- **Syllabus: Informed Search Strategies:** Heuristic functions

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

- **Text book 1:Chapter 3 - 3.5, 3.6**

  Chapter 4 – 4.1, 4.2 Chapter 7- 7.1, 7.2, 7.3, 7.4

Informed (Heuristic) Search Strategies

- 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 construed 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.**

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.**

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.

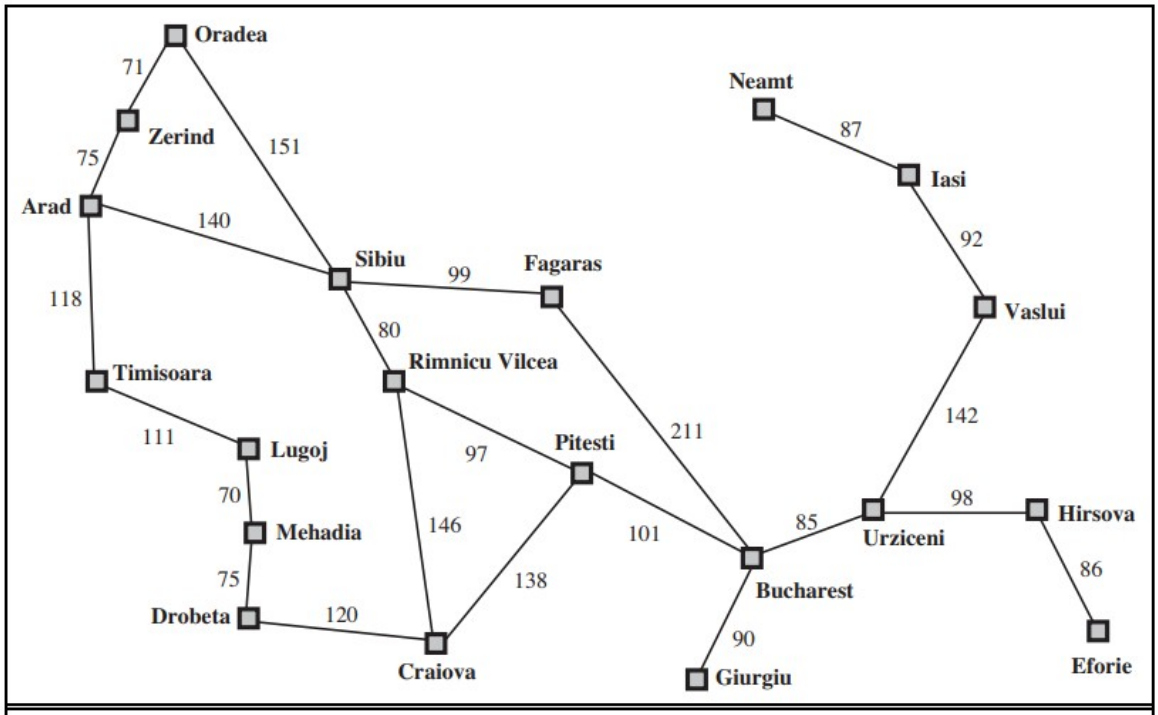
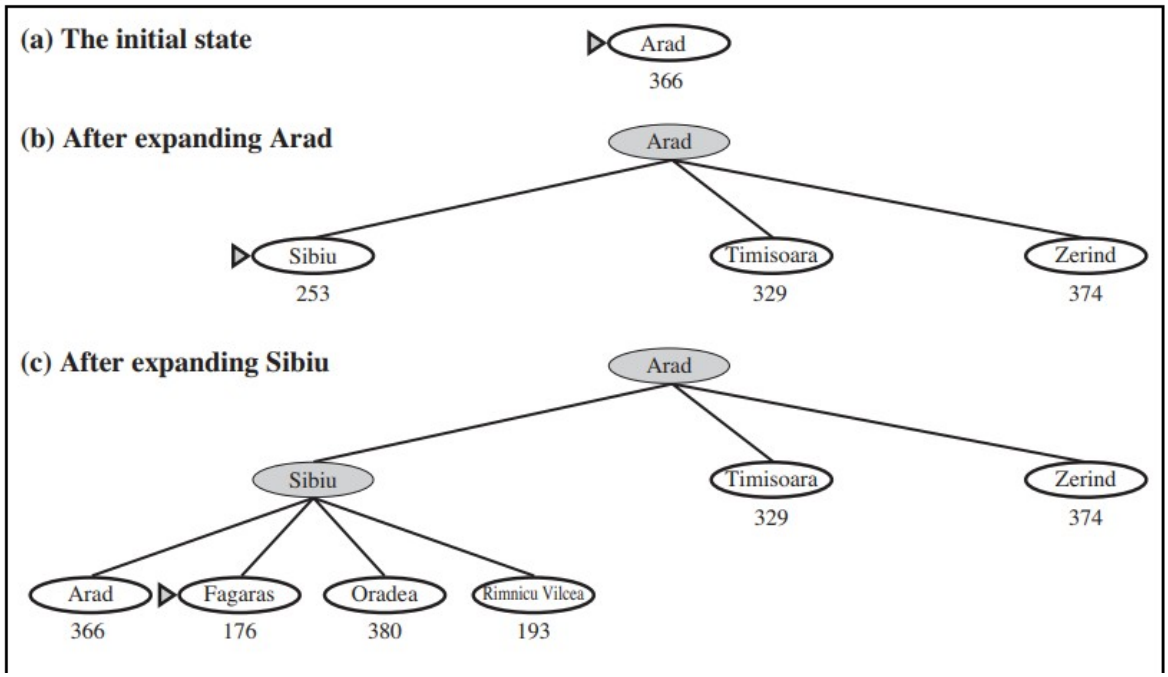


Figure 3.2 A simplified road map of part of Romania.

- 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.**
- **Greedy best-first tree search is also incomplete** even in a finite state space, much like depth-first search.
- Consider the problem of getting from **Iasi to Fagaras**. The heuristic suggests that **Neamt be expanded first** because it is closest to **Fagaras, but it is a dead end.**
- The solution is to **go first to Vaslui**—a step that is actually farther from the **goal according to the heuristic**—and then to continue to **Urziceni, Bucharest, and Fagaras**.
- The algorithm will never find this solution, however, because expanding Neamt puts Iasi back into the frontier, Iasi is closer to Fagaras than Vaslui is, and so Iasi will be expanded again, leading to an infinite loop. (The graph search version is complete in finite spaces, but not in infinite ones.)
- The worst-case time and space complexity for the tree version is O**)** 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.**



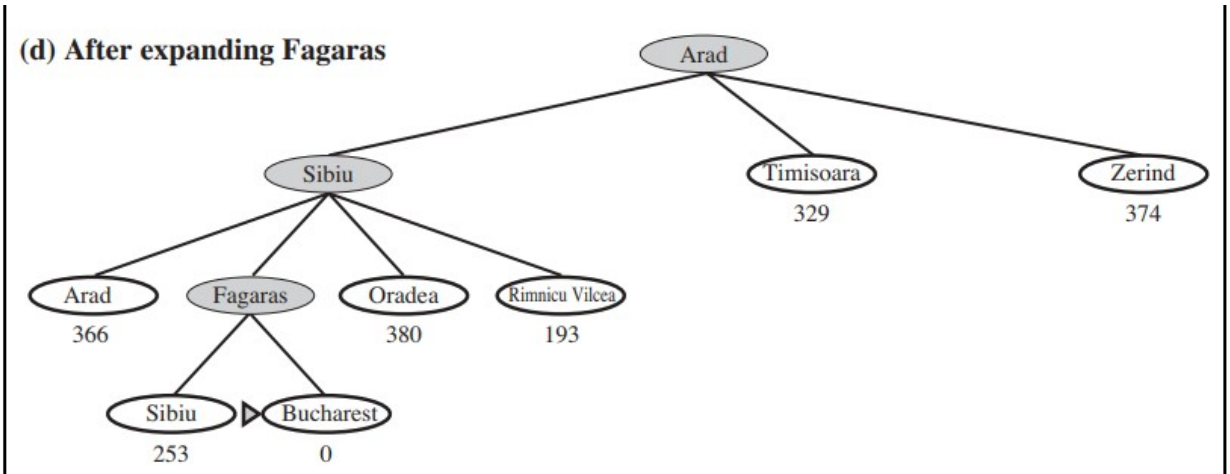
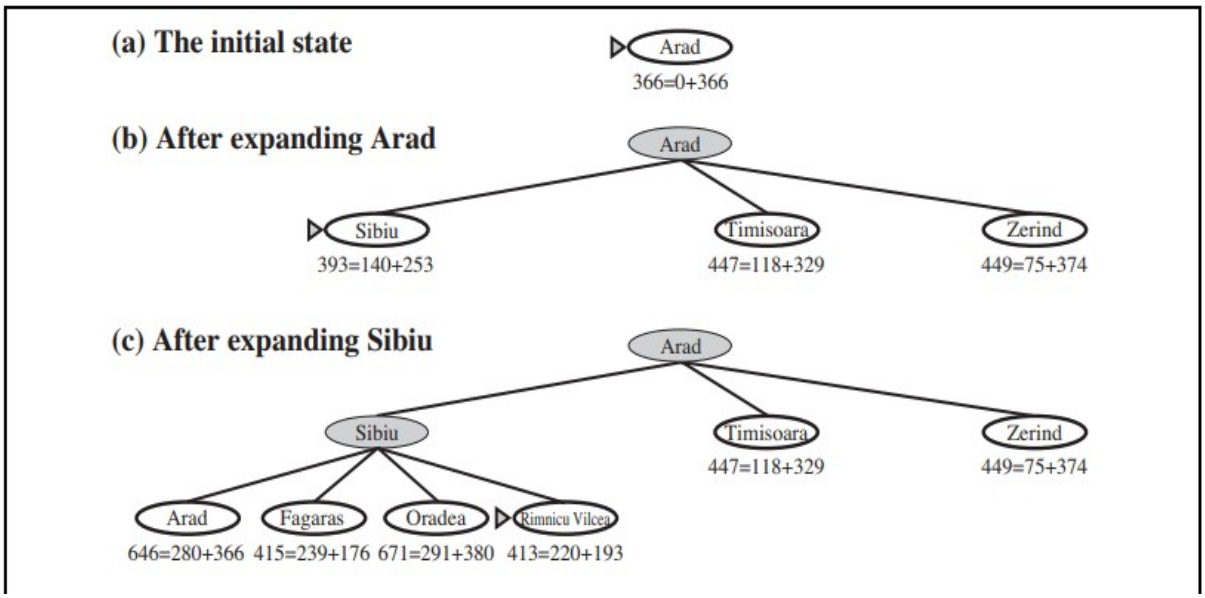


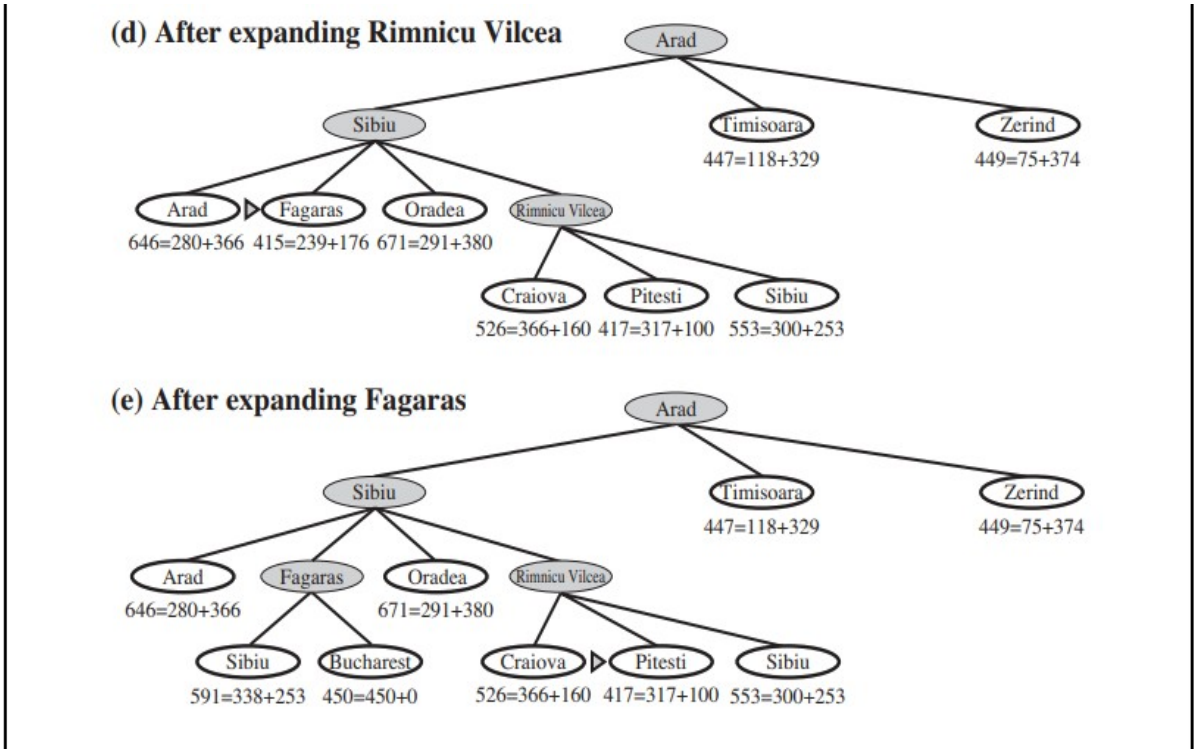
Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

A* search: Minimizing the total estimated solution cost

- The most widely known form of **best-first search** is called **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 UNIFORM-COST-SEARCH except that **A∗ uses g + h instead of g**.





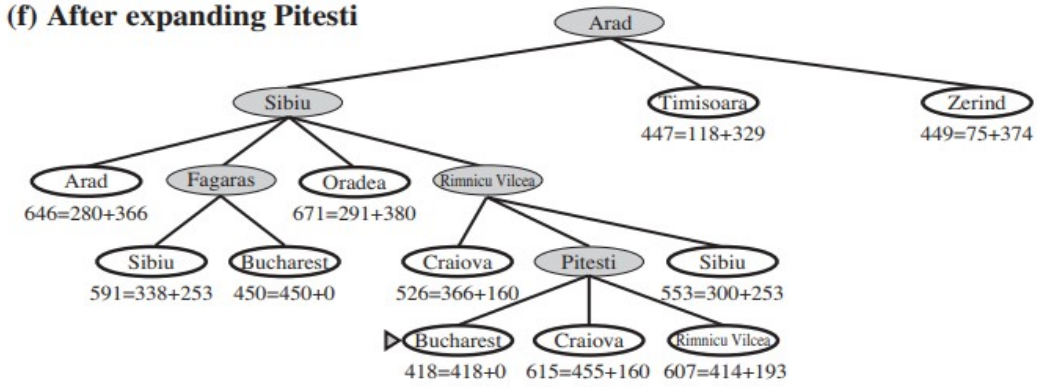


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

- **<u>Conditions for optimality: Admissibility and consistency</u>:**
- 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.
- 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 Gn 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.
- 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
- The next step is to prove that whenever **A∗ selects a node n for expansion, the optimal path to that node has been found.**

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n) .$$

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∗) algorithm.**
- 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.

The algorithm for recursive best-first search.

- **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 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 reexpansions 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**.

```
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.

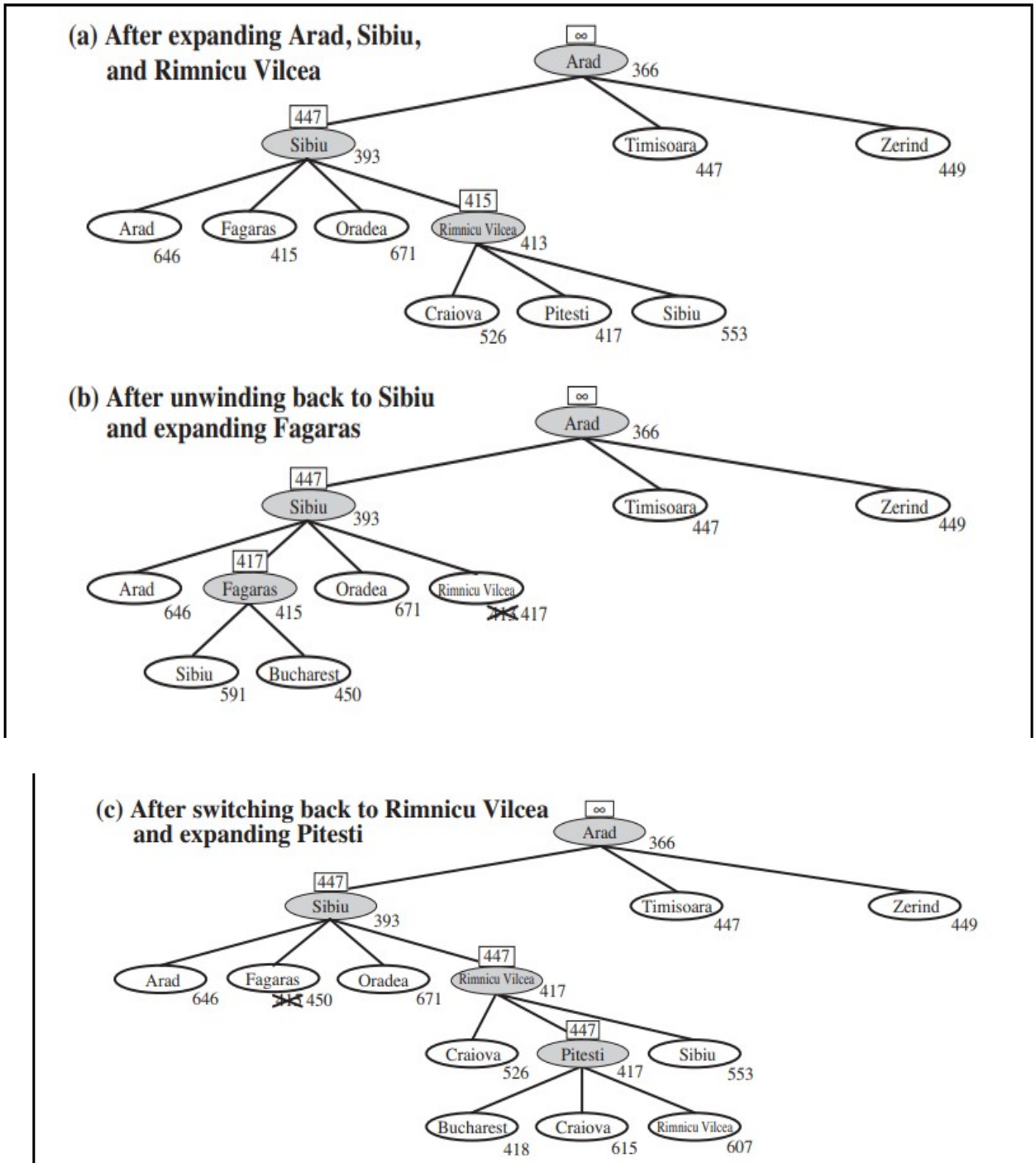


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse 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.

Heuristic Functions

- In this section, we look at heuristics for the 8-puzzle, in order to shed light on the nature of heuristics in general.
- The **8-puzzle was one of the earliest heuristic search problems**.
- As mentioned in Section 3.2, **the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration** (Figure 3.28).
- The average solution cost for a randomly generated **8-puzzle instance is about 22 steps.** The **branching factor is about 3.** (When the empty tile is in the middle, four moves are possible; when it is in a corner, two; and when it is along an edge, three.) This means that an exhaustive tree search to depth 22 would look at about $322 \approx 3.1 \times 10^{10}$ states.
- If we want to find the shortest solutions by using A$*$, we need a **heuristic function that never overestimates the number of steps to the goal.**
- There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:
- **h1 = the number of misplaced tiles.** For Figure 3.28, all of the eight tiles are out of position, so the start state would have h1 = 8. h1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.
- **h2 = the sum of the distances of the tiles from their goal positions**. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance.

- h2 is also admissible because all any move can do is move one tile one step closer to the goal.
- **Tiles 1 to 8 in the start state give a Manhattan distance of h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18** .
- As expected, neither of these overestimates the true solution cost, which is 26.

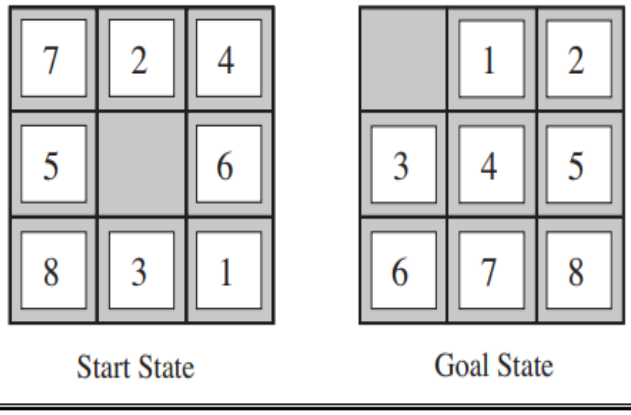


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

<u>The effect of heuristic accuracy on performance</u>:

- 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.**

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d .$$

- For example, if A∗ finds a solution at **depth 5 using 52 nodes, then the effective branching factor is 1.92.**

- To test the heuristic functions h1 and h2, we generated 1200 random problems with solution lengths from 2 to 24 (100 for each even number) and solved them with **iterative deepening search and with A∗ tree search** using both h1 and h2.
- Figure 3.29 gives the average number of nodes generated by each strategy and the effective branching factor.
- The results suggest that **h2 is better than h1**, and is far better than using **iterative deepening search.**
- Even for small problems with **d = 12, A∗ with h2 is 50,000 times more efficient than uninformed iterative deepening search.**
-

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

- One might ask whether h2 is always better than h1.

- The answer is “Essentially, yes.” It is easy to see from the definitions of the two heuristics that, for any node n, h2(n) ≥ h1(n).
- We thus say that h2 dominates h1.

Generating admissible heuristics from relaxed problem:

- We have seen that both **h1 (misplaced tiles) and h2 (Manhattan distance)** are fairly good heuristics for the 8-puzzle and that h2 is better.
- How might one have come up with h2? Is it possible for a computer to invent such a heuristic mechanically?
- **h1 and h2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for simplified versions of the puzzle.**
- If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then **h1 would give the exact number of steps in the shortest solution.**
- Similarly, if a tile could move one square in any direction, even onto an occupied square, then h2 would give the exact number of steps in the shortest solution.
- **A problem with fewer restrictions on the actions is called a relaxed problem.**
- For example, if the 8-puzzle actions are described as
- A tile can move from square A to square B if
- A is **horizontally or vertically adjacent to B and B is blank,**
- we can generate three relaxed problems by removing one or both of the conditions:

(a) A tile can move from square A to square B if A is adjacent to B.

(b) A tile can move from square A to square B if B is blank.

(c) A tile can move from square A to square B.

- From (a), we can derive h2 (Manhattan distance).
- From (c), we can derive h1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step.
- **Generating admissible heuristics from subproblems: Pattern databases:**

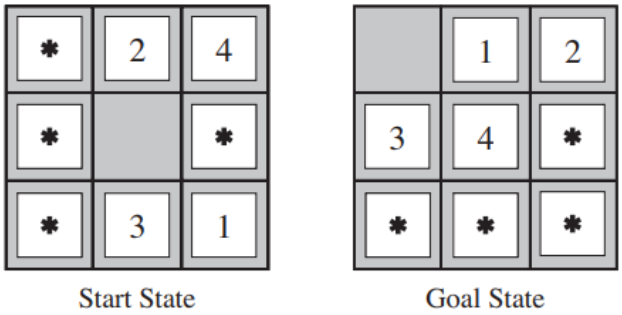


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

- 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.

Learning heuristics from experience

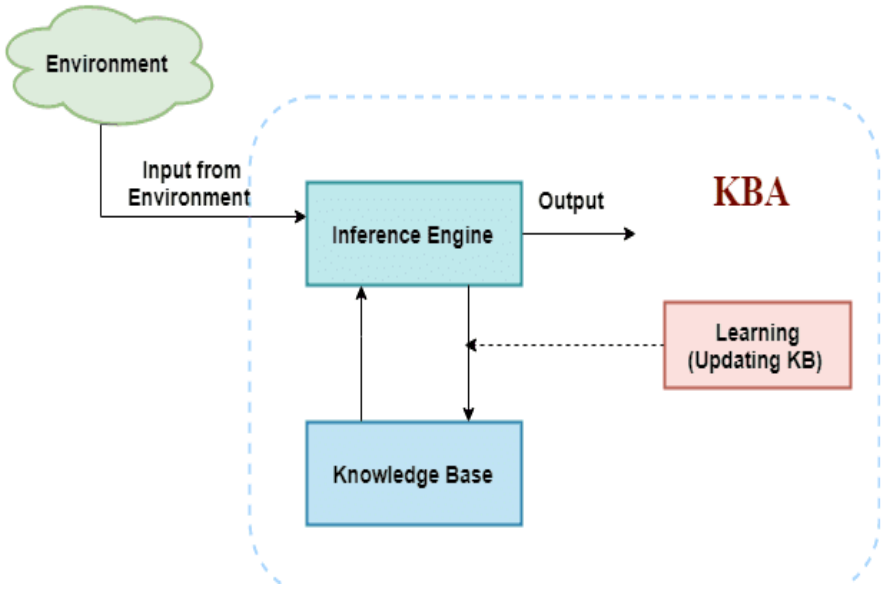
- "Experience" here means solving lots of 8-puzzles, for instance.
- Each optimal solution to an **8-puzzle problem provides examples from which h(n) can be learned.**
- Each example consists of a **state from the solution path and the actual cost of the solution from that point.**
- From these examples, a **learning algorithm can be used to construct a function h(n) that can (with luck) predict solution costs for other states that arise during search.**
- Techniques for doing just this using **neural nets, decision trees, and other methods.**
- Inductive learning methods work best when supplied with features of a state that are relevant to predicting the state's value, rather than with just the raw state description.
- How should $\mathbf{x_1(n)}$ **and** $\mathbf{x_2(n)}$ be combined to predict h(n)?
- **A common approach is to use a linear combination:**
- $\mathbf{h(n) = c_1x_1(n) + c_2x_2(n)}$ **.**

Chapter 7 : Logical Agents

- Humans, it seems, know things; and what they know helps them do things.
- These are not empty statements. **They make strong claims about how the intelligence of humans is achieved**—not by purely reflex mechanisms but by processes of reasoning that operate on **internal representations of knowledge.**
- In AI, this approach to **intelligence is embodied in knowledge-based agents.**

7.1 Knowledge Based Agents

- The central component of a knowledge-based agent is its **knowledge base, or KB.**
- A knowledge base is a **set of sentences**. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.)
- **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.



- 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**.
- The agent in Figure 7.1 appears quite similar to the agents with internal state described in Chapter 2.
- Because of the definitions of **TELL and ASK, however, the knowledge-based agent** is not an arbitrary program for calculating actions.
- It is amenable to a **description at the knowledge level**, where we need specify only what the **agent knows** and **what its goals** are, in order to fix its behavior.
- For example, an **automated taxi might have the goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge** is the only link between the two locations.
- Then we can expect it to **cross the Golden Gate Bridge** because it knows that that will **achieve its goal**.
- Notice that this analysis is **independent of how the taxi works at the implementation level.**
- It doesn’t matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by

manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

- A knowledge-based agent can be built simply by **TELLing it what it needs to know.**
- Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment.
- **This is called the declarative approach** to system building. In contrast, the procedural approach encodes desired **behaviors directly as program code.**
- In the 1970s and 1980s, advocates of the two approaches engaged in heated debates.
- We now understand that a **successful agent** often combines both **declarative and procedural elements** in its design, and that **declarative knowledge** can often be compiled into more efficient procedural code.
- We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself.
- A learning agent can be fully autonomous.

```
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.

7.2 THE WUMPUS WORLD

Environment in which k**nowledge-based agents** can show their worth.

- 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 bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, 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. 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◦,orTurn Right by 90◦.** The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.)
- 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 any where in the cave.
- the agent program will get **[Stench,Breeze, None,None,None].**

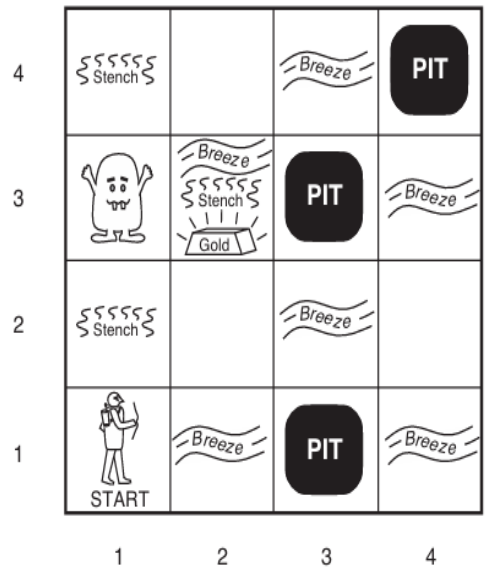


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

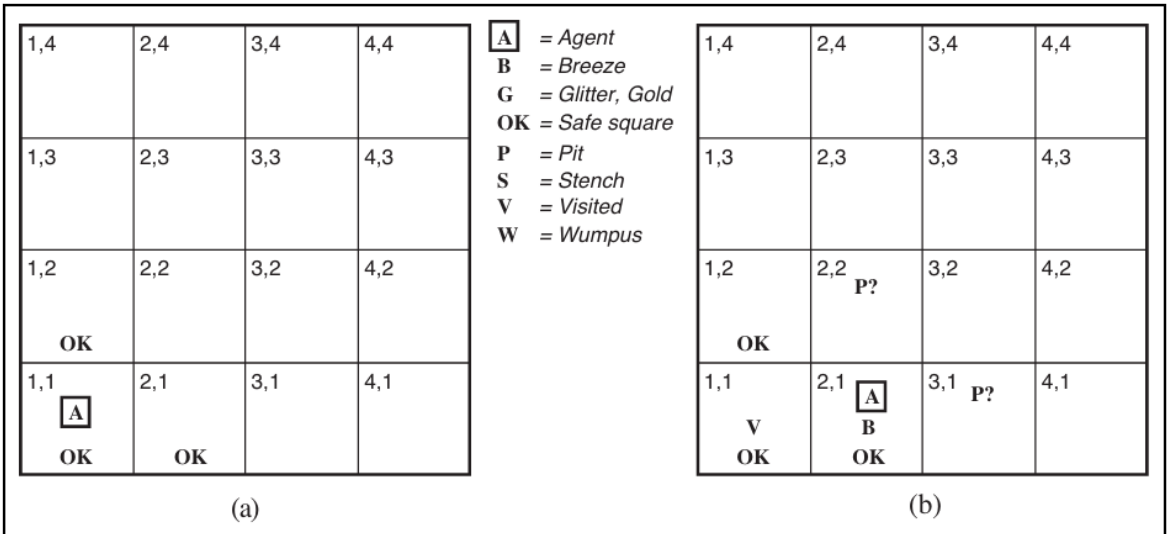


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept $[None, None, None, None, None]$. (b) After one move, with percept $[None, Breeze, None, None, None]$.

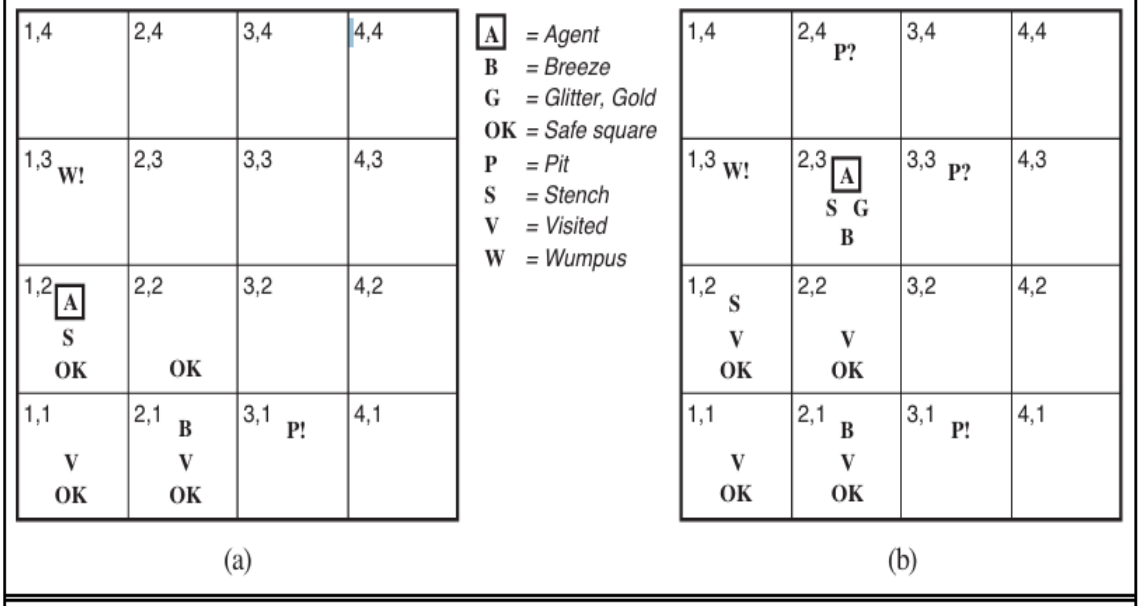


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

7.3 Logic

- we said that **knowledge bases consist of sentences.** These sentences are expressed according to 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.

- When we need to be precise, we use the **term model** in place of "**possible world**."
- Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, **models** are **mathematical abstractions**, each of which simply fixes the **truth** or **falsehood** of every **relevant sentence.**
- **Informally, we may think of a possible world as, for example, having x men and y women sitting at a table playing bridge, and the sentence x + y = 4 is true when there are four people in total.**
- Formally, the **possible models** are just all **possible assignments** of real numbers to the **variables x and y**.
- Each such assignment fixes the truth of any sentence of arithmetic whose variables are x and y.
- **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.
- **Logical consequence (also entailment**) is a fundamental concept in **logic** which describes the relationship between statements that hold true when one statement logically ***follows from*** **one or more statements.**
- **In mathematical notation, we write α |= β**

- to mean that the **sentence α entails the sentence β.**
- The formal definition of entailment is this: **α |= β** if and only if, in **every model in which α is true, β is also true.**
- Using the notation just introduced, we can write α |= β if and only **if M(α) ⊆ M(β) .**
- (Note the direction of the **⊆ here: if α |= β, then α is a stronger assertion than β: it rules out more possible worlds.)**
- The relation of entailment is familiar from arithmetic; we are happy with the idea that the sentence x = 0 entails the sentence xy = 0.
- **Obviously, in any model where x is zero, it is the case that xy is zero** (regardless of the value of y). We can apply the same kind of analysis to the wumpus-world reasoning example given in the preceding section.
- Consider the situation in Figure 7.3(b): **the agent has detected nothing in [1,1] and a breeze in [2,1].**
- These percepts, combined with the **agent's knowledge of the rules of the wumpus world, constitute the KB.**
- The **agent is interested (among other things)** in whether the adjacent squares **[1,2], [2,2], and [3,1]** contain pits.
- Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are 2^3**= 8 possible models.**
- **These eight models are shown in Figure 7.5**

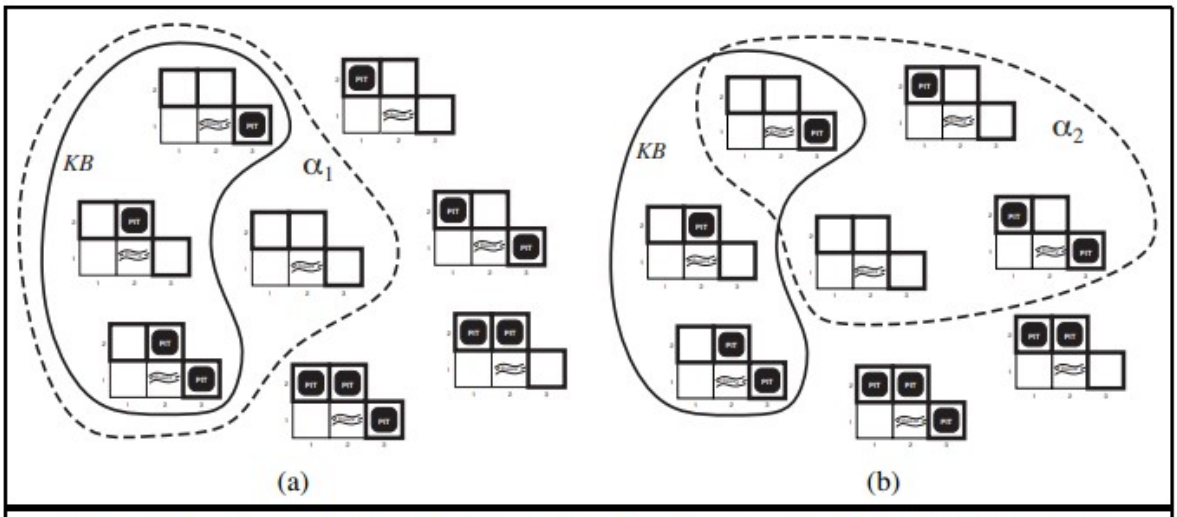


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in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

- The KB can be thought of as a set of sentences or as a single sentence that asserts all the individual sentences.
- The **KB is false in models that contradict what the agent knows**— for example, the KB is false in any model in which **[1,2] contains a pit, because there is no breeze in [1,1].**
- There are in fact just three models in which the KB is true, and these are shown surrounded by a solid line in Figure 7.5.
- **Now let us consider two possible conclusions:**
- **α1 = "There is no pit in [1,2]."**
- **α2 = "There is no pit in [2,2]."**
- We have surrounded the models of **α1 and α2** with dotted lines in Figures 7.5(a) and 7.5(b), respectively. By inspection, we see the following:
- **in every model in which KB is true, α1 is also true. Hence, KB |= α1: there is no pit in [1,2].**

- We can also see that in some models in which **KB is true, α2 is false. Hence, KB |= α2:** the agent **cannot conclude that there is no pit in [2,2].**

  (Nor can it conclude that there is a 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 is, that 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
- The final issue to consider is **grounding—the connection between logical reasoning processes and the real environment in which the agent exists.**

- In particular, how do we know that KB is true in the real world? (After all, KB is just “syntax” inside the agent’s head.)
- **A simple answer is that the agent’s sensors create the connection. For example, our wumpus-world agent has a smell sensor.**
- The **agent program creates a suitable sentence whenever there is a smell.**
- Then, whenever that sentence is in the knowledge base, it is true in the real world.
- Thus, the **meaning and truth of percept sentences are defined by the processes of sensing and sentence construction that produce them.**
- What about the rest of the agent’s knowledge, such as its belief that wumpuses **cause smells in adjacent squares**? This is not a direct representation of a single percept, but a general rule—derived, perhaps, from **perceptual experience** but not identical to a statement of that experience.
- General rules like this are **produced by a sentence construction process called learning.**

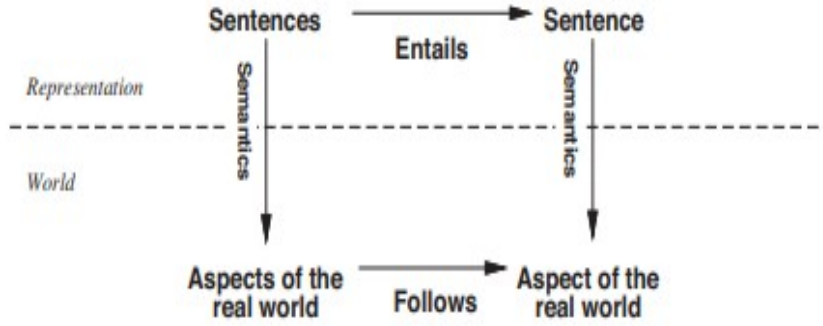


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations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

7.4 Propositional Logic: A simple Logic

- We now present a simple **but powerful logic** called **propositional logic.** We cover the **syntax of propositional logic and its semantics**—the way in which the **truth of sentences is determined**.
- Then we look at entailment—the relation between a sentence and another sentence that follows from it—and see how this leads to a **simple algorithm for logical inference**.
- **Everything takes place, of course, in the wumpus world.**

7.4.1 Syntax

- The syntax of 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:
- In logic and analytic philosophy, an **atomic sentence is a type of declarative sentence which is either true or false** (may also be referred to as a **proposition, statement or truthbearer**) and which cannot be broken down **into other simpler sentences.**
- For example, **"The dog ran"** is an **atomic sentence** in natural language, whereas "**The dog ran and the cat hid" is a molecular sentence in natural language.**
- **NEGATION** ¬ (not). A sentence such as **¬W1,3** is called the negation of W1,3.
- **LITERAL** : 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 conjunction; 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. (Historically, the ∨ comes from the Latin "vel," which means "or." For most people, it is easier to remember ∨ as an upside-down ∧.)
- **IMPLICATION** ⇒ (implies). A sentence such as (W1,3∧P3,1) ⇒ ¬W2,2 is called an **implication (or conditional).**
- 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. Some other books write this as ≡.

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \mid [\textit{Sentence}] \\
&\mid \neg \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
\text{OPERATOR PRECEDENCE} &: \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

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

7.4.2 Semantics

- Having specified the syntax of propositional logic, we now specify its 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}** . With three proposition symbols, there are 2^3**= 8** possible models—exactly those depicted in Figure 7.5.
- 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 **sub sentences 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*.

- The truth tables for **"and," "or," and "not"** are in close accord with our intuitions about the English words.

- The main point of possible confusion is that P ∨ Q is true when P is true or Q is true or both. A different connective, called "exclusive or" ("xor" for short), yields false when both disjuncts are true.
- There is no consensus on the symbol for exclusive or; some choices are $\vee^{\cdot}$ or = or ⊕.
- The truth **table for ⇒** may not quite fit one's intuitive understanding of "**P implies Q" or "if P then Q."**
- For one thing, propositional logic does not require any relation of causation or relevance between P and Q.

The biconditional, P ⇔ Q, is true whenever both **P ⇒ Q and Q ⇒ P are true. I**n English, this is often written as **"P if and only if Q."**

- Another point of confusion is that any implication is true whenever its antecedent is false. For example, "5 is even implies Sam is smart" is true, regardless of whether Sam is smart.
- This seems bizarre, but it makes sense if you think of "P ⇒ Q" as saying, "If P is true, then I am claiming that Q is true.
- Otherwise I am making no claim." The only way for this sentence to be false is if P is true but Q is false.
- Many of the rules of the **wumpus world are best written using ⇔. For example, a square is breezy if a neighboring square has a pit, and a square is breezy only if a neighboring square has a pit.**
- So we need a biconditional, **B1,1 ⇔ (P1,2 ∨ P2,1) , where B1,1 means that there is a breeze in [1,1].**