wedge \neg(A \Rightarrow B)$ *is* satisfiable; model has A and $\neg B$.

k. $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ *is* satisfiable; RHS is entailed by LHS so models are those of $A \Leftrightarrow B$.

l. $(A \Leftrightarrow B) \Leftrightarrow C$ does have the same number of models as $(A \Leftrightarrow B)$; half the models of $(A \Leftrightarrow B)$ satisfy $(A \Leftrightarrow B) \Leftrightarrow C$, as do half the non-models, and there are the same numbers of models and non-models.

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

a. α is valid if and only if $True \models \alpha$.
Forward: If $alpha$ is *valid* it is true in all models, hence it is true in all models of $True$.
Backward: if $True \models \alpha$ then α must be true in all models of $True$, i.e., in all models, hence α must be valid.

b. For any α, $False \models \alpha$.
$False$ doesn't hold in any model, so α trivially holds in every model of $False$.

c. $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.
Both sides are equivalent to the assertion that there is no model in which α is true and β is false, i.e., no model in which $\alpha \Rightarrow \beta$ is false.

d. $\alpha \equiv \beta$ if and only if the sentence $(\alpha \Leftrightarrow \beta)$ is valid.
Both sides are equivalent to the assertion that α and β have the same truth value in every model.

e. $\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.
As in **c**, both sides are equivalent to the assertion that there is no model in which α is true and β is false.

7.6

a. If $\alpha \models \gamma$ or $\beta \models \gamma$ (or both) then $(\alpha \wedge \beta) \models \gamma$.
True. This follows from monotonicity.

b. If $\alpha \models (\beta \wedge \gamma)$ then $\alpha \models \beta$ and $\alpha \models \gamma$.
True. If $\beta \wedge \gamma$ is true in every model of α, then β and γ are true in every model of α, so $\alpha \models \beta$ and $\alpha \models \gamma$.

c. If $\alpha \models (\beta \vee \gamma)$ then $\alpha \models \beta$ or $\alpha \models \gamma$ (or both).
False. Consider $\beta \equiv A, \gamma \equiv \neg A$.

7.7 These can be computed by counting the rows in a truth table that come out true, but each has some simple property that allows a short-cut:

a. Sentence is false only if B and C are false, which occurs in 4 cases for A and D, leaving 12.

b. Sentence is false only if A, B, C, and D are false, which occurs in 1 case, leaving 15.

c. The last four conjuncts specify a model in which the first conjunct is false, so 0.

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

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

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

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

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

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

```
------------------------------------------------------------------------
 F  F  F    F         F           F                    T
 T  F  F    F         T           T                    T
 F  T  F    T         T           T                    T
 T  T  F    T         T           T                    T
 F  F  T    T         T           T                    T
 T  F  T    T         T           T                    T
 F  T  T    T         T           T                    T
 T  T  T    T         T           T                    T
------------------------------------------------------------------------
NIL
```

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

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

7.10

a. Valid.

b. Neither.

c. Neither.

d. Valid.

e. Valid.

f. Valid.

g. Valid.

7.11 Each possible world can be written as a conjunction of literals, e.g. $(A \wedge B \wedge \neg C)$. Asserting that a possible world is not the case can be written by negating that, e.g. $\neg(A \wedge B \wedge \neg C)$, which can be rewritten as $(\neg A \vee \neg B \vee C)$. This is the form of a clause; a conjunction of these clauses is a CNF sentence, and can list the negations of all the possible worlds that would make the sentence false.

7.12 To prove the conjunction, it suffices to prove each literal separately. To prove $\neg B$, add the negated goal S7: B.

- Resolve S7 with S5, giving S8: F.
- Resolve S7 with S6, giving S9: C.
- Resolve S8 with S3, giving S10: $(\neg C \vee \neg B)$.
- Resolve S9 with S10, giving S11: $\neg B$.
- Resolve S7 with S11 giving the empty clause.

To prove $\neg A$, add the negated goal S7: A.

- Resolve S7 with the first clause of S1, giving S8: $(B \vee E)$.
- Resolve S8 with S4, giving S9: B.
- Proceed as above to derive the empty clause.

7.13

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

b. A clause can have positive and negative literals; let the negative literals have the form $\neg P_1, \ldots, \neg P_m$ and let the positive literals have the form $Q_1, \ldots, Q_n$, where the P_is and Q_js are symbols. Then the clause can be written as $(\neg P_1 \vee \cdots \vee \neg P_m \vee Q_1 \vee \cdots \vee Q_n)$. By the previous argument, with $Q = Q_1 \vee \cdots \vee Q_n$, it is immediate that the clause is equivalent to

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

c. For atoms p_i, q_i, r_i, s_i where $p_j = q_k$:

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

7.14

a. Correct representations of "a person who is radical is electable if he/she is conservative, but otherwise is not electable":

(i) $(R \wedge E) \iff C$
No; this sentence asserts, among other things, that all conservatives are radical, which is not what was stated.

(ii) $R \Rightarrow (E \iff C)$
Yes, this says that if a person is a radical then they are electable if and only if they are conservative.

(iii) $R \Rightarrow ((C \Rightarrow E) \vee \neg E)$
No, this is equivalent to $\neg R \vee \neg C \vee E \vee \neg E$ which is a tautology, true under any assignment.

b. Horn form:

(i) Yes:

$$\begin{aligned}(R \wedge E) \iff C &\equiv ((R \wedge E) \Rightarrow C) \wedge (C \Rightarrow (R \wedge E)) \\ &\equiv ((R \wedge E) \Rightarrow C) \wedge (C \Rightarrow R) \wedge (C \Rightarrow E)\end{aligned}$$

(ii) Yes:

$$\begin{aligned}R \Rightarrow (E \iff C) &\equiv R \Rightarrow ((E \Rightarrow C) \wedge (C \Rightarrow E)) \\ &\equiv \neg R \vee ((\neg E \vee C) \wedge (\neg C \vee E)) \\ &\equiv (\neg R \vee \neg E \vee C) \wedge (\neg R \vee \neg C \vee E)\end{aligned}$$

(iii) Yes, e.g., $True \Rightarrow True$.

7.15

a. The graph is simply a connected chain of 5 nodes, one per variable.

b. $n + 1$ solutions. Once any X_i is true, all subsequent X_js must be true. Hence the solutions are i falses followed by $n - i$ trues, for $i = 0, \ldots, n$.

c. The complexity is $O(n^2)$. This is somewhat tricky. Consider what part of the complete binary tree is explored by the search. The algorithm must follow all solution sequences, which themselves cover a quadratic-sized portion of the tree. Failing branches are all those trying a $false$ after the preceding variable is assigned $true$. Such conflicts are detected immediately, so they do not change the quadratic cost.

d. These facts are not obviously connected. Horn-form logical inference problems need not have tree-structured constraint graphs; the linear complexity comes from the nature of the constraint (implication) not the structure of the problem.

7.16 A clause is a disjunction of literals, and its models are the *union* of the sets of models of each literal; and each literal satisfies half the possible models. (Note that $False$ is unsatisfiable, but it is really another name for the empty clause.) A 3-SAT clause with three distinct variables rules out exactly 1/8 of all possible models, so five clauses can rule out no more than 5/8 of the models. Eight clauses are needed to rule out all models. Suppose we have variables A, B, C. There are eight models, and we write one clause to rule out each model. For example, the model $\{A = false, B = false, C = false\}$ is ruled out by the clause $(\neg A \vee \neg B \vee \neg C)$.

7.17

a. The negated goal is $\neg G$. Resolve witht he last two clauses to produce $\neg C$ and $\neg D$. Resolve with the second and third clauses to produce $\neg A$ and $\neg B$. Resolve these successively against the first clause to produce the empty clause.

b. This can be answered with or without $True$ and $False$ symbols; we'll omit them for simplicity. First, each 2-CNF clause has two places to put literals. There are $2n$ distinct literals, so there are $(2n)^2$ syntactically distinct clauses. Now, many of these clauses are semantically identical. Let us handle them in groups. There are $C(2n, 2) = (2n)(2n - 1)/2 = 2n^2 - n$ clauses with two different literals, if we ignore ordering. All these

clauses are semantically distinct except those that are equivalent to $True$ (e.g., $(A \vee \neg A)$), of which there are n, so that makes $2n^2 - 2n + 1$ clauses with distinct literals. There are $2n$ clauses with repeated literals, all distinct. So there are $2n^2 + 1$ distinct clauses in all.

c. Resolving two 2-CNF clauses cannot increase the clause size; therefore, resolution can generate only $O(n^2)$ distinct clauses before it must terminate.

d. First, note that the number of 3-CNF clauses is $O(n^3)$, so we cannot argue for nonpolynomial complexity on the basis of the number of different clauses! The key observation is that resolving two 3-CNF clauses can *increase* the clause size to 4, and so on, so clause size can grow to $O(n)$, giving $O(2^n)$ possible clauses.

7.18

a. A simple truth table has eight rows, and shows that the sentence is true for all models and hence valid.

b. For the left-hand side we have:

$$(Food \Rightarrow Party) \vee (Drinks \Rightarrow Party)$$
$$(\neg Food \vee Party) \vee (\neg Drinks \vee Party)$$
$$(\neg Food \vee Party \vee \neg Drinks \vee Party)$$
$$(\neg Food \vee \neg Drinks \vee Party)$$

and for the right-hand side we have

$$(Food \wedge Drinks) \Rightarrow Party$$
$$\neg(Food \wedge Drinks) \vee Party$$
$$(\neg Food \vee \neg Drinks) \vee Party$$
$$(\neg Food \vee \neg Drinks \vee Party)$$

The two sides are identical in CNF, and hence the original sentence is of the form $P \Rightarrow P$, which is valid for any P.

c. To prove that a sentence is valid, prove that its negation is unsatisfiable. I.e., negate it, convert to CNF, use resolution to prove a contradiction. We can use the above CNF result for the LHS.

$$\neg[[(Food \Rightarrow Party) \vee (Drinks \Rightarrow Party)] \Rightarrow [(Food \wedge Drinks) \Rightarrow Party]]$$
$$[(Food \Rightarrow Party) \vee (Drinks \Rightarrow Party)] \wedge \neg[(Food \wedge Drinks) \Rightarrow Party]$$
$$(\neg Food \vee \neg Drinks \vee Party) \wedge Food \wedge Drinks \wedge \neg Party$$

Each of the three unit clauses resolves in turn against the first clause, leaving an empty clause.

7.19

a. Each possible world can be expressed as the conjunction of all the literals that hold in the model. The sentence is then equivalent to the disjunction of all these conjunctions, i.e., a DNF expression.

b. A trivial conversion algorithm would enumerate all possible models and include terms corresponding to those in which the sentence is true; but this is necessarily exponential-time. We can convert to DNF using the same algorithm as for CNF except that we distribute $\wedge$ over $\vee$ at the end instead of the other way round.

c. A DNF expression is satisfiable if it contains at least one term that has no contradictory literals. This can be checked in linear time, or even during the conversion process. Any completion of that term, filling in missing literals, is a model.

d. The first steps give

$$(\neg A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee \neg A) \ .$$

Converting to DNF means taking one literal from each clause, in all possible ways, to generate the terms (8 in all). Choosing each literal corresponds to choosing the truth value of each variable, so the process is very like enumerating all possible models. Here, the first term is $(\neg A \wedge \neg B \wedge \neg C)$, which is clearly satisfiable.

e. The problem is that the final step typically results in DNF expressions of exponential size, so we require both exponential time AND exponential space.

7.20 The CNF representations are as follows:

S1: $(\neg A \vee B \vee E) \wedge (\neg B \vee A) \wedge (\neg E \vee A)$.
S2: $(\neg E \vee D)$.
S3: $(\neg C \vee \neg F \vee \neg B)$.
S4: $(\neg E \vee B)$.
S5: $(\neg B \vee F)$.
S6: $(\neg B \vee C)$.

We omit the DPLL trace, which is easy to obtain from the version in the code repository.

7.21 It is more likely to be solvable: adding literals to disjunctive clauses makes them easier to satisfy.

7.22

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

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

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

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

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

d. To encode the global constraint that there are M mines altogether, we can construct a disjunct with $\binom{M}{N}$ disjuncts, each of size N. Remember, $\binom{M}{N=M!/(M-N)!}$. So for

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

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

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

|-|1|-|1|-|1|-|1|-|1|-|1|-|1|-|1|-|

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

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

7.24 We omit the DPLL trace, which is easy to obtain from the version in the code repository. The behavior is very similar: the unit-clause rule in DPLL ensures that all known atoms are propagated to other clauses.

7.25

$$Locked^{t+1} \Leftrightarrow [Lock^t \vee (Locked^t \wedge \neg Unlock^t)] .$$

7.26 The remaining fluents are the orientation fluents ($FacingEast$ etc.) and $WumpusAlive$. The successor-state axioms are as follows:

$$\begin{aligned} FacingEast^{t+1} &\Leftrightarrow (FacingEast^t \wedge \neg(TurnLeft^t \vee TurnRight^t)) \\ &\vee (FacingNorth^t \wedge TurnRight^t) \\ &\vee (FacingSouth^t \wedge TurnLeft^t) \\ WumpusAlive^{t+1} &\Leftrightarrow WumpusAlive^t \wedge \neg(WumpusAhead^t \wedge HaveArrow^t \wedge Shoot^t) . \end{aligned}$$

The $WumpusAhead$ fluent does not need a successor-state axiom, since it is definable synchronously in terms of the agent location and orientation fluents and the wumpus location. The definition is extraordinarily tedious, illustrating the weakness of proposition logic. Note also that in the second edition we described a successor-state axiom (in the form of a circuit)

for $WumpusAlive$ that used the $Scream$ observation to infer the wumpus's death, with no need for describing the complicated physics of shooting. Such an axiom suffices for state estimation, but nor for planning.

7.27

The required modifications are to add definitional axioms such as

$$P_{3,1\ or\ 2,2} \Leftrightarrow P_{3,1} \vee P_{2,2}$$

and to include the new literals on the list of literals whose truth values are to be inferred at each time step.

One natural way to extend the 1-CNF representation is to add test additional non-literal sentences. The sentences we choose to test can depend on inferences from the current KB. This can work if the number of additional sentences we need to test is not too large.

For example, we can query the knowledge base to find out which squares we know have pits, which we know might have pits, and which states are breezy (we need to do this to compute the un-augmented 1-CNF belief state). Then, for each breezy square, test the sentence "one of the neighbours of this square which might have a bit does have a pit." For example, this would test $P_{3,1} \vee P_{2,2}$ if we had perceived a breeze in square (2,1). Under the Wumpus physics, this literal will be true iff the breezy square has no known pit around it.

Solutions for Chapter 8
First-Order Logic

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

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

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

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

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

using a digital plotter. In this case, the advantage of the map is really in the ease of inference combined with suitability for human "visual computing" apparatus.

e. Examples of other analogical representations:

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

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

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

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

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

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

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

8.6 Validity in first-order logic requires truth in all possible models:

a. $(\exists x\ x = x) \Rightarrow (\forall y\ \ \exists z\ y = z)$.
Valid. The LHS is valid by itself—in standard FOL, every model has at least one object; hence, the whole sentence is valid iff the RHS is valid. (Otherwise, we can find a model where the LHS is true and the RHS is false.) The RHS is valid because for every value of y in any given model, there is a z—namely, the value of y itself—that is identical to y.

b. $\forall x\ \ P(x) \vee \neg P(x)$.
Valid. For any relation denoted by P, every object x is either in the relation or not in it.

c. $\forall x\ \ Smart(x) \vee (x = x)$.
Valid. In every model, every object satisfies $x = x$, so the disjunction is satisfied regardless of whether x is smart.

8.7 This version of FOL, first studied in depth by Mostowski (1951), goes under the title of **free logic** (Lambert, 1967). By a natural extension of the truth values for empty conjunctions (true) and empty disjunctions (false), every universally quantified sentence is true in empty models and every existentially quantified sentence is false. The semantics also needs to be adjusted to handle the fact that constant symbols have no referent in an empty model.

Examples of sentences valid in the standard semantics but not in free logic include $\exists x\ \ x = x$ and $[\forall x\ \ P(x)] \Rightarrow [\exists x\ \ P(x)]$. More importantly, perhaps, the equivalence of $\phi \vee \exists x\ \ \psi$ and $\exists x\ \ \phi \vee \psi$ when x does not occur free in ϕ, which is used for putting sentences into CNF, does not hold.

One could argue that $\exists x\ \ x = x$, which simply states that the model is nonempty, is not naturally a valid sentence, and that it ought to be possible to contemplate a universe with no objects. However, experience has shown that free logic seems to require extra work to rule out the empty model in many commonly occurring cases of logical representation and reasoning.

8.8 The fact $\neg Spouse(George, Laura)$ does not follow. We need to assert that at most one person can be the spouse of any given person:

$$\forall x, y, z\ \ Spouse(x, z) \wedge Spouse(y, z) \Rightarrow x = y\ .$$

With this axiom, a resolution proof of $\neg Spouse(George, Laura)$ is straightforward.

If $Spouse$ is a unary function symbol, then the question is whether $\neg Spouse(Laura) = George$ follows from $Jim \neq George$ and $Spouse(Laura) = Jim$. The answer is yes, it does follow. They could not both be the value of the function applied to the same argument if they were different objects.

8.9

a. Paris and Marseilles are both in France.

(i) $In(Paris \wedge Marseilles, France)$.
(2) Syntactically invalid. Cannot use conjunction inside a term.

(ii) $In(Paris, France) \wedge In(Marseilles, France)$.
(1) Correct.

(iii) $In(Paris, France) \vee In(Marseilles, France)$.
(3) Incorrect. Disjunction does not express "both."

b. There is a country that borders both Iraq and Pakistan.

(i) $\exists\, c \quad Country(c) \wedge Border(c, Iraq) \wedge Border(c, Pakistan)$.
(1) Correct.

(ii) $\exists\, c \quad Country(c) \Rightarrow [Border(c, Iraq) \wedge Border(c, Pakistan)]$.
(3) Incorrect. Use of implication in existential.

(iii) $[\exists\, c \quad Country(c)] \Rightarrow [Border(c, Iraq) \wedge Border(c, Pakistan)]$.
(2) Syntactically invalid. Variable c used outside the scope of its quantifier.

(iv) $\exists\, c \quad Border(Country(c), Iraq \wedge Pakistan)$.
(2) Syntactically invalid. Cannot use conjunction inside a term.

c. All countries that border Ecuador are in South America.

(i) $\forall\, c \;\; Country(c) \wedge Border(c, Ecuador) \Rightarrow In(c, SouthAmerica)$.
(1) Correct.

(ii) $\forall\, c \;\; Country(c) \Rightarrow [Border(c, Ecuador) \Rightarrow In(c, SouthAmerica)]$.
(1) Correct. Equivalent to (i).

(iii) $\forall\, c \;\; [Country(c) \Rightarrow Border(c, Ecuador)] \Rightarrow In(c, SouthAmerica)$.
(3) Incorrect. The implication in the LHS is effectively an implication in an existential; in particular, it sanctions the RHS for all non-countries.

(iv) $\forall\, c \;\; Country(c) \wedge Border(c, Ecuador) \wedge In(c, SouthAmerica)$.
(3) Incorrect. Uses conjunction as main connective of a universal quantifier.

d. No region in South America borders any region in Europe.

(i) $\neg[\exists\, c, d \;\; In(c, SouthAmerica) \wedge In(d, Europe) \wedge Borders(c, d)]$.
(1) Correct.

(ii) $\forall\, c, d \;\; [In(c, SouthAmerica) \wedge In(d, Europe)] \Rightarrow \neg Borders(c, d)]$.
(1) Correct.

(iii) $\neg\forall\, c \;\; In(c, SouthAmerica) \Rightarrow \exists\, d \;\; In(d, Europe) \wedge \neg Borders(c, d)$.
(3) Incorrect. This says there is some country in South America that borders every country in Europe!

(iv) $\forall\, c \;\; In(c, SouthAmerica) \Rightarrow \forall\, d \;\; In(d, Europe) \Rightarrow \neg Borders(c, d)$.
(1) Correct.

e. No two adjacent countries have the same map color.

(i) $\forall\, x, y \;\; \neg Country(x) \vee \neg Country(y) \vee \neg Borders(x, y) \vee$
$\neg(MapColor(x) = MapColor(y))$.
(1) Correct.

(ii) $\forall\, x, y \;\; (Country(x) \wedge Country(y) \wedge Borders(x, y) \wedge \neg(x = y)) \Rightarrow$
$\neg(MapColor(x) = MapColor(y))$.
(1) Correct. The inequality is unnecessary because no country borders itself.

(iii) $\forall\, x, y \;\; Country(x) \wedge Country(y) \wedge Borders(x, y) \wedge$
$\neg(MapColor(x) = MapColor(y))$.
(3) Incorrect. Uses conjunction as main connective of a universal quantifier.

(iv) $\forall x, y \;\; (Country(x) \wedge Country(y) \wedge Borders(x, y)) \;\Rightarrow\; MapColor(x \neq y)$.
(2) Syntactically invalid. Cannot use inequality inside a term.

8.10

a. $O(E, S) \vee O(E, L)$.
b. $O(J, A) \wedge \exists p \;\; p \neq A \wedge O(J, p)$.
c. $\forall p \;\; O(p, S) \;\Rightarrow\; O(p, D)$.
d. $\neg\exists p \;\; C(J, p) \wedge O(p, L)$.
e. $\exists p \;\; B(p, E) \wedge O(p, L)$.
f. $\exists p \;\; O(p, L) \wedge \forall q \;\; C(q, p) \;\Rightarrow\; O(q, D)$.
g. $\forall p \;\; O(p, S) \;\Rightarrow\; \exists q \;\; O(q, L) \wedge C(p, q)$.

8.11

a. People who speak the same language understand each other.
b. Suppose that an extended interpretation with $x \rightarrow A$ and $y \rightarrow B$ satisfy

$$SpeaksLanguage(x, l) \wedge SpeaksLanguage(y, l)$$

for some l. Then from the second sentence we can conclude $Understands(A, B)$. The extended interpretation with $x \rightarrow B$ and $y \rightarrow A$ also must satisfy

$$SpeaksLanguage(x, l) \wedge SpeaksLanguage(y, l) \;,$$

allowing us to conclude $Understands(B, A)$. Hence, whenever the second sentence holds, the first holds.
c. Let $Understands(x, y)$ mean that x understands y, and let $Friend(x, y)$ mean that x is a friend of y.
(i) It is not completely clear if the English sentence is referring to mutual understanding and mutual friendship, but let us assume that is what is intended:
$\forall x, y \;\; Understands(x, y) \wedge Understands(y, x) \Rightarrow (Friend(x, y) \wedge Friend(y, x))$.
(ii) $\forall x, y, z \;\; Friend(x, y) \wedge Friend(y, z) \;\Rightarrow\; Friend(x, z)$.

8.12 This exercise requires a rewriting similar to the Clark completion of the two Horn clauses:

$$\forall n \;\; NatNum(n) \;\Leftrightarrow\; [n = 0 \vee \exists m \;\; NatNum(m) \wedge n = S(m)] \;.$$

8.13

a. The two implication sentences are

$$\forall s \;\; Breezy(s) \;\Rightarrow\; \exists r \;\; Adjacent(r, s) \wedge Pit(r)$$
$$\forall s \;\; \neg Breezy(s) \;\Rightarrow\; \neg\exists r \;\; Adjacent(r, s) \wedge Pit(r) \;.$$

The converse of the second sentence is

$$\forall s \;\; \exists r \;\; Adjacent(r, s) \wedge Pit(r) \;\Rightarrow\; Breezy(s)$$

which, combined with the first sentence, immediately gives

$$\forall s \;\; Breezy(s) \;\Leftrightarrow\; \exists r \;\; Adjacent(r, s) \wedge Pit(r) \;.$$

b. To say that a pit causes all adjacent squares to be breezy:

$$\forall s \;\; Pit(s) \;\Rightarrow\; [\forall r \;\; Adjacent(r,s) \;\Rightarrow\; Breezy(r)] \; .$$

This axiom allows for breezes to occur spontaneously with no adjacent pits. It would be incorrect to say that a non-pit causes all adjacent squares to be non-breezy, since there might be pits in other squares causing one of the adjacent squares to be breezy. But if *all* adjacent squares have no pits, a square is non-breezy:

$$\forall s \;\; [\forall r \;\; Adjacent(r,s) \;\Rightarrow\; \neg Pit(r)] \;\Rightarrow\; \neg Breezy(s) \; .$$

8.14 Make sure you write definitions with $\Leftrightarrow$. If you use $\Rightarrow$, you are only imposing constraints, not writing a real definition. Note that for aunts and uncles, we include the relations whom the OED says are more strictly defined as aunts-in-law and uncles-in-law, since the latter terms are not in common use.

$$Grandchild(c,a) \;\Leftrightarrow\; \exists b \;\; Child(c,b) \wedge Child(b,a)$$
$$Greatgrandparent(a,d) \;\Leftrightarrow\; \exists b,c \;\; Child(d,c) \wedge Child(c,b) \wedge Child(b,a)$$
$$Ancestor(a,x) \;\Leftrightarrow\; Child(x,a) \vee \exists b \;\; Child(b,a) \wedge Ancestor(b,x)$$
$$Brother(x,y) \;\Leftrightarrow\; Male(x) \wedge Sibling(x,y)$$
$$Sister(x,y) \;\Leftrightarrow\; Female(x) \wedge Sibling(x,y)$$
$$Daughter(d,p) \;\Leftrightarrow\; Female(d) \wedge Child(d,p)$$
$$Son(s,p) \;\Leftrightarrow\; Male(s) \wedge Child(s,p)$$
$$FirstCousin(c,d) \;\Leftrightarrow\; \exists p_1,p_2 \;\; Child(c,p_1) \wedge Child(d,p_2) \wedge Sibling(p_1,p_2)$$
$$BrotherInLaw(b,x) \;\Leftrightarrow\; \exists m \;\; Spouse(x,m) \wedge Brother(b,m)$$
$$SisterInLaw(s,x) \;\Leftrightarrow\; \exists m \;\; Spouse(x,m) \wedge Sister(s,m)$$
$$Aunt(a,c) \;\Leftrightarrow\; \exists p \;\; Child(c,p) \wedge [Sister(a,p) \vee SisterInLaw(a,p)]$$
$$Uncle(u,c) \;\Leftrightarrow\; \exists p \;\; Child(c,p) \wedge [Brother(a,p) \vee BrotherInLaw(a,p)]$$

There are several equivalent ways to define an mth cousin n times removed. One way is to look at the distance of each person to the nearest common ancestor. Define $Distance(c,a)$ as follows:

$$Distance(c,c) = 0$$
$$Child(c,b) \wedge Distance(b,a) = k \;\Rightarrow\; Distance(c,a) = k+1 \; .$$

Thus, the distance to one's grandparent is 2, great-great-grandparent is 4, and so on. Now we have

$$MthCousinNTimesRemoved(c,d,m,n) \;\Leftrightarrow$$
$$\exists a \;\; Distance(c,a) = m+1 \wedge Distance(d,a) = m+n+1 \; .$$

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

8.15 Although these axioms are sufficient to prove set membership when x is in fact a member of a given set, they have nothing to say about cases where x is not a member. For

example, it is not possible to prove that x is not a member of the empty set. These axioms may therefore be suitable for a logical system, such as Prolog, that uses negation-as-failure.

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

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

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

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

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

8.18 We need the following sentences:

$\forall s_1 \;\; Smelly(s_1) \;\Leftrightarrow\; \exists s_2 \;\; Adjacent(s_1, s_2) \wedge In(Wumpus, s_2)$
$\exists s_1 \;\; In(Wumpus, s_1) \wedge \forall s_2 \;\; (s_1 \neq s_2) \;\Rightarrow\; \neg In(Wumpus, s_2)$.

8.19

a. $\exists x \;\; Parent(Joan, x) \wedge Female(x)$.

b. $\exists^1 x \; Parent(Joan, x) \wedge Female(x)$.

c. $\exists x \;\; Parent(Joan, x) \wedge Female(x) \wedge [\forall y \;\; Parent(Joan, y) \;\Rightarrow\; y = x]$.
(This is sometimes abbreviated "$Female(\iota(x) Parent(Joan, x))$".)

d. $\exists^1 c \; Parent(Joan, c) \wedge Parent(Kevin, c)$.

e. $\exists c \;\; Parent(Joan, c) \wedge Parent(Kevin, c) \wedge \forall d, p \;\; [Parent(Joan, d) \wedge Parent(p, d)] \;\Rightarrow\; [p = Joan \vee p = Kevin]$

8.20

a. $\forall x \;\; Even(x) \;\Leftrightarrow\; \exists y \;\; x = y + y$.

b. $\forall x \;\; Prime(x) \;\Leftrightarrow\; \forall y, z \;\; x = y \times z \;\Rightarrow\; y = 1 \vee z = 1$.

c. $\forall x \;\; Even(x) \;\Rightarrow\; \exists y, z \;\; Prime(y) \wedge Prime(z) \wedge x = y + z$.

8.21 If we have $WA = red$ and $Q = red$ then we could deduce $WA = Q$, which is undesirable to both Western Australians and Queenslanders.

8.22

$$\forall k \;\; Key(k) \;\Rightarrow\; [\exists t_0 \;\; Before(Now, t_0) \wedge \forall t \;\; Before(t_0, t) \;\Rightarrow\; Lost(k, t)]$$
$$\forall s_1, s_2 \;\; Sock(s_1) \wedge Sock(s_2) \wedge Pair(s_1, s_2) \;\Rightarrow$$
$$[\exists t_1 \;\; Before(Now, t_1) \wedge \forall t \;\; Before(t_1, t) \;\Rightarrow\; Lost(s_1, t)] \vee$$
$$[\exists t_2 \;\; Before(Now, t_2) \wedge \forall t \;\; Before(t_2, t) \;\Rightarrow\; Lost(s_2, t)] \; .$$

Notice that the disjunction allows for both socks to be lost, as the English sentence implies.

8.23

a. "No two people have the same social security number."

$$\neg\exists x, y, n \;\; Person(x) \wedge Person(y) \;\Rightarrow\; [HasSS\#(x, n) \wedge HasSS\#(y, n)].$$

This uses $\Rightarrow$ with $\exists$. It also says that no person has a social security number because it doesn't restrict itself to the cases where x and y are not equal. Correct version:

$$\neg\exists x, y, n \;\; Person(x) \wedge Person(y) \wedge \neg(x = y) \wedge [HasSS\#(x, n) \wedge HasSS\#(y, n)]$$

b. "John's social security number is the same as Mary's."

$$\exists n \;\; HasSS\#(John, n) \wedge HasSS\#(Mary, n).$$

This is OK.

c. "Everyone's social security number has nine digits."

$$\forall x, n \;\; Person(x) \;\Rightarrow\; [HasSS\#(x, n) \wedge Digits(n, 9)].$$

This says that everyone has every number. $HasSS\#(x, n)$ should be in the premise:

$$\forall x, n \;\; Person(x) \wedge HasSS\#(x, n) \;\Rightarrow\; Digits(n, 9)$$

d. Here $SS\#(x)$ denotes the social security number of x. Using a function enforces the rule that everyone has just one.

$$\neg\exists x, y \;\; Person(x) \wedge Person(y) \;\Rightarrow\; [SS\#(x) = SS\#(y)]$$ manca x≠y
$$SS\#(John) = SS\#(Mary)$$
$$\forall x \;\; Person(x) \;\Rightarrow\; Digits(SS\#(x), 9)$$

8.24 In this exercise, it is best not to worry about details of tense and larger concerns with consistent ontologies and so on. The main point is to make sure students understand connectives and quantifiers and the use of predicates, functions, constants, and equality. Let the basic vocabulary be as follows:
$Takes(x, c, s)$: student x takes course c in semester s;
$Passes(x, c, s)$: student x passes course c in semester s;
$Score(x, c, s)$: the score obtained by student x in course c in semester s;
$x > y$: x is greater than y;
F and G: specific French and Greek courses (one could also interpret these sentences as referring to *any* such course, in which case one could use a predicate $Subject(c, f)$ meaning that the subject of course c is field f;
$Buys(x, y, z)$: x buys y from z (using a binary predicate with unspecified seller is OK but

less felicitous);
$Sells(x, y, z)$: x sells y to z;
$Shaves(x, y)$: person x shaves person y
$Born(x, c)$: person x is born in country c;
$Parent(x, y)$: x is a parent of y;
$Citizen(x, c, r)$: x is a citizen of country c for reason r;
$Resident(x, c)$: x is a resident of country c;
$Fools(x, y, t)$: person x fools person y at time t;
$Student(x)$, $Person(x)$, $Man(x)$, $Barber(x)$, $Expensive(x)$, $Agent(x)$, $Insured(x)$, $Smart(x)$, $Politician(x)$: predicates satisfied by members of the corresponding categories.

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

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

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

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

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

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

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

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

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

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

$$\forall x \;\; Person(x) \wedge \neg Born(x, UK) \wedge (\exists y \;\; Parent(y, x) \wedge Citizen(y, UK, Birth)) \;\Rightarrow\; Citizen(x, UK, Descent).$$

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

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

l. All Greeks speak the same language.

$$\forall x, y, l \;\; Person(x) \wedge [\exists r \;\; Citizen(x, Greece, r)] \wedge Person(y) \wedge [\exists r \;\; Citizen(y, Greece, r)] \\ \wedge Speaks(x, l) \;\Rightarrow\; Speaks(y, l)$$

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

$$\exists t \;\; Heard(W, DeathOf(N), t)$$
$$\forall x, e, t \;\; Heard(x, e, t) \;\Rightarrow\; Alive(x, t)$$
$$\forall x, e, t_2 \;\; Heard(x, e, t_2) \;\Rightarrow\; \exists t_1 \;\; Occurred(e, t_1) \wedge t_1 < t_2$$
$$\forall t_1 \;\; Occurred(DeathOf(x), t_1) \;\Rightarrow\; \forall t_2 \;\; t_1 < t_2 \;\Rightarrow\; \neg Alive(x, t_2)$$
$$\forall t_1, t_2 \;\; \neg(t_2 < t_1) \;\Rightarrow\; ((t_1 < t_2) \vee (t_1 = t_2))$$
$$\forall t_1, t_2, t_3 \;\; (t_1 < t_2) \wedge ((t_2 < t_3) \vee (t_2 = t_3)) \;\Rightarrow\; (t_1 < t_3)$$
$$\forall t_1, t_2, t_3 \;\; ((t_1 < t_2) \vee (t_1 = t_2)) \wedge (t_2 < t_3) \;\Rightarrow\; (t_1 < t_3)$$

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

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

$$Add_1(0, 0, 0) = [0, 0]$$
$$Add_1(0, 0, 1) = [0, 1]$$
$$Add_1(0, 1, 0) = [0, 1]$$
$$Add_1(0, 1, 1) = [1, 0]$$
$$Add_1(1, 0, 0) = [0, 1]$$
$$Add_1(1, 0, 1) = [1, 0]$$
$$Add_1(1, 1, 0) = [1, 0]$$
$$Add_1(1, 1, 1) = [1, 1]$$

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

$$Add_n([], [], b) = [b]$$
$$Add_1(b_1, b_2, b) = [b_3, b_4] \;\Rightarrow\; Add_n([b_1|x_1], [b_2|x_2], b) = [b_3|Add_n(x_1, x_2, b_4)]$$

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

connections:

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

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

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

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

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

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

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

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

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

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

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

8.27 The answers here will vary by country. The two key rules for UK passports are given above.

8.28

a. $W(G,T)$.
b. $\neg W(G,E)$.
c. $W(G,T) \vee W(M,T)$.
d. $\exists\, s \;\; W(J,s)$.
e. $\exists\, x \;\; C(x,R) \wedge O(J,x)$.
f. $\forall\, s \;\; S(M,s,R) \;\Rightarrow\; W(M,s)$.
g. $\neg[\exists\, s \;\; W(G,s) \wedge \exists\, p \;\; S(p,s,R)]$.
h. $\forall\, s \;\; W(G,s) \;\Rightarrow\; \exists\, p,a \;\; S(p,s,a)$.
i. $\exists\, a \;\; \forall\, s \;\; W(J,s) \;\Rightarrow\; \exists\, p \;\; S(p,s,a)$.
j. $\exists\, d,a,s \;\; C(d,a) \wedge O(J,d) \wedge S(B,T,a)$.
k. $\forall\, a \;\; [\exists\, s \;\; S(M,s,a)] \;\Rightarrow\; \exists\, d \;\; C(d,a) \wedge O(J,d)$.
l. $\forall\, a \;\; [\forall\, s,p \;\; S(p,s,a) \;\Rightarrow\; S(B,s,a)] \;\Rightarrow\; \exists\, d \;\; C(d,a) \wedge O(J,d)$.