Problem-solving: Informed Search Strategies, Heuristic functions

Logical Agents: Knowledge‑based agents, The Wumpus world, Logic, Propositional logic, Reasoning patterns in Propositional Logic

Chapter 3 - 3.5, 7.6

Chapter 7 - 7.1, 7.2, 7.3, 7.4

Text book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson, 2015

Data: Raw facts, unformatted information.

Information: It is the result of processing, manipulating and organizing data in response to aspecific need. Information relates to the understanding of the problem domain.

Knowledge: It relates to the understanding of the solution domain – what to do?

Intelligence: It is the knowledge in operation towards the solution – how to do? How to applythe solution?

This section shows how an **informed search** strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy. The general approach we consider is called **best-first search**. Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, f(n). The evaluation function is constructed as a cost estimate, so the node with the *lowest* evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search (Figure 3.14), except for the use of f instead of g to order the priority queue.

The choice of f determines the search strategy. (For example, as Exercise 3.21 shows, best-first tree search includes depth-first search as a special case.) Most best-first algorithms include as a component of f a **heuristic function**, denoted h(n):

h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state. (Notice that h(n) takes a *node* as input, but, unlike g(n), it depends only on the *state* at that node.) For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest. Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. For now, we consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then h(n)=0. The remainder of this section covers two ways to use heuristic information to guide search.

3.5.1 Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is

likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n). Let us see how this works for route-finding problems in Romania; we use the **straight- line distance** heuristic, which we will call h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure 3.22. For example, h_{SLD} (In(Arad))=366. Notice that the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic. Figure 3.23 shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is closest. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever

expanding a node that is not on the solution path; hence, its search cost is minimal. It is not optimal, however: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti. This shows why the algorithm is called "greedy"—at each step it tries to get as close to the goal as it can.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

The worst-case time and space complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

3.5.2 A* search: Minimizing the total estimated solution cost

The most widely known form of best-first search is called **A∗** A **search** (pronounced "A-star ∗ search"). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

f(n) = g(n) + h(n) .

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A∗ search is both complete and optimal. The algorithm is identical to except that A∗ uses g + h instead of g.

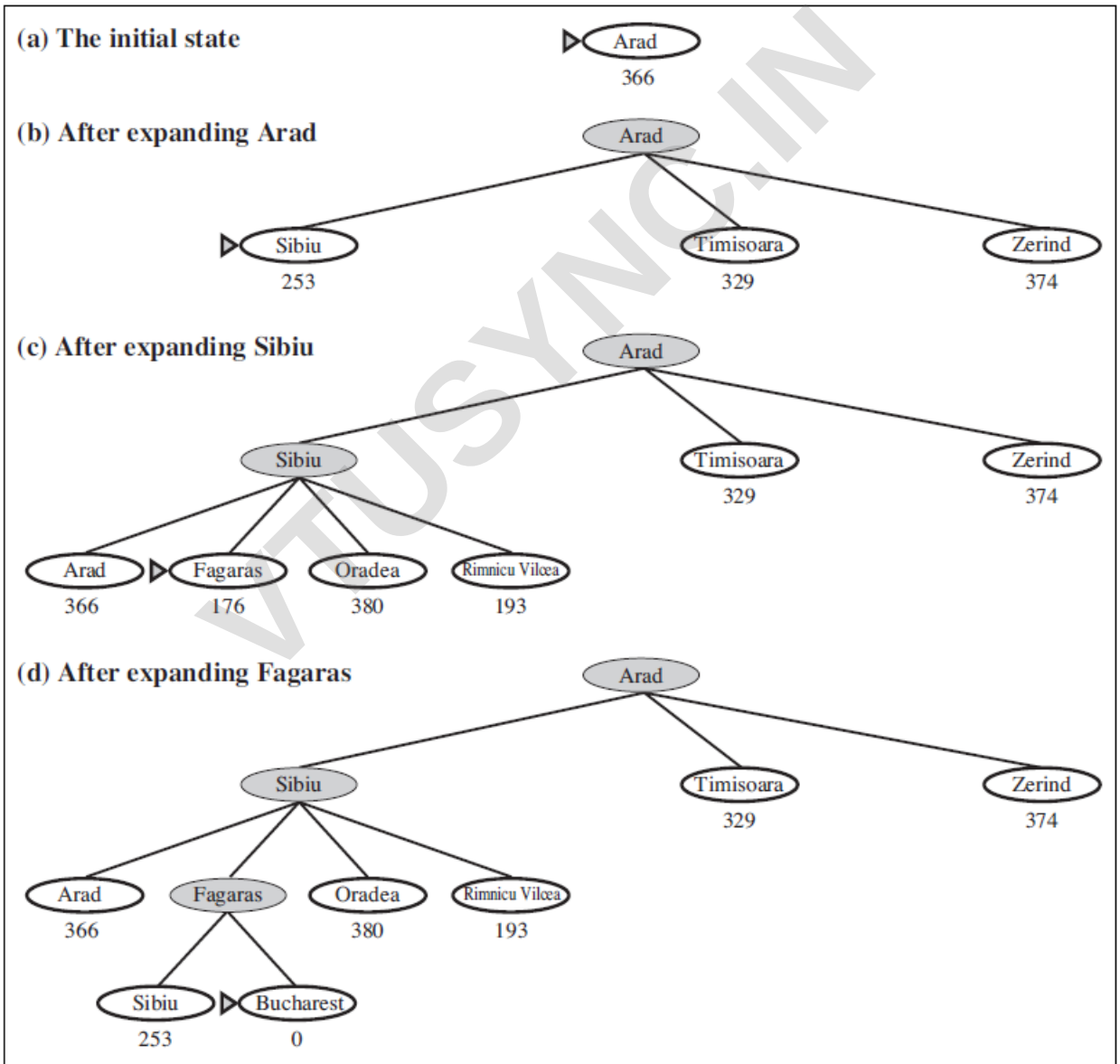

Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Conditions for optimality: Admissibility and consistency

The first condition we require for optimality is that h(n) be an **admissible heuristic**. An admissible heuristic is one that *never overestimates* the cost to reach the goal. Because g(n) is the actual cost to reach n along the current path, and f(n)=g(n) + h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n.

Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is. An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate. In Figure 3.24, we show the progress of an A∗ tree search for Bucharest. The values of g are computed from the step costs in Figure 3.2, and the values of h_{SLD} are given in Figure 3.22. Notice in particular that Bucharest first appears on the frontier at step (e), but it is not selected for expansion because its f-cost (450) is higher than that of Pitesti (417). Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450. A second, slightly stronger condition called **consistency** (or sometimes **monotonicity**) is required only for applications of A∗ to graph search.A heuristic h(n) is consistent if, for every node n and every successor n of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n: $h(n) \leq c(n, a, n) + h(n)$.

This is a form of the general **triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n, n, and the goal Gn closest to n. For an admissible heuristic, the inequality makes perfect sense:

if there were a route from n to G_n via n that was cheaper than h(n), that would violate the property that h(n) is a lower bound on the cost to reach Gn.It is fairly easy to show (Exercise 3.29) that every consistent heuristic is also admissible. Consistency is therefore a stricter requirement than admissibility, but one has to work quite hard to concoct heuristics that are admissible but not consistent. All the admissible heuristics we discuss in this chapter are also consistent. Consider, for example, h_{SLD}. We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance and that the straight-line distance between n and n is no greater than c(n, a, n).

Hence, h_{SLD} is a consistent heuristic.

Optimality of A*

As we mentioned earlier, A∗ has the following properties: *the tree-search version of* A * *is optimal if* h(n)

is admissible, while the graph-search version is optimal if h(n) *is consistent.* We show the second of these two claims since it is more useful. The argument essentially mirrors the argument for the optimality of uniform-cost search, with g replaced by f—just as in the A∗ algorithm itself.

The first step is to establish the following: *if* h(n) *is consistent, then the values of* f(n) *along any path are nondecreasing.* The proof follows directly from the definition of consistency. Suppose n is a successor of n; then

g(n)=g(n) + c(n, a, n) for some action a, and we have

f(n) = g(n) + h(n) = g(n) + c(n, a, n) + h(n) ≥ g(n) + h(n) = f(n) .

The next step is to prove that *whenever* A∗ *selects a node* n *for expansion, the optimal path to that node has been found.* Were this not the case, there would have to be another frontier node n on the optimal path from the start node to n, by the graph separation property of Figure 3.9; because f is nondecreasing along any path, n would have lower f-cost than n and would have been selected first.

From the two preceding observations, it follows that the sequence of nodes expanded by A∗ using is in nondecreasing order of f(n). Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have h=0) and all later goal nodes will be at least as expensive.

The fact that f-costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map. Figure 3.25 shows an example. Inside the contour labeled 400, all nodes have f(n) less than or equal to 400, and so on. Then, because A∗ expands the frontier node of lowest f-cost, we can see that an A∗ search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

With uniform-cost search (A∗ search using h(n) = 0), the bands will be "circular" around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path. If C∗ is the cost of the optimal solution path, then we can say the following:

· A∗ expands all nodes with f(n) < C∗.

· A∗ might then expand some of the nodes right on the "goal contour" (where f(n) = C∗) before selecting a goal node. Completeness requires that there be only finitely many nodes with cost less than or equal to C ∗, a condition that is true if all step costs exceed some finite _ and if b is finite.

Notice that A∗ expands no nodes with f(n) > C* for example, Timisoara is not expanded in Figure 3.24 even though it is a child of the root. We say that the subtree below Timisoara is **pruned**; because h_{SLD} is

admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI. One final observation is that among optimal algorithms of this type—algorithms that extend search paths from the root and use the same heuristic information—A∗ is **optimally efficient** for any given consistent heuristic. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A∗ (except possibly through tie-breaking among nodes with
f(n)=C∗). This is because any algorithm that *does not* expand all nodes with f(n) < C ∗
runs the risk of missing the optimal solution.

That A∗ search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A∗ is the answer to all our searching needs. The catch is that, for most problems, the number of states within the goal contour search space is still exponential in the length of the solution. The details of the analysis are beyond the scope of this book, but the basic results are as follows. For problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the **absolute error** or the **relative error** of the heuristic. The absolute error is defined as $\Delta \equiv h* - h$,
where h∗ is the actual cost of getting from the root to the goal, and the relative error is defined as
$\equiv (h* - h)/h*$.

The complexity results depend very strongly on the assumptions made about the state space. The simplest model studied is a state space that has a single goal and is essentially a tree with reversible actions. (The 8-puzzle satisfies the first and third of these assumptions.) In this case, the time complexity of A∗ is exponential in the maximum absolute error, that is,$O(b^{\Delta})$. For constant step costs, we can write this as O(b_d), where d is the solution depth. For almost all heuristics in practical use, the absolute error is at least proportional to the pathcost h ∗, so _ is constant or growing and the time complexity is exponential in d. We can also see the effect of a more accurate heuristic: O(b_d)=O((b_)d), so the effective branching factor (defined more formally in the next section) is b_.

When the state space has many goal states—particularly *near-optimal* goal states—the search process can be led astray from the optimal path and there is an extra cost proportional to the number of goals whose cost is within a factor _ of the optimal cost. Finally, in the general case of a graph, the situation is even worse. There can be exponentially many states with f(n) < C ∗ even if the absolute error is bounded by a constant. For example, consider a version of the vacuum world where the agent can clean up any square for unit cost without even having to visit it: in that case, squares can be cleaned in any order. With N

initially dirty squares, there are 2N states where some subset has been cleaned and all of them are on an optimal solution path—and hence satisfy f(n) < C ∗—even if the heuristic has an error of 1. The complexity of A∗ often makes it impractical to insist on finding an optimal solution. One can use variants of A∗ that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search. In Section 3.6, we look at the question of designing good heuristics. Computation time is not, however, A∗'s main drawback. Because it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), A∗ usually runs out of space long before it runs out of time. For this reason, A∗ is not practical for many large-scale problems. There are, however, algorithms that overcome the space problem without sacrificing optimality or completeness, at a small cost in execution time.

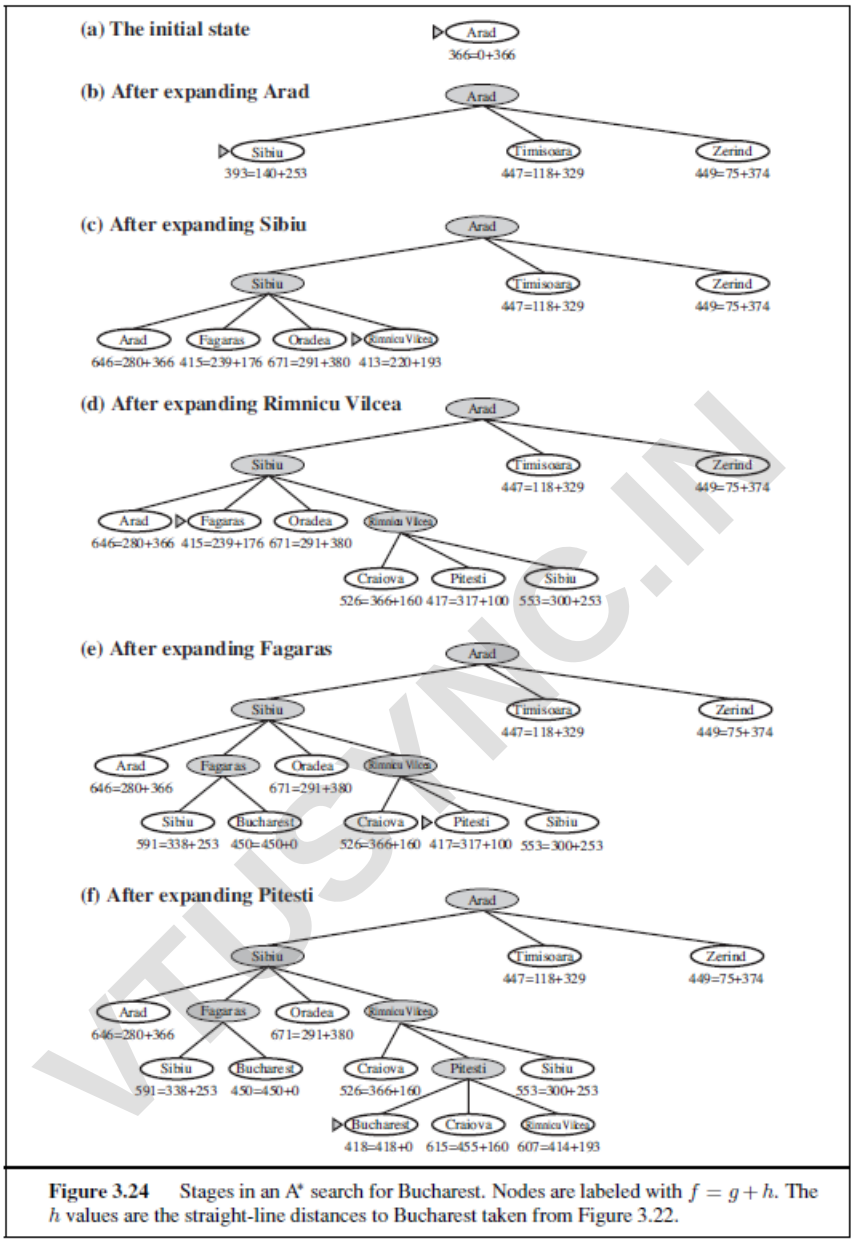

Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

Figure 3.26 The algorithm for recursive best-first search.

3.5.3 Memory-bounded heuristic search

The simplest way to reduce memory requirements for A∗ is to adapt the idea of iterative deepening to the heuristic search context, resulting in the **iterative-deepening A∗** (IDA∗) algorithmA∗ The main difference between IDA∗ and standard iterative deepening is that the cutoff used is the f-cost (g+h) rather than the depth; at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. IDA∗ is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes. Unfortunately, it suffers from the same difficulties with real valued costs as does the iterative version of uniform-cost search.

This section briefly examines two other memory-bounded algorithms, called RBFS and MA∗.(Memory Bounded A*)

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. The algorithm is shown in Figure 3.26. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back

to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a **backed-up value**—the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. Figure 3.27 shows how RBFS reaches Bucharest.

RBFS is somewhat more efficient than IDA∗, but still suffers from excessive node regeneration. In the example in Figure 3.27, RBFS follows the path via Rimnicu Vilcea, then "changes its mind" and tries

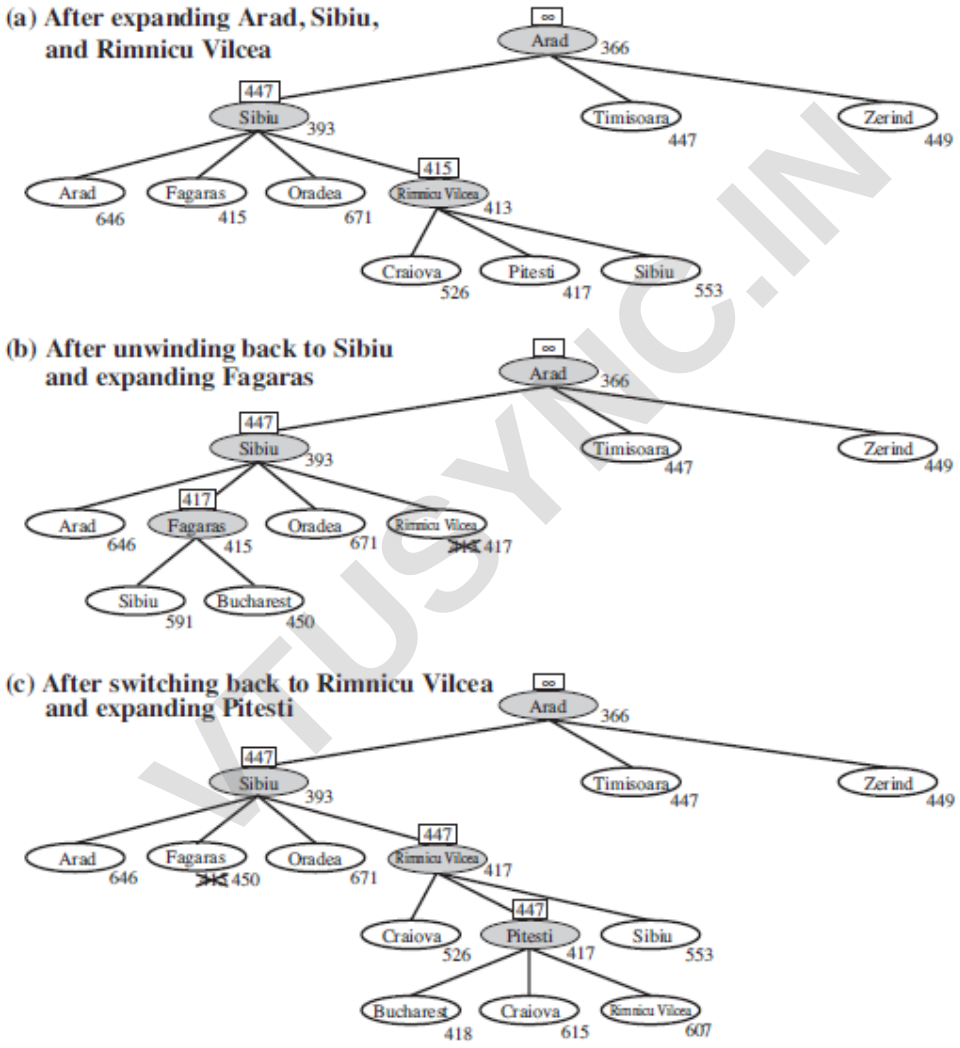

Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

Fagaras, and then changes its mind back again. These mind changes occur because every time the current best path is extended, its f-value is likely to increase—h is usually less optimistic for nodes closer to the goal. When this happens, the second-best path might become the best path, so the search has to backtrack to follow it . Each mind change corresponds to an iteration of IDA∗ and could require many reexpansion of forgotten nodes to recreate the best path and extend it one more node. Like A∗ tree search, RBFS is an optimal algorithm if the heuristic function h(n) is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. IDA∗ and RBFS suffer from using *too little* memory. Between iterations, IDA∗ retains only a single number: the current f-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up reexpanding the same states many times over. Furthermore, they suffer the potentially exponential increase in complexity associated with redundant paths in graphs.

SMA∗ is—well—simpler, so

SMA* we will describe it. SMA∗ proceeds just like A∗, expanding the best leaf until memory is full.

At this point, it cannot add a new node to the search tree without dropping an old one. SMA∗ always drops the *worst* leaf node—the one with the highest f-value. Like RBFS, SMA∗ then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA∗ regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that, if all the descendants of a node n are forgotten, then we will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n. The complete algorithm is too complicated to reproduce here, but there is one subtlety worth mentioning. We said that SMA∗ expands the best leaf and deletes the worst leaf. What if *all* the leaf nodes have the same f-value? To avoid selecting the same node for deletion and expansion, SMA∗ expands the *newest* best leaf and deletes the *oldest* worst leaf. These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then *even if it is on an optimal solution path*, that solution is not reachable with the available memory. Therefore,the node can be discarded exactly as if it had no successors.

HEURISTIC FUNCTIONS

The effect of heuristics on performance

One way to characterize the quality of a heuristic is the effective branching factor b. If the total number of nodes generated by A∗ for a particular problem is N and the solution depth is d, then b ∗ is the branching factor that a uniform tree of depth d would have to have in order to contain N + 1 nodes. Thus, $N + 1 = 1+b^* + (b^*)^2 + \cdots + (b^*)^d$

For example, if A∗ finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems.

Generating Admissible heuristics from relaxed problems

A problem with fewer restrictions on the actions is called a **relaxed problem**. The state-space graph of the relaxed problem is a *supergraph* of the original state space because the removal of restrictions creates added edges in the graph. Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have *better* solutions if the added edges provide short cuts. Hence, *the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.*

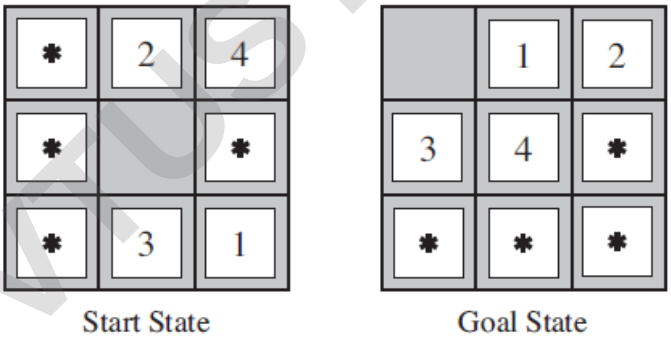

Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

Generating admissible heuristics from subproblems: Pattern databases

Admissible heuristics can also be derived from the solution cost of a **subproblem** of a given problem. For example, Figure 3.30 shows a subproblem of the 8-puzzle instance. The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions. Clearly , the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem. It turns out to be more accurate than Manhattan distance in some cases. The idea behind **pattern databases** is to store these exact solution costs for every possible

subproblem instance—in our example, every possible configuration of the four tiles and the blank. Then we compute an admissible heuristic hDB for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database. The database itself is constructed by searching back from the goal and recording the cost of each new pattern encountered; the expense of this search is amortized over many subsequent problem instances.

LOGICAL AGENTS

KNOWLEDGE BASED AGENTS

The central component of a knowledge-based agent is its KNOWLEDGE BASE knowledge base, or KB. A knowl edge base is a set of sentences. Each sentence is expressed in a language called a knowledge representation language and represents some assertion about the world. Sometimes we dignify a sentence with the name axiom, when the sentence is taken as given without being derived from other sentences. There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively.Both operations may involve inference—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously. Later

in this chapter, we will be more precise about the crucial word "follow." For now, take it to mean that the inference process should not make things up as it goes along. Figure 7.1 shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, KB, which may initially contain some background knowledge.Each time the agent program is called, it does three things. First, it TELLs the knowledge

base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action. The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other. MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time. MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time. Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK.

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

THE WUMPUS WORLD

The wumpus world is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence. A sample wumpus world is shown in Figure 7.2. The precise definition of the task environment is given, as suggested in Section 2.3, by the PEAS description

· **Performance measure**: +1000 for climbing out of the cave with the gold, –1000 for falling into a pit or being eaten by the wumpus, –1 for each action taken and –10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.

· **Environment**: A 4×4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.

· **Actuators**: The agent can move *Forward*, *TurnLeft* by 90◦, or *TurnRight* by 90◦. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits

a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].

· **Sensors**: The agent has five sensors, each of which gives a single bit of information:

– In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.

– In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.

– In the square where the gold is, the agent will perceive a *Glitter*.

– When an agent walks into a wall, it will perceive a *Bump*.

– When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get

[Stench, Breeze, None, None, None].

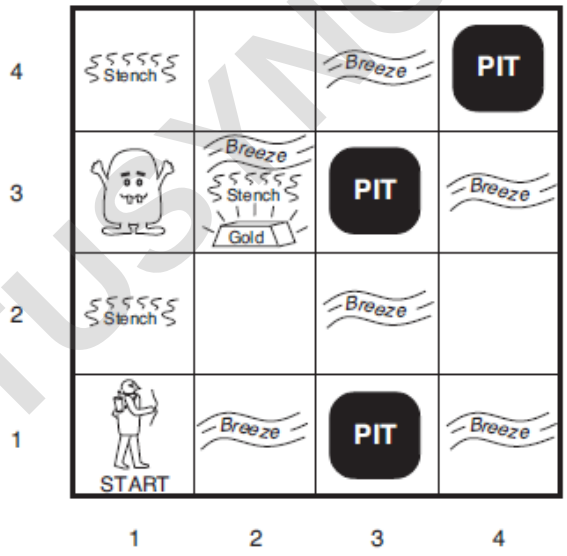

Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

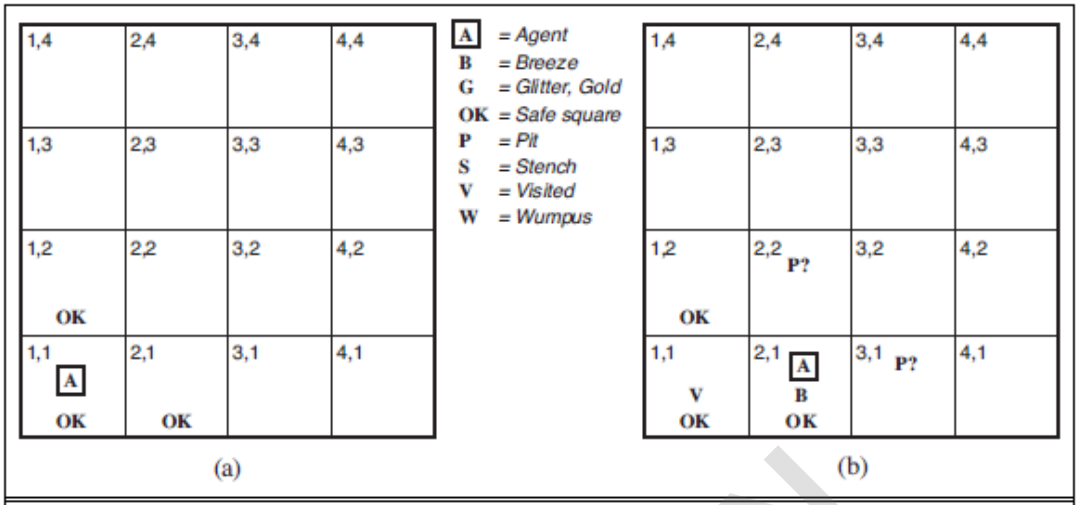

Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After one move, with percept [*None*, *Breeze*, *None*, *None*, *None*].

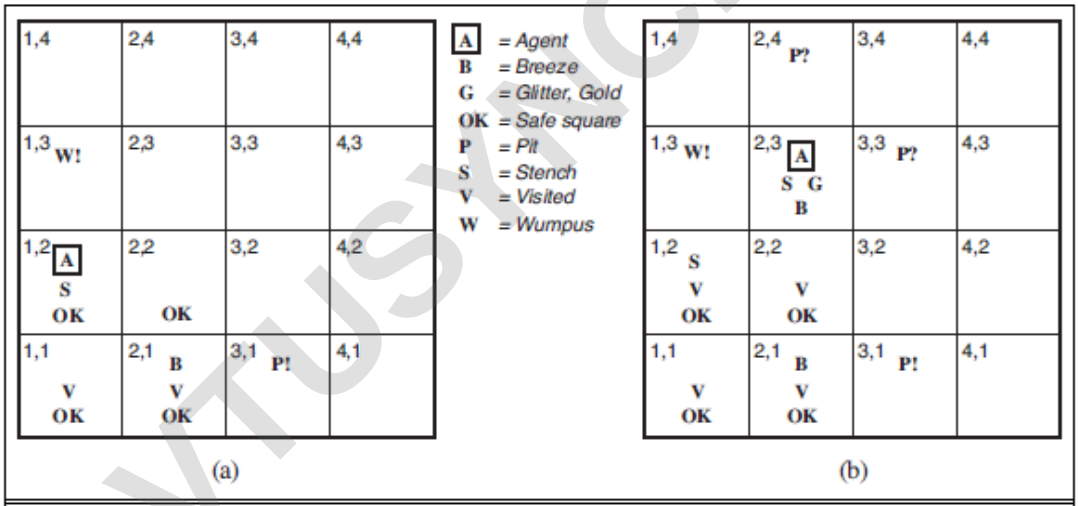

Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

The agent if decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation "P?" in Figure 7.3(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step. The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 7.4(b). In [2,3], the agent detects a glitter, so it should grab the gold and then return home.

LOGIC

Knowledge bases consist of sentences. These sentences are expressed according to SYNTAX the syntax of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: "x + y = 4" is a well-formed sentence, whereas "x4y+ =" is not. A logic must also define the semantics or meaning of sentences. The semantics defines the truth of each sentence with respect to each possible world. For example, the semantics for arithmetic specifies that the sentence

"x + y =4" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no "in between."

If a sentence α is true in model m, we say that m **satisfies** α or sometimes m **is a model of** α. We use the notation M(α) to mean the set of all models of α. Now that we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, we write $\alpha \models \beta$ to mean that the sentence α entails the sentence β. The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write $\alpha \models \beta$ if and only if $M(\alpha) \subseteq M(\beta)$.

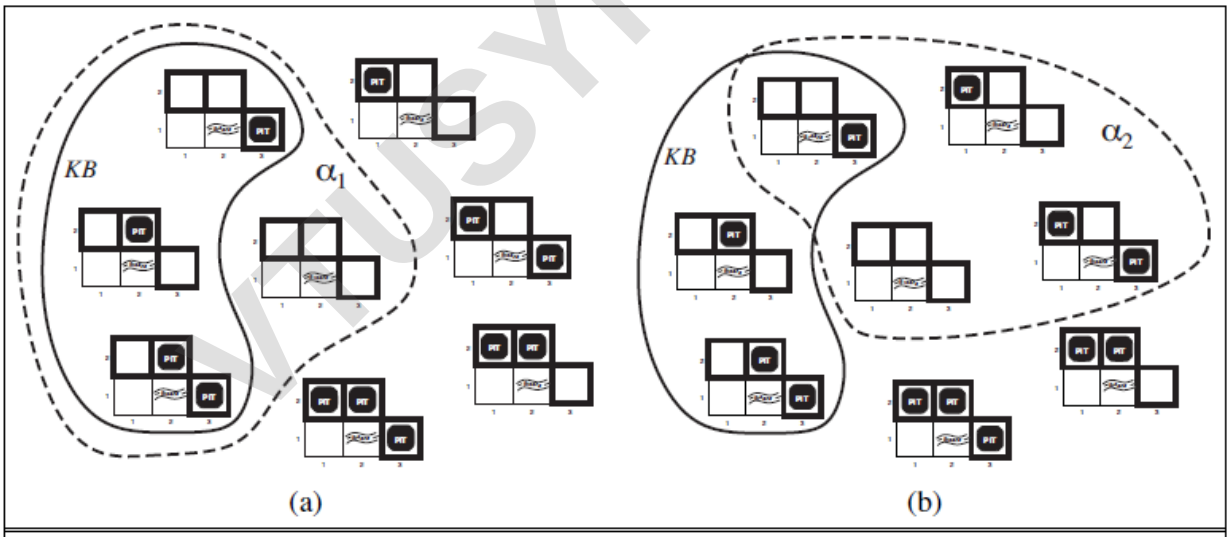

Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

The preceding example not only illustrates entailment but also shows how the definition of entailment can be applied to derive conclusions—that is, to carry out **logical inference**.The inference algorithm illustrated in Figure 7.5 is called **model checking**,

because it enumerates all possible models to check that α is true in all models in which KB is true, that isthat M(KB) ⊆ M(α).

In understanding entailment and inference, it might help to think of the set of all consequences of KB as a haystack and of α as a needle. Entailment is like the needle being in the haystack; inference is like finding it. This distinction is embodied in some formal notation: if an inference algorithm i can derive α from KB, we write KB _i α ,which is pronounced "α is derived from KB by i" or "i derives α from KB." An inference algorithm that derives only entailed sentences is called **sound** or **truth preserving**. Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent needles. It is easy to see that model checking, when it is applicable, is a sound procedure. The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue. Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases. We have described a reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, *if* KB *is true in the* real *world, then any sentence* α *derived from* KB *by a sound inference procedure is also true in the real world.* So, while an inference process operates on "syntax"—internal physical configurations such as bits in registers or patterns of electrical blips in brains—the process corresponds to the real-world relationship whereby some aspect of the real world is the case by virtue of other aspects of the real world being the case. This correspondence between world and representation is illustrated in Figure 7.6.

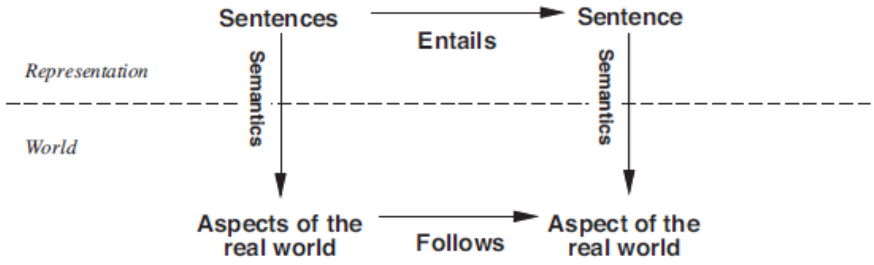

Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

PROPOSITIONAL LOGIC: A VERY SIMPLE LOGIC

SYNTAX

The syntax of ATOMIC SENTENCES propositional logic defines the allowable sentences. The atomic sentences consist of a single proposition symbol. Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, W1,3 and North. The names are arbitrary but are often chosen to have some mnemonic value—we use W1,3 to stand for the proposition that the wumpus is in [1,3]. (Remember that symbols such as W1,3 are *atomic*, i.e., W, 1, and 3 are not meaningful parts of the symbol.) There are two proposition symbols with fixed meanings: True is the always-true proposition and False is the always-false proposition. Complex sentences are constructed from simpler sentences, using parentheses and logical connectives. There are five connectives in common use:

NEGATION ¬ (not). A sentence such as ¬W1,3 is called the negation of W1,3. A literal is either an atomic sentence (a positive literal) or a negated atomic sentence (a negative literal).

CONJUNCTION ∧ (and). A sentence whose main connective is ∧, such as W1,3 ∧ P3,1, is called a con junction; its parts are the conjuncts. (The ∧ looks like an "A" for "And.")

DISJUNCTION ∨ (or). A sentence using ∨, such as (W1,3∧P3,1)∨W2,2, is a disjunction of the disjuncts (W1,3 ∧ P3,1) and W2,2.

IMPLICATION ⇒ (implies). A sentence such as (W1,3∧P3,1) ⇒ ¬W2,2 is called an implication . Its premise or antecedent is (W1,3 ∧P3,1), and its conclusion or consequent is ¬W2,2. Implications are also known as rules or if–then statements. The implication symbol is sometimes written in other books as ⊃ or →.

BICONDITIONAL ⇔ (if and only if). The sentence W1,3 ⇔ ¬W2,2 is a biconditional.

Sentence	→	*AtomicSentence* | *ComplexSentence*
AtomicSentence	→	*True* | *False* | *P* | *Q* | *R* | ...
ComplexSentence	→	(*Sentence*) | [*Sentence*]
	|	¬ *Sentence*
	|	*Sentence* ∧ *Sentence*
	|	*Sentence* ∨ *Sentence*
	|	*Sentence* ⇒ *Sentence*
	|	*Sentence* ⇔ *Sentence*
OPERATOR PRECEDENCE	:	¬, ∧, ∨, ⇒, ⇔

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Figure 7.7 gives a formal grammar of propositional logic.The BNF grammar by itself is ambiguous; a sentence with several operators can be parsed by the grammar in multiple ways. To eliminate the ambiguity we define a precedence for each operator. The "not" operator (¬) has the highest precedence, which means that in the sentence ¬A ∧ B the ¬ binds most tightly, giving us the equivalent of (¬A)∧B rather than ¬(A∧B). (The notation for ordinary arithmetic is the same: −2+4 is 2, not –6.) When in doubt, use parentheses to

make sure of the right interpretation. Square brackets mean the same thing as parentheses; the choice of square brackets or parentheses is solely to make it easier for a human to read a sentence.

SEMANTICS

The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply fixes the truth value—true or false—for every proposition symbol. For example, if the sentences in the knowledge base make use of the proposition symbols P1,2, P2,2, and P3,1, then one possible model is m1 = {P1,2 =false, P2,2 =false, P3,1 =true}. Notice, however, that the models are purely mathematical objects with no necessary connection to wumpus worlds. P1,2 is just a symbol; it might mean "there is a pit in [1,2]" or "I'm in Paris today and tomorrow." The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives. Atomic sentences are easy:

· True is true in every model and False is false in every model.

· The truth value of every other proposition symbol must be specified directly in the model.

For example, in the model m1 given earlier, P1,2 is false. For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m (here "iff" means "if and only if"):

· ¬P is true iff P is false in m.

· P ∧ Q is true iff both P and Q are true in m.

· P ∨ Q is true iff either P or Q is true in m.

· P ⇒ Q is true unless P is true and Q is false in m.

· P ⇔ Q is true iff P and Q are both true or both false in m.

The rules can also be expressed with truth tables that specify the truth value of a complex sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8. From these tables, the truth value of any sentence s can be computed with respect to any model m by a simple recursive evaluation.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

A simple inference procedure

Our goal now is to decide whether KB |= α for some sentence α. For example, is ¬P1,2 entailed by our KB? Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. Models are assignments of true or false to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are B1,1, B2,1, P1,1, P1,2, P2,1, P2,2, and P3,1. With seven symbols, there are 27 =128 possible models; in three of these, KB is true (Figure 7.9). In those three models, ¬P1,2 is true, hence there is no pit in [1,2]. On the other hand, P2,2 is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2]. Figure 7.9 reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is shown in Figure 7.10. The algorithm is sound because it implements directly the definition of entailment, and complete because it works for any KB and α and always terminates—there are only finitely

many models to examine. Of course, “finitely many” is not always the same as “few.” If KB and α contain n symbols in all, then there are 2n models. Thus, the time complexity of the algorithm is O(2n). (The space complexity is only O(n) because the enumeration is depth-first.) Unfortunately, propositional entailment is co-NP-complete so every known inference algorithm for propositional logic has a worst-case complexity that is exponential in the size of the input**.**

A simple Knowledge Base

$P_{x,y}$ is true if there is a pit in $[x, y]$.
$W_{x,y}$ is true if there is a wumpus in $[x, y]$, dead or alive.
$B_{x,y}$ is true if the agent perceives a breeze in $[x, y]$.
$S_{x,y}$ is true if the agent perceives a stench in $[x, y]$.

The sentences we write will suffice to derive ¬P1,2 (there is no pit in [1,2]) .We label each sentence Ri so that we can refer to them.

There is no pit in [1,1]: R_1 : ¬P1,1 .

• A square is breezy if and only if there is a pit in a neighboring square.

This has to be stated for each square; for now, we include just the relevant squares:

R2 : B1,1 ⇔ (P1,2 ∨ P2,1) .

R3 : B2,1 ⇔ (P1,1 ∨ P2,2 ∨ P3,1) .

- The preceding sentences are true in all wumpus worlds.
- Now we include the breeze percepts for the first two squares visited in the specific world the agent is in R4 : ¬B1,1 . R5 : B2,1 .

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 A truth table constructed for the knowledge base given in the text. *KB* is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].


```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, { })

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
      if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
      else return true // when KB is false, always return true
  else do
      P ← FIRST(symbols)
      rest ← REST(symbols)
      return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
              and
              TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword "**and**" is used here as a logical operation on its two arguments, returning *true* or *false*.

- **PL-TRUE?(KB, model)**: This function checks if a given model makes the knowledge base KB true.

- **Vacuous Truth**: If KB is false in a model, we return true immediately, because KB can't falsify α\alphaα in that case (any false knowledge base trivially entails any statement).
- The recursion explores all possible combinations of truth assignments for the propositional symbols in KB and α.

This approach is essentially a brute-force truth table method for checking entailment, where every possible combination of truth values for the propositional variables is tested. While it works correctly, it's computationally expensive (exponential in the number of symbols), making it impractical for large knowledge bases.

How It Works:

1. The TT-ENTAILS? function starts the process by identifying all the symbols involved in both the knowledge base KB and the query α.
2. It then calls TT-CHECK-ALL, which checks all possible truth assignments to these symbols.
3. For each truth assignment, it checks whether KB is true and, if so, whether α is also true. If all such assignments satisfy KB $\Rightarrow$ α, KB$\Rightarrow$α, then KB entails α and the function returns true. If any assignment violates the entailment (i.e., KB is true and α is false), it returns false.