Module 3

3.5 INFORMED SEARCH STRATEGIES

Informed search strategy is 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 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 except for the use of f instead of g to order the priority queue.

The choice of f determines the search strategy. Most best-first algorithms include as a component of a **heuristic function**, denoted h(n): h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.

• A search strategy that uses problem-specific knowledge can find solutions more efficientlythan an uninformed strategy. We use best-first search as general approach. A node is selected for expansion based on an evaluation function $f(n)$.

• Informed search algorithms include a heuristic function $h(n)$ as part of $f(n)$.

• Often $f(n) = g(n) + h(n)$

• $h(n)$ = estimate of the cheapest cost from the state at node n to a goal state; $h(goal) = 0$.

3.5.1 Greedy best-first search

Greedy best-first search is a form of best-first search that expands first the node with the Greedy best-first Search lowest $h(n)$ value−the node that appears to be closest to the goal−on the grounds that this is likely to lead to a solution quickly. So, the evaluation function $f(n) = h(n)$. Let us see how this works for route finding problems in Romania; we use the straight line distance heuristic, which we will call *hSLD*. If the goal is Bucharest, we need to know Straight-line distance the straight-line distances to Bucharest, which are shown in Figure 3.22. For example, $hSLD(Arad)$=366. Notice that the values of *hSLD* cannot be computed from the problem description itself (that is, the ACTIONS and RESULT functions). Moreover, it takes a certain amount of world knowledge to know that *hSLD* 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 *hSLD* to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because the heuristic says it is closer to Bucharest than is either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is now closest according to the heuristic. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using *hSLD* finds a solution without ever expanding a node that is not on the solution path. The solution it found does not have optimal cost, however: the path via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti. This is why the algorithm is called "greedy"−on each iteration it tries to get as close to a goal as it can, but greediness can lead to worse results than being careful. Greedy best-first graph search is complete in finite state spaces, but not in infinite ones. The worst-case time and space complexity is $O(|V|)$. With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching $O(bm)$.

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.

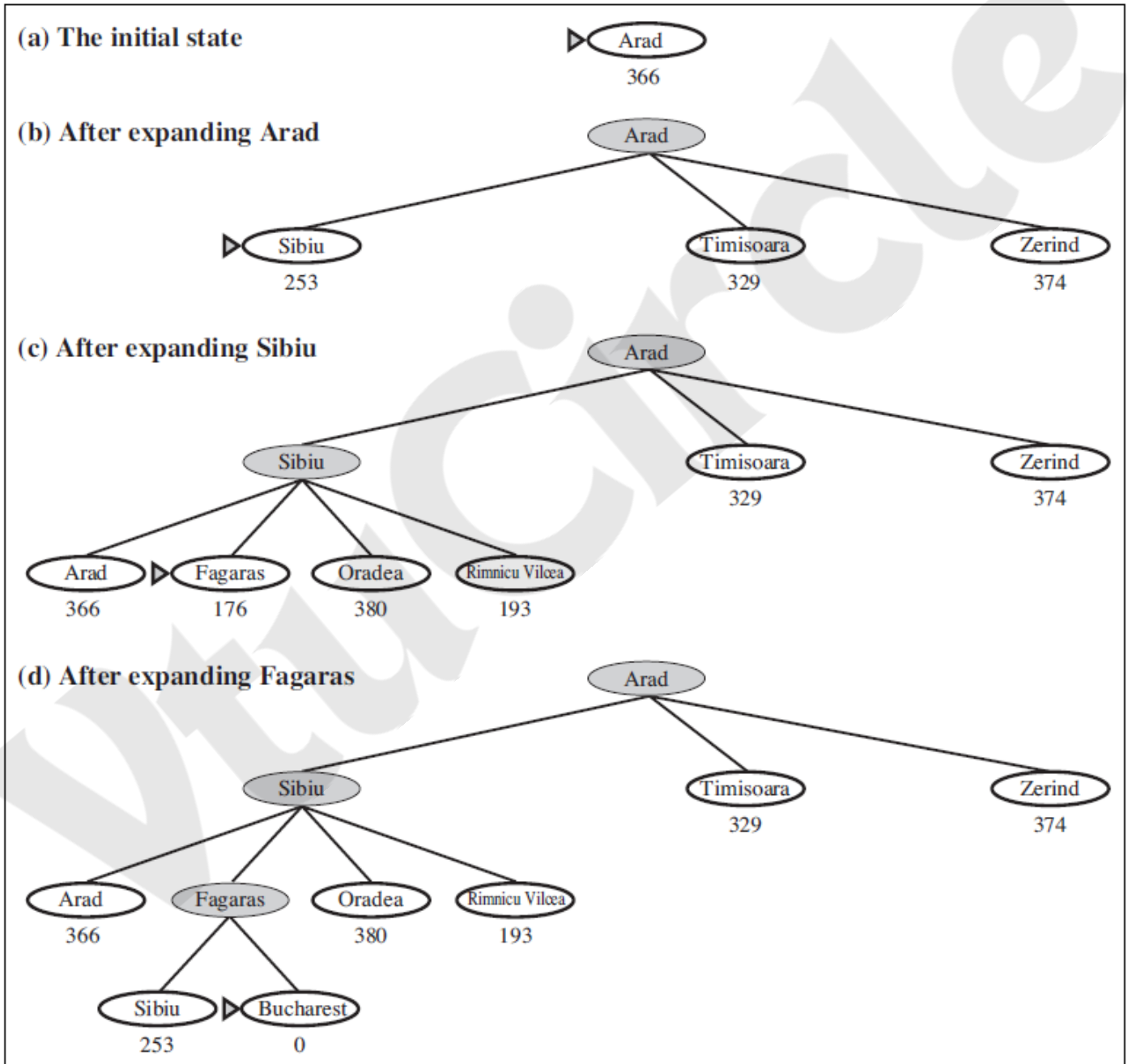


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

3.5.2 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

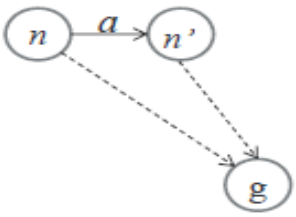
$$f(n) = \text{ estimated cost of the cheapest solution through } n .$$

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.

Conditions for optimality: Admissibility and consistency- Optimality of A*

In order for A* to be optimal

- $h(n)$ must be admissible, i.e. it never overestimates the cost to reach the goal.
- Then, as a consequence, f(n) = $g(n)$ + $h(n)$ never overestimates the true cost of a solution along the current path through n.
- $h(n)$ must be consistent (monotonic) in graph search, i.e. for every node n and every successor n' of n generated by action a, This is a form of the triangle inequality.
- Every consistent heuristic is also admissible.



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

•The graph-search version of A* is optimal if $h(n)$ is consistent. We show this in 2 steps:

1. If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing Optimality of A* (Graph Search) nondecreasing.

- Suppose n' is a succ. of n, then $g(n') = g(n) + c(n, a, n')$
- Therefore $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n')$

$$\ge g(n) + h(n) = f(n)$$

2. Whenever A* selects a node n for expansion, the optimal path to that node has been found. otherwise there must be another frontier node n' on the optimal path from start node to n; as f is nondecreasing along any path, n' would have lower f-cost than n and would have been selected first.

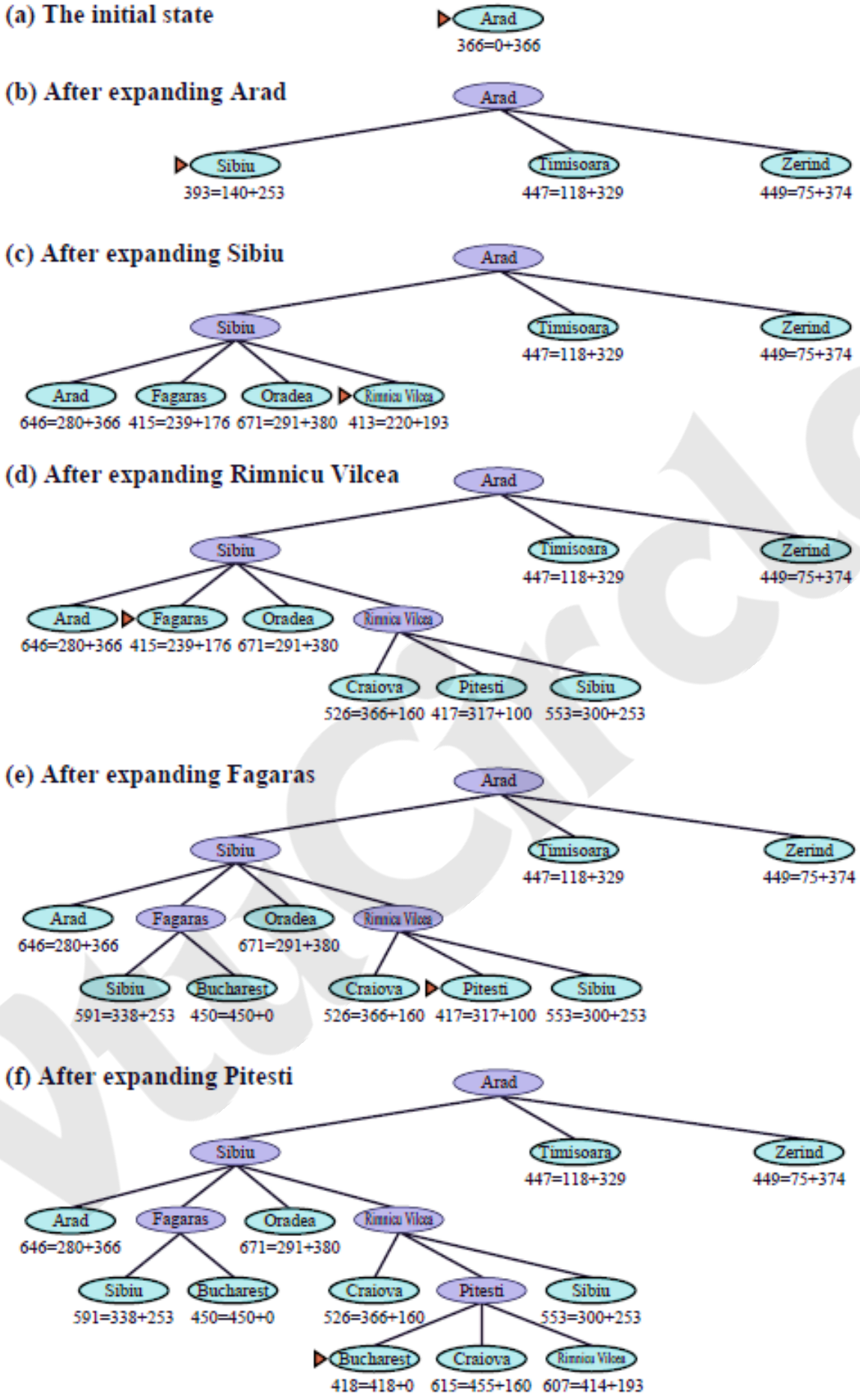


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

Contours in State Space:

• As *f*-costs are nondecreasing along any path we can draw contours in the state space.

• See figure 3.25 on the following page

• With uniform-cost search (A* with $h(n) = 0$), contours are circular around the start state.

• With more accurate heuristics, the bands will stretch towards the goal state and become more narrow

• If C^* is the cost of the optimal solution path, then

  • A* expands all nodes with $f(n) < C^*$

  • A* may expand some nodes with $f(n) = C^*$ before finding the goal state.

• Completeness requires that there are only finitely many nodes with cost less than or equal to C^*, which is true if all step costs exceed some finite ε and if b is finite.

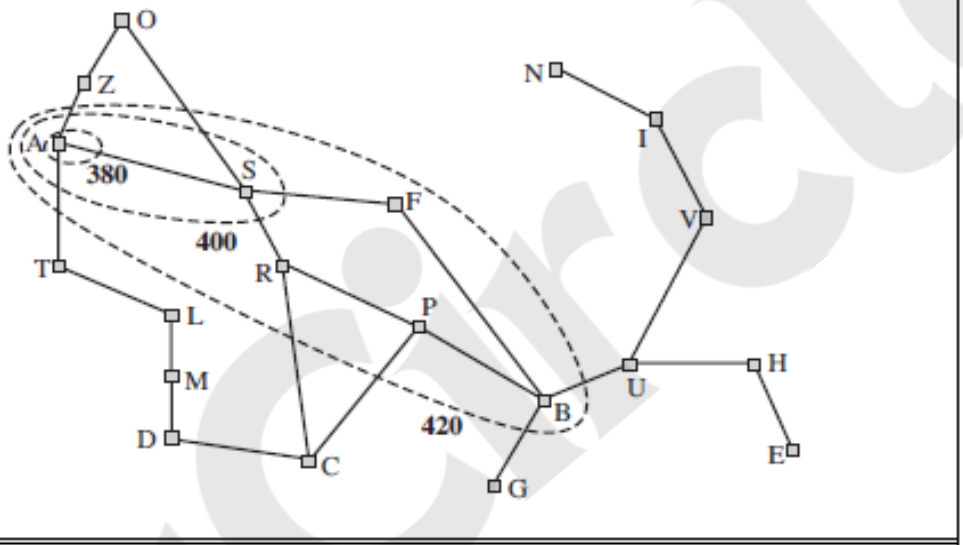


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

A* is Optimally Efficient

• A* is optimally efficient for any given consistent heuristic, i.e. no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly for nodes with f(n) = C*).

  • This is because any algorithm that does not expand all nodes with f(n) < C* may miss the optimal solution.

• So A* is complete, optimal and optimally efficient, but it still is not the solution to all search problems:

• As A* keeps all generated solutions in memory it runs out of space long before it runs out of time.

3.5.3 Memory-bounded Heuristic Search

How to reduce memory requirements of A*?

  • Use iterative deepening A* (IDA*) .

• Cutoff used is f(n) = g(n) + h(n) 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.

• This works well with unit step costs.

• It suffers from severe problems with real-valued costs.

=>Recursive Best-First Search (RBFS)

•Simple recursive algorithm that tries to mimic the operation of standard best-first search in linear space .

• It is similar to recursive DFS, but uses a variable *f_limit* to keep track of the best alternative path available from any ancestor of the current node.

• If the current node exceeds *f_limit*, the recursion unwinds back to the alternative path. Then, RBFS replaces the *f*-value of each node on the path with a backed-up value, the best value of its children.

• In this way, RBFS remembers the *f*-value of the best leaf in the forgotten subtree and may decide to re-expand it later.

```
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.

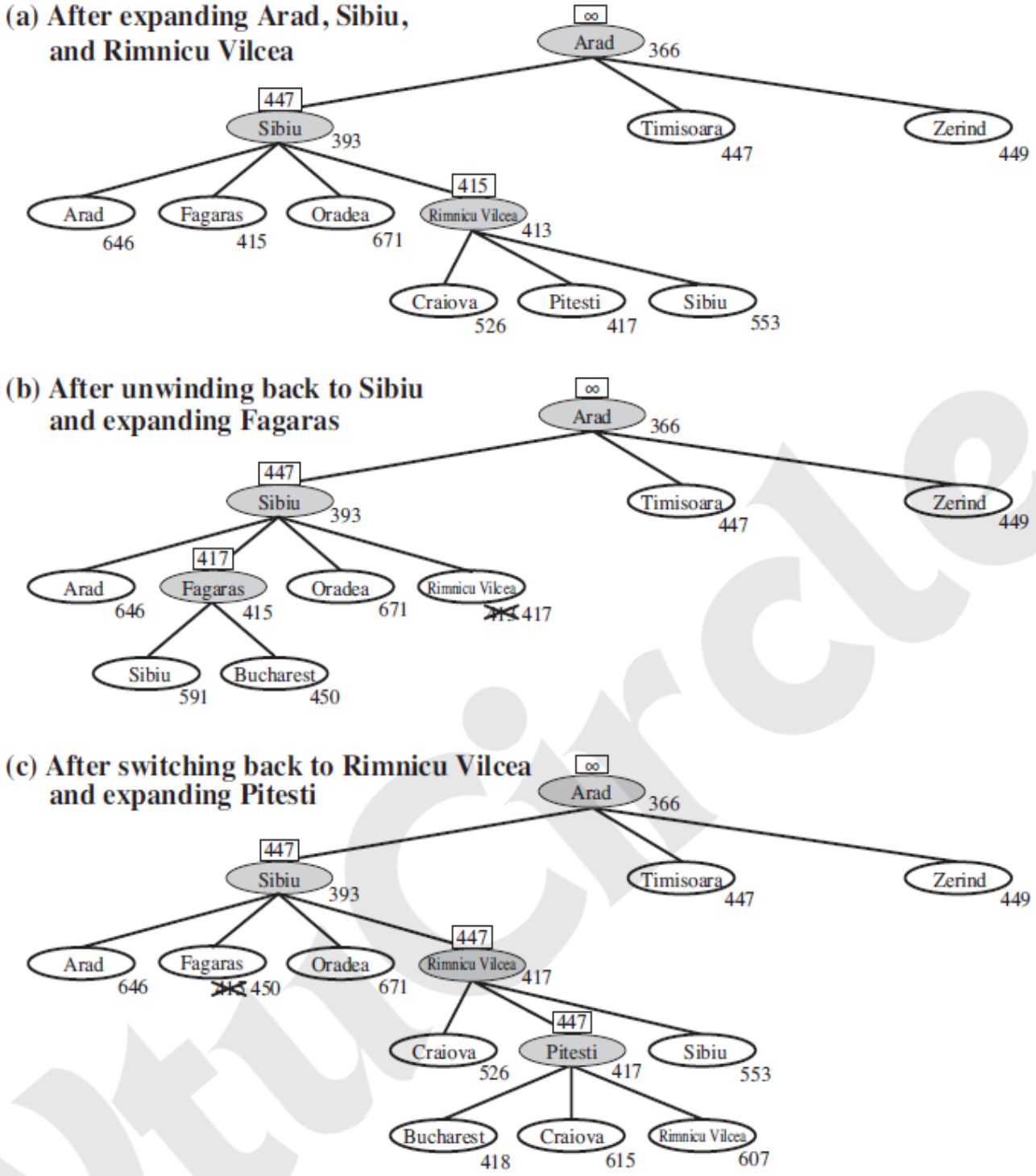


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse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

Simplified Memory-bounded A* (SMA*)

•IDA* and RBFS use too little memory

- IDA* retains only 1 number between iterations (*f*-cost limit)
- RBFS retains more information, but uses only linear space

• SMA* proceeds just like A*, expanding the best leaf until memory is full. Then it must drop an old node.

• SMA* always drops the worst leaf node (highest *f*value). (In case of ties with same *f*-value, SMA* expands the newest leaf and deletes the oldest leaf)

• 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.

3.6 Heuristic Functions

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 $322 \approx 3.1 \times 1010$ states.

A graph search would cut this down by a factor of about 170,000 because only 9!/2 =181, 440 distinct states are reachable. (See Exercise 3.4.) This is a manageable number, but the corresponding number for the 15-puzzle is roughly 1013, so the next order of business is to find a good heuristic function. 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.

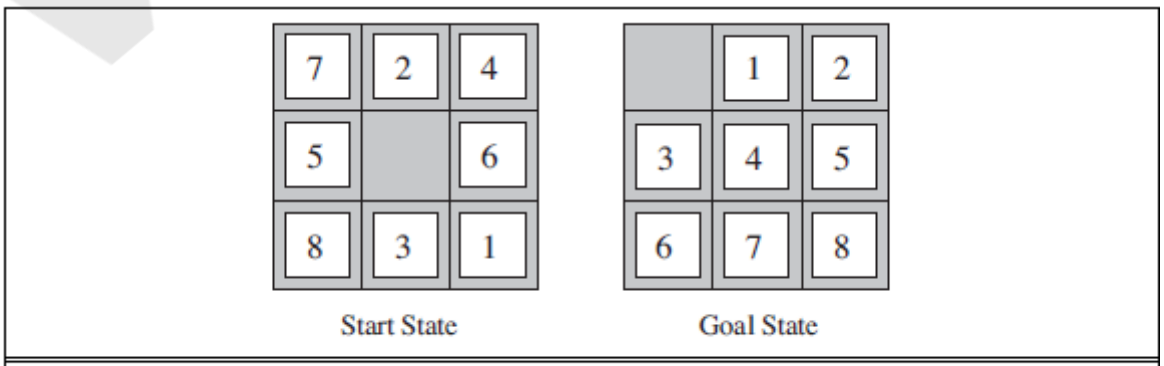


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

Heuristic Functions:

The quality of any heuristic search algorithm depends on its heuristic

• Two admissible heuristics for the 8-puzzle:

  • h1 = # misplaced tiles

  • h2 = sum of Manhattan dist. of all tiles to goal position

  • h2 is better for search than h1.

• The effective branching factor b^* may characterize the quality of a heuristics:

• If N is the # nodes generated by a search algorithm and d is the solution depth, then b^* ist the branching factor, which a uniform tree of depth d would need to have to contain $N+1$ nodes. Thus

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

• The performance of heuristic search algorithms depends on the quality of their heuristic function.

• Methods to construct good heuristics are: relaxing the problem definition, using a pattern DB with precomputed solution costs, or automated learning from experience.

7.1 KNOWLEDGE-BASED AGENTS

Logical agents are always definite. each proposition is either true/false or unknown (agnostic).

Knowledge representation language - a language used to express knowledge about the world.

- Declarative approach - language is designed to be able to easily express knowledge for the world the language is being implemented for.
- Procedural approach - encodes desired behaviours directly in program code.
- Sentence - a statement expressing a truth about the world in the knowledge representation language

Knowledge Base (KB) - a set of sentences describing the world

- Background knowledge - initial knowledge in KB
- Knowledge level -we only need to specify what the agent knows and what its goals are in order to specify its behaviour
- Tell(P) - function that adds knowledge P to the KB
- Ask(P) - function that queries the agent about the truth of P.
- Inference -the process of deriving new sentences from the knowledge base
  - ✓ *When the agent draws a conclusion from available information, it is guaranteed to be correct if the available information is correct.*

```
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

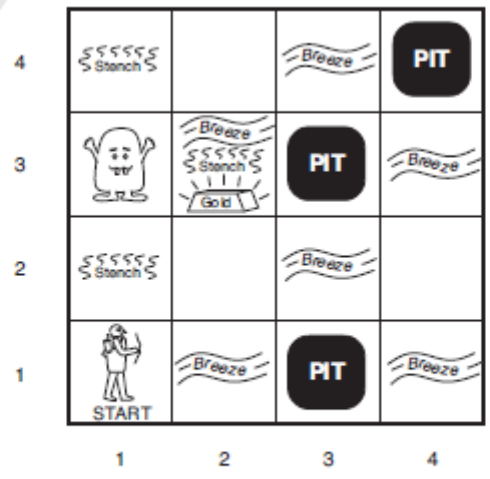


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

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 labelled [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.
    - Squares adjacent to wumpus are smelly
    - Squares adjacent to pit are breezy.
    - Glitter iff gold is in the same square.
    - Shooting kills wumpus if you are facing it.
    - Shooting uses up the only arrow.
    - Grabbing picks up gold if in same square.
    - Releasing drops the gold in same square.
- **Actuators:**
    - Turn Left (90°),
    - Turn Right (90°),
    - Forward,
    - Grab (gold),
    - Shoot (arrow),
    - Climb (at 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.
    - Stench, Breeze, Glitter, Bump, Scream.
- Observable? No . only local perception
- Deterministic? Yes . outcomes exactly specified
- Episodic? No . sequential at the level of actions
- Static? Yes . Wumpus and Pits do not move
- Discrete? Yes
- Single-agent? Yes . Wumpus is essentially a natural feature.

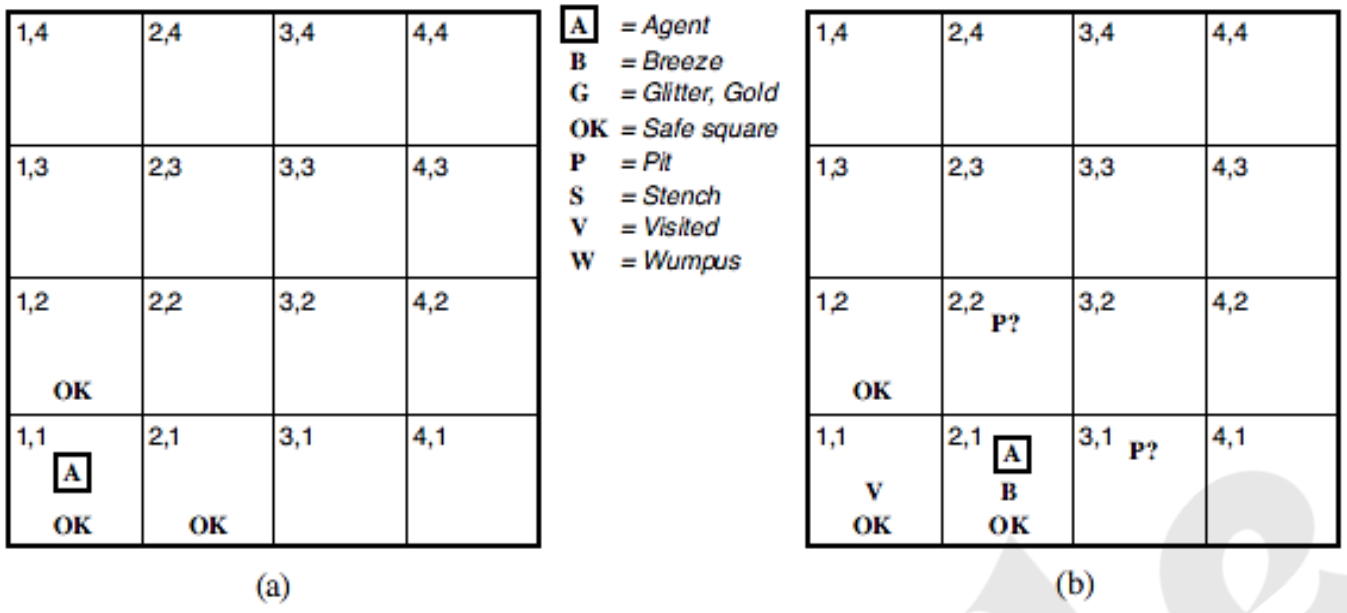


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After one move, with percept [*None*, *Breeze*, *None*, *None*, *None*].

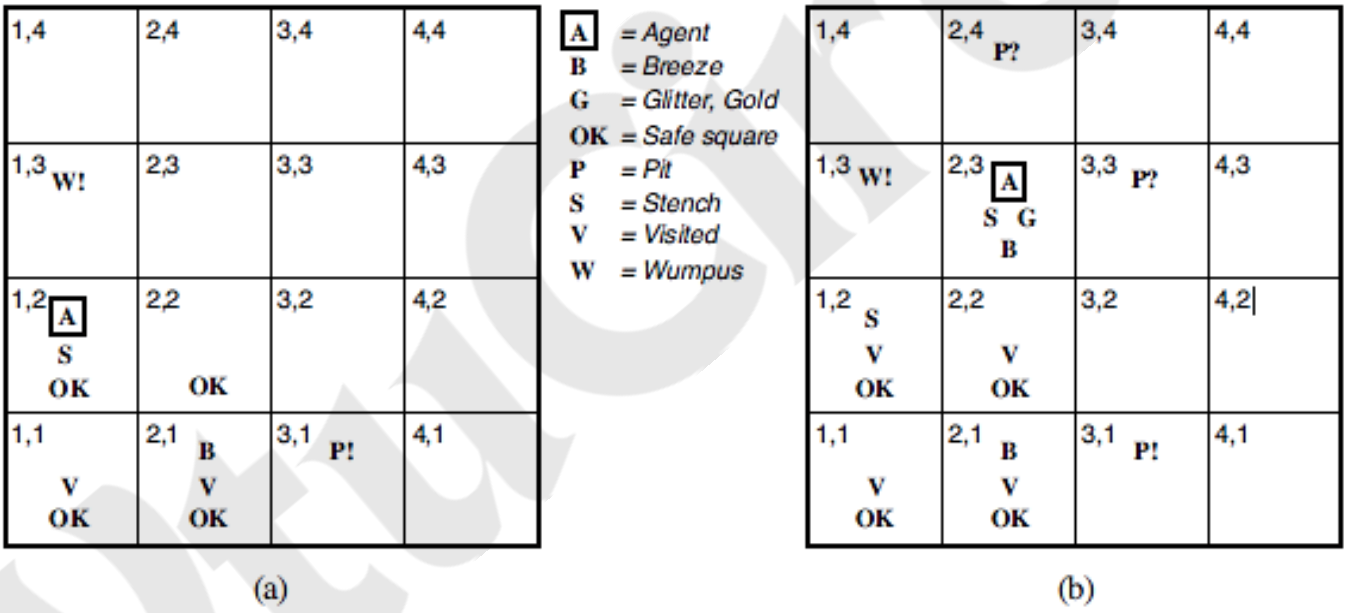


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

7.3 LOGIC

Logics - formal languages for representing information such that conclusions can be drawn

- Syntax -description of a representative language in terms of well-formed sentences of the language
- Semantics -defines the meaning (truth) of a sentence in the representative language w.r.t. each possible world
- Model - the world being described by a KB
- Satisfaction - model *m* satifies a sentence α, if α is true in *m*.
- Entailment - the concept that a sentence follows from another sentence:
  - $\alpha \models \beta$ if α is true then β must also be true
- Logical inference - the process of using entailment to derive conclusions
- Model checking - enumeration of all possible models to ensure that a sentence α is true in all models in which KB is true
- $M(\alpha)$ is the set of all models of α.
- Using the notation just introduced, we can write

$$\alpha \models \beta \text{ if and only if } M(\alpha) \subseteq M(\beta)$$

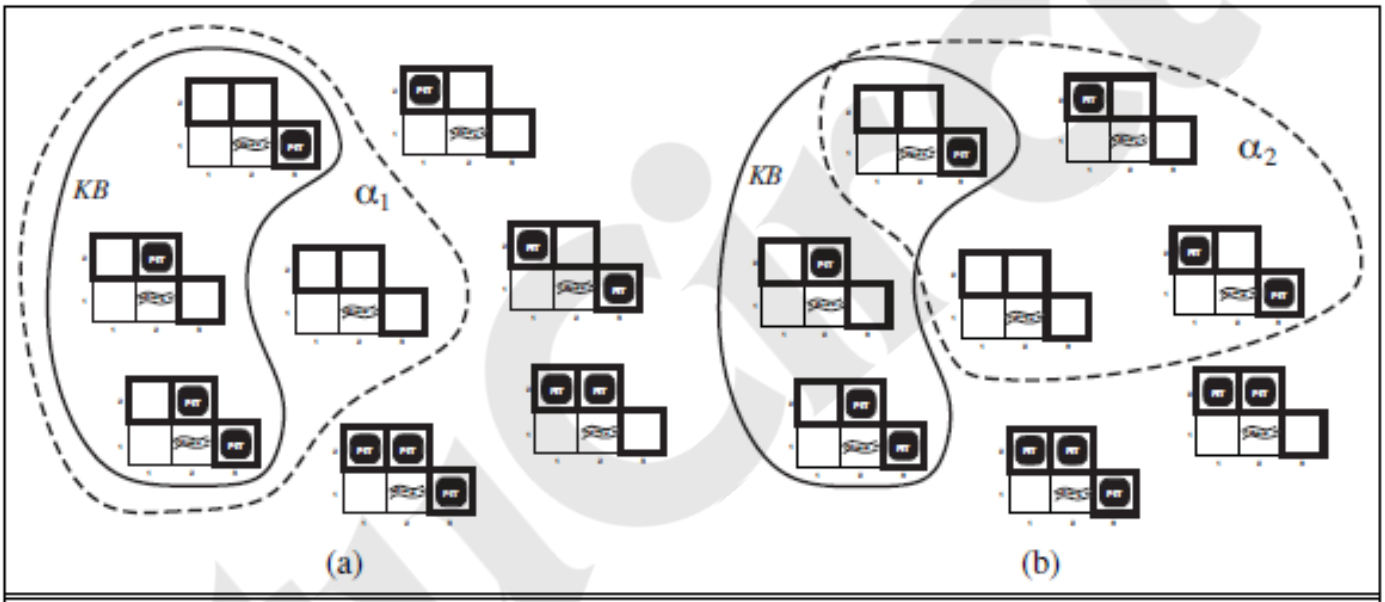


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in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

The 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.

α_1 = "There is no pit in [1,2]."
α_2 = "There is no pit in [2,2]."

We have surrounded the models of α_1 and α_2 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, α_1 is also true.

Hence, $KB \models \alpha_1$: there is no pit in [1,2]. We can also see that

in some models in which KB is true, α_2 is false.

Hence, $KB \not\models \alpha_2$: the agent *cannot* conclude that there is no pit in [2,2]. (Nor can it conclude that there *is* a pit in [2,2].)[3]

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 $M(KB) \subseteq M(\alpha)$.

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

$$KB \vdash_i \alpha ,$$

which is pronounced "α is derived from KB by i" or "i derives α from KB."

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 case6 by virtue of other aspects of the real world being the case. This correspondence between world and representation is illustrated in Figure 7.6.

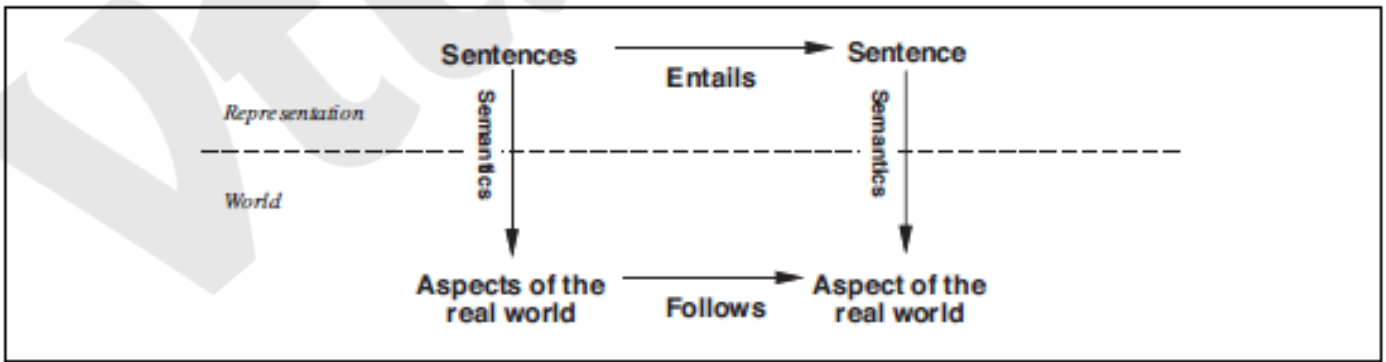


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations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

7.4 PROPOSITIONAL LOGIC: A VERY SIMPLE LOGIC

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, $W_{1,3}$ and $North$. The names are arbitrary but are often chosen to have some mnemonic value—we use $W_{1,3}$ to stand for the proposition that the wumpus is in [1,3]. (Remember that symbols such as $W_{1,3}$ 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:

- $\neg$ (not). A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).
- $\wedge$ (and). A sentence whose main connective is $\wedge$, such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction**; its parts are the **conjuncts**. (The $\wedge$ looks like an "A" for "And.")
- $\vee$ (or). A sentence using $\vee$, such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction** of the **disjuncts** $(W_{1,3} \wedge P_{3,1})$ and $W_{2,2}$. (Historically, the $\vee$ comes from the Latin "vel," which means "or." For most people, it is easier to remember $\vee$ as an upside-down $\wedge$.)
- $\Rightarrow$ (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is $(W_{1,3} \wedge P_{3,1})$, and its **conclusion** or **consequent** is $\neg W_{2,2}$. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as $\supset$ or $\rightarrow$.
- $\Leftrightarrow$ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**. Some other books write this as $\equiv$.

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

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 $P_{1,2}$, $P_{2,2}$, and $P_{3,1}$, then one possible model is

$$m_1 = \{P_{1,2} = false,\ P_{2,2} = false,\ P_{3,1} = true\}\ .$$

With three proposition symbols, there are $2^3 = 8$ possible models—exactly those depicted in Figure 7.5. Notice, however, that the models are purely mathematical objects with no necessary connection to wumpus worlds. $P_{1,2}$ 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 m_1 given earlier, $P_{1,2}$ 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"):

- $\neg P$ is true iff P is false in m.
- $P \wedge Q$ is true iff both P and Q are true in m.
- $P \vee Q$ is true iff either P or Q is true in m.
- $P \Rightarrow Q$ is true unless P is true and Q is false in m.
- $P \Leftrightarrow Q$ 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. For example,

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 sentence $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$, evaluated in m_1, gives $true \wedge (false \vee true) = true \wedge true = true$. Exercise 7.3 asks you to write the algorithm PL-TRUE?(s, m), which computes the truth value of a propositional logic sentence s in a model m.

7.4.3 A simple knowledge base

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We focus first on the *immutable* aspects of the wumpus world, leaving the mutable aspects for a later section. For now, we need the following symbols for each [x, y] location:

$P_{x,y}$ 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 $\neg P_{1,2}$ (there is no pit in [1,2]), as was done informally in Section 7.3. We label each sentence R_i so that we can refer to them:

- There is no pit in [1,1]:

$$R_1 : \quad \neg P_{1,1} .$$

- 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:

$$R_2 : \quad B_{1,1} \quad \Leftrightarrow \quad (P_{1,2} \vee P_{2,1}) .$$
$$R_3 : \quad B_{2,1} \quad \Leftrightarrow \quad (P_{1,1} \vee P_{2,2} \vee P_{3,1}) .$$

- 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, leading up to the situation in Figure 7.3(b).

$$R_4 : \quad \neg B_{1,1} .$$
$$R_5 : \quad B_{2,1} .$$

7.4.4 A simple inference procedure

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. Like the BACKTRACKING-SEARCH algorithm on page 215, TT-ENTAILS? performs a recursive enumeration of a finite space of assignments to symbols. 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.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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].

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.)

```
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*.

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.