Module 3

INFORMED (HEURISTIC) SEARCH STRATEGIES

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.

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.

3.1 Greedy best-first search

Greedy best-first search[8] 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;

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.

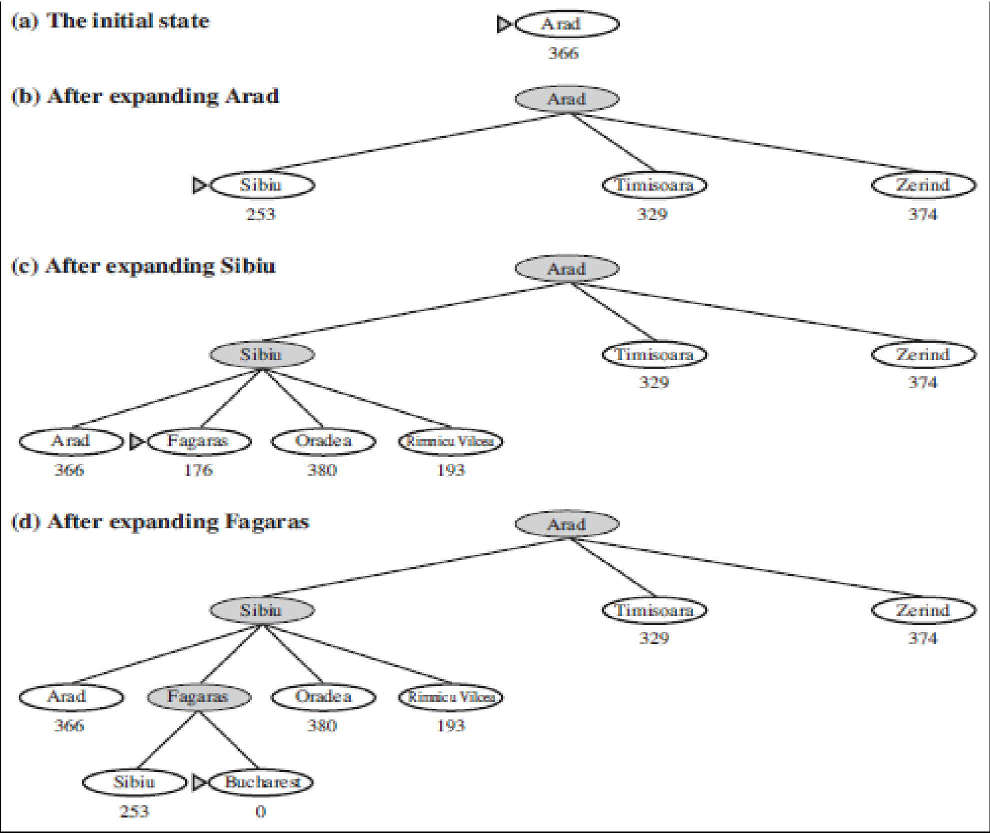


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

3.2 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

$f(n) =$ 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.

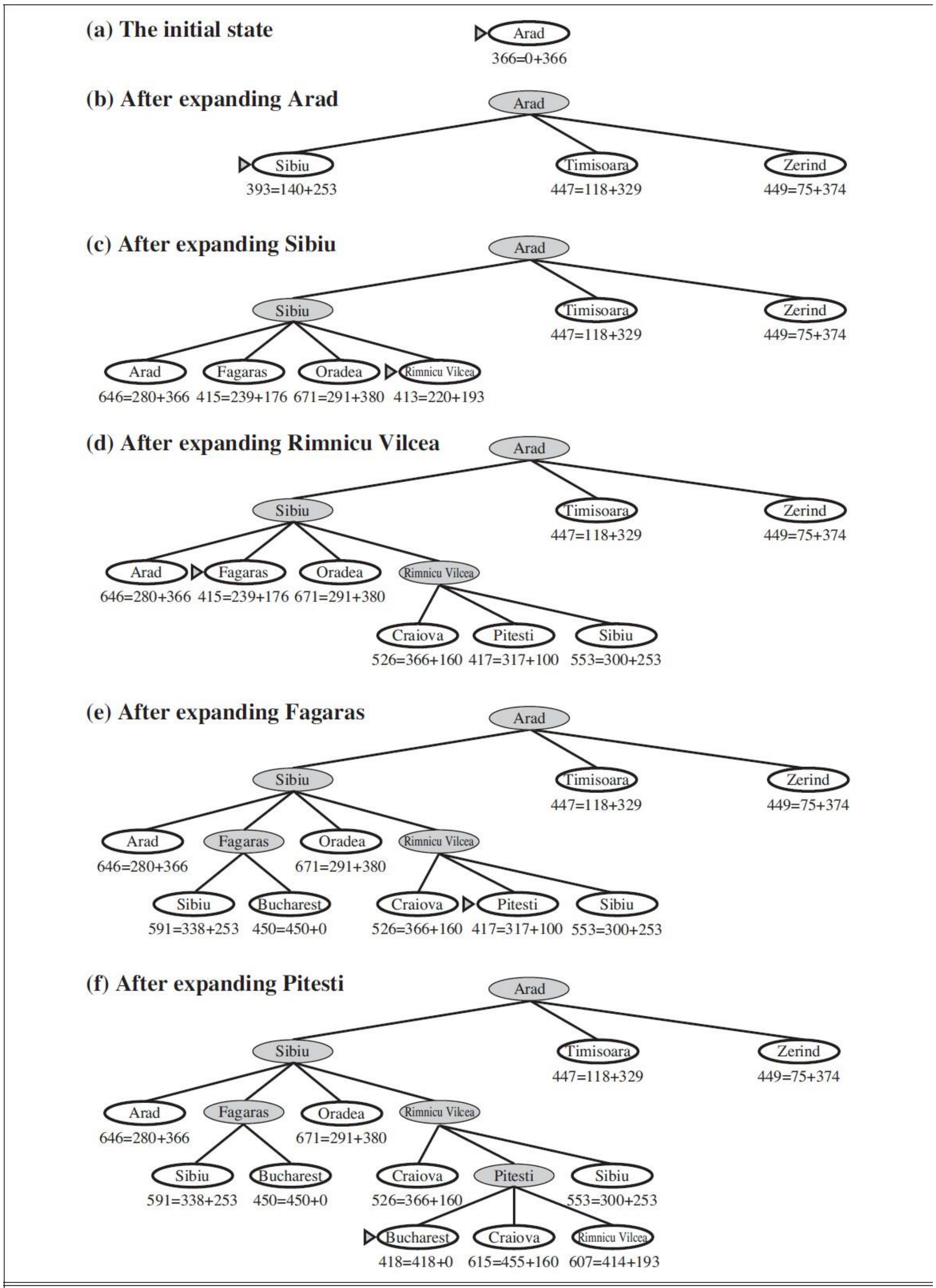


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

<u>Conditions for optimality: Admissibility and consistency</u>

1. Admissible heuristic

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.

2. Consistency

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^t 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^t plus the estimated cost of reaching the goal from n^t:

$$h(n) \leq c(n, a, n^t) + h(n^t) .$$

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^t, and the goal G_n closest to n.

For an admissible heuristic, the inequality makes perfect sense: if there were a route from n to G_n via n^t that was cheaper than $h(n)$, that would violate the property that $h(n)$ is a lower bound on the cost to reach G_n.

<u>Optimality of A*</u>

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.*

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^t is a successor of n; then $g(n^t)= g(n)+ c(n, a, n^t)$ for some action a, and we have

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

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^t on the optimal path from the start node to n, by the graph separation property of GRAPH-SEARCH; because f is nondecreasing along any path, n^t would have lower f-cost than n and would have been selected first.

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.

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.

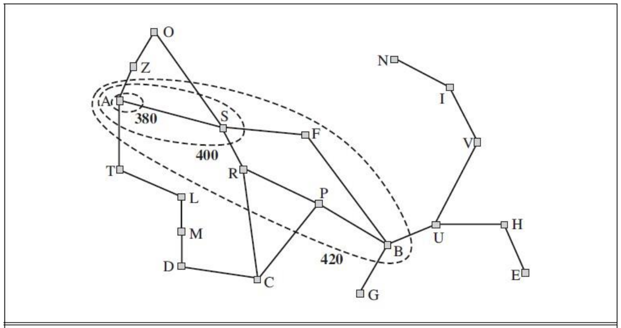


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

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 E and if b is finite.

Notice that A* expands no nodes with $f(n) > C^*$

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.

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 $E \equiv (h* - h)/h*$.

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 $O(b^{cd})$, where d is the solution depth. For almost all heuristics in practical use, the absolute error is at least proportional to the path cost h^*, so E is constant or growing and the time complexity is exponential in d. We can also see the effect of a more accurate heuristic: $O(b^{cd}) = O((b^c)^d)$.

3.3 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∗) 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 small- est 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.

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.

- 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.

- RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth re expanding 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.

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

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 $\leftarrow$ [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, $\infty$
  for each s in successors do  /* update f with value from previous search, if any */
      $s.f \leftarrow \max(s.g + s.h,\ node.f)$
  loop do
      best $\leftarrow$ the lowest f-value node in successors
      if $best.f > f\_limit$ then return failure, best.f
      alternative $\leftarrow$ the second-lowest f-value among successors
      result, best.f $\leftarrow$ RBFS(problem, best, min(f_limit, alternative))
      if result $\neq$ failure then return result
```

Figure 3.26 The algorithm for recursive best-first search.

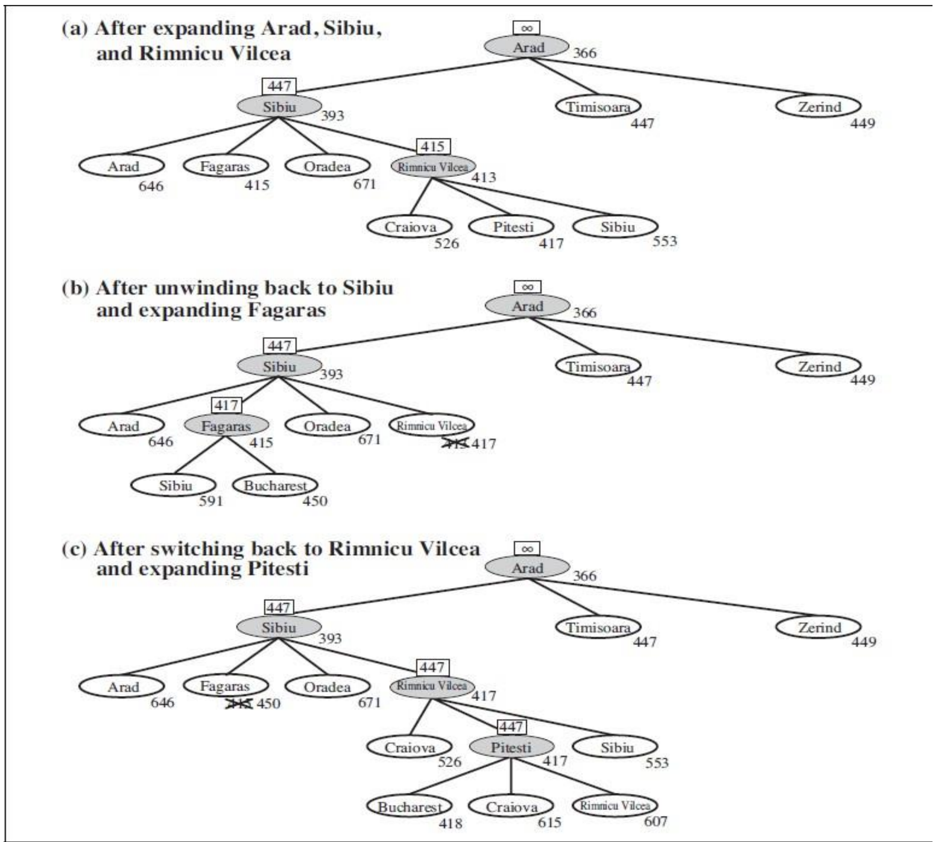


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse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

Limitations

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 re-expanding the same states many times over.
- Furthermore, they suffer the potentially exponential increase in complexity associated with redundant paths in graphs.

MA∗(memory-bounded A∗)

Two algorithms that use all available memory are MA∗ (memory-bounded A∗) and SMA∗ (simplified MA∗).

- 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.
- 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 is, if all the descendants of a node n are forgotten, then 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.

3.4 HEURISTIC FUNCTIONS

We look at heuristics for the 8-puzzle, in order to shed light on the nature of heuristics in general.

- 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 $3^{22} \approx 3.1\times10^{10}$ 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.

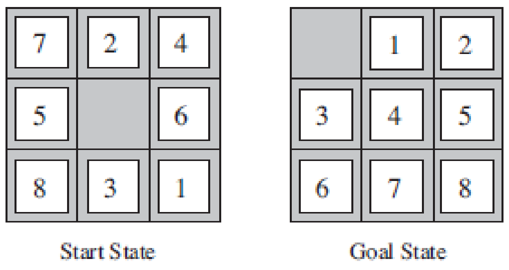


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

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

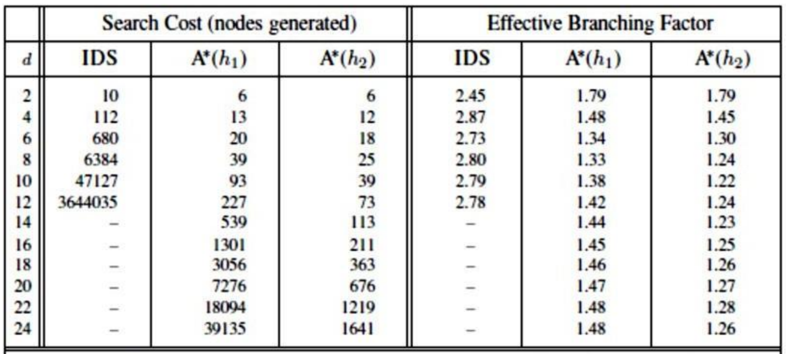
h2 = 3+1 + 2 + 2+ 2 + 3+ 3 + 2 = 18 .

As expected, neither of these overestimates the true solution cost, which is 26.

i. The effect of heuristic accuracy 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, $N + 1 = 1+b^* + (b^*)^2 + \cdot \cdot \cdot + (b^*)^d$.
- For example, if A* finds a solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.
- A well designed heuristic would have a value of b* close to 1.



	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.

To test the heuristic functions h_1 and h_2, 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 h_1 and h_2. Figure 3.29 gives the average number of nodes generated by each strategy and the effective branching factor.

One might ask whether h_2 is *always* better than h_1. The answer is "Essentially, yes." It is easy to see from the definitions of the two heuristics that, for any node n, $h_2(n) \geq h_1(n)$. We thus say that h_2 **dominates** h_1. Domination translates directly into efficiency: A* using h_2 will never expand more nodes than A* using h_1.

ii. 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.* Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore **consistent**.

If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.[11] 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 h_2 (Manhattan distance). The reasoning is that h_2 would be the proper score if we moved each tile in turn to its destination. The heuristic derived from (b) is h_1 (misplaced tiles). From (c), we can derive h_1 (misplaced tiles).

One problem with generating new heuristic functions is that one often fails to get a single "clearly best" heuristic. If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem and none of them dominates any of the others, which should we choose? As it turns out, we need not make a choice. We can have the best of all worlds, by defining

$$h(n) = \max\{h_1(n), \ldots, h_m(n)\}$$

This composite heuristic uses whichever function is most accurate on the node in question. Because the component heuristics are admissible, h is admissible; it is also easy to prove that h is consistent. Furthermore, h dominates all of its component heuristics.

iii. 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.

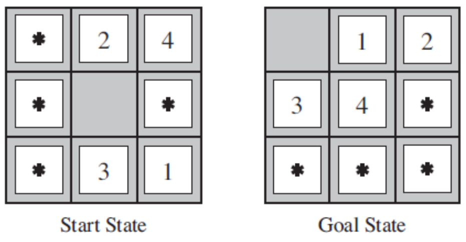


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

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 h_{DB} 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.

The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, for 2-4-6-8, and so on. Each database yields an admissible heuristic, and these heuristics can be combined, as explained earlier, by taking the maximum value.

The heuristics obtained from the 1-2-3-4 database and the 5-6-7-8 could be *added*, since the two subproblems seem not to overlap. This is not an admissible heuristic, because the solutions of the 1-2-3-4 subproblem and the 5-6-7-8 subproblem for a given state will almost certainly share some moves it is unlikely that 1-2-3-4 can be moved into place without touching 5-6-7-8, and vice versa.

The sum of the two costs is still a lower bound on the cost of solving the entire problem is a **disjoint pattern databases**.

iv. Learning heuristics from experience

A heuristic function $h(n)$ is supposed to estimate the cost of a solution beginning from the state at node n.

How could an agent construct such a function?

Solution: learn from experience.

Example:

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.

For example, the feature “number of misplaced tiles” might be helpful in predicting the actual distance of a state from the goal. Let’s call this feature $x_1(n)$. We could take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs. We might find that when $x_1(n)$ is 5, the average solution cost is around 14, and so on. Given these data, the value of x_1 can be used to predict $h(n)$. Of course, we can use several features. A second feature $x_2(n)$ might be “number of pairs of adjacent tiles that are not adjacent in the goal state.” How should $x_1(n)$ and $x_2(n)$ be combined to predict $h(n)$? A common approach is to use a linear combination:

$$h(n) = c_1x_1(n) + c_2x_2(n) .$$

The constants c_1 and c_2 are adjusted to give the best fit to the actual data on solution costs.

<u>LOGICAL AGENTS</u>

3.5 <u>Knowledge—based agents</u>

- An intelligent agent needs knowledge about the real world for taking decisions and reasoning to act efficiently.
- Knowledge-based agents are those agents who have the capability of maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently.
- Knowledge-based agents are composed of two main parts:
    - Knowledge-base and

- Inference system.

- A knowledge-based agent must able to do the following:
    1. An agent should be able to represent states, actions, etc.
    2. An agent Should be able to incorporate new percepts
    3. An agent can update the internal representation of the world
    4. An agent can deduce the internal representation of the world
    5. An agent can deduce appropriate actions

Knowledge base: It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English). These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.

Why use a knowledge base?

Knowledge-base is required for updating knowledge for an agent to learn with experiences and take action as per the knowledge.

Inference system

Inference means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information. Inference system generates new facts so that an agent can update the KB.

Operations Performed by KBA.

Following are two operations which are performed by KBA in order to show the intelligent behavior:

- **TELL:** This operation tells the knowledge base what it perceives from the environment.
- **ASK:** This operation asks the knowledge base what action it should perform.

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.

```
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
```

Each time when the function is called, it performs its three operations:

- Firstly it TELLs the KB what it perceives.
- Secondly, it asks KB what action it should take
- Third agent program TELLS the KB that which action was chosen.
- The MAKE-PERCEPT-SENTENCE generates a sentence as setting that the agent perceived the given percept at the given time.
- The MAKE-ACTION-QUERY generates a sentence to ask which action should be done at the current time.
- MAKE-ACTION-SENTENCE generates a sentence which asserts that the chosen action was executed.

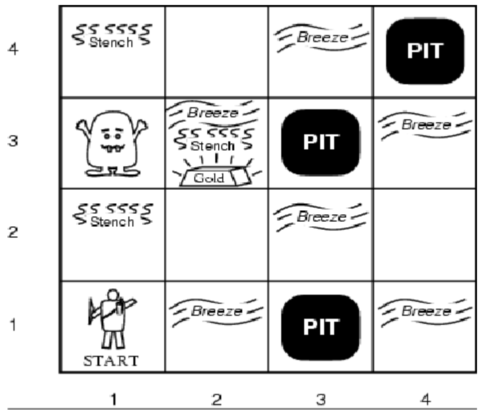
Various levels of knowledge-based agent:

A knowledge-based agent can be viewed at different levels which are given below:

1. Knowledge level
   - Knowledge level is the first level of knowledge-based agent, and in this level, we need to specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.
2. Logical level:
   - At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs. Example: Links(GoldenGateBridge, SanFrancisco, MarinCounty).
3. Implementation level:
   - This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

3.6 The Wumpus World environment

The Wumpus world is a cave which has 4/4 rooms connected with passageways. So there are total 16 rooms which are connected with each other. We have a knowledge-based agent who will go forward in this world. The cave has a room with a beast which is called Wumpus, who eats anyone who enters the room. The Wumpus can be shot by the agent, but the agent has a single arrow.



- The agent explores a cave consisting of rooms connected by passageways.
- Lurking somewhere in the cave is the Wumpus, a beast that eats any agent that enters its room.
- Some rooms contain bottomless pits that trap any agent that wanders into the room.
- Occasionally, there is a heap of gold in a room.
- The goal is to collect the gold and exit the world without being eaten.

PEAS description of Wumpus world:

Performance measure:

- +1000 reward points if the agent comes out of the cave with the gold.
- -1000 points penalty for being eaten by the Wumpus or falling into the pit.
- -1 for each action, and -10 for using an arrow.
- The game ends if either agent dies or came out of the cave.

Environment:

- A 4*4 grid of rooms.
- The agent initially in room square [1, 1], facing toward the right.
- Location of Wumpus and gold are chosen randomly except the first square [1,1].
- Each square of the cave can be a pit with probability 0.2 except the first square.

Actions/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

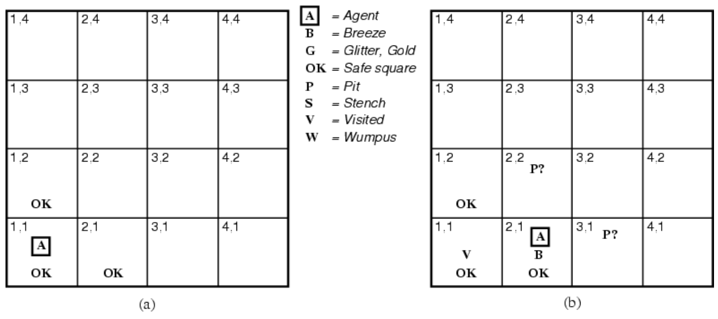
[Stench, Breeze, None, None, None].

The Wumpus agent's first step

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].



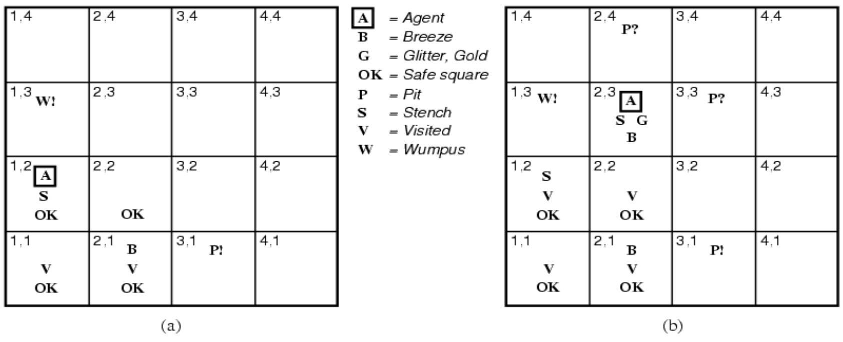
- Now agent needs to move forward, so it will either move to [1, 2], or [2,1]. Let's suppose agent moves to the room [2, 1], at this room agent perceives some breeze which means Pit is around this room. The pit can be in [3, 1], or [2,2], so we will add symbol P? to say that, is this Pit room?
- Now agent will stop and think and will not make any harmful move. The agent will go back

to the [1, 1] room. The room [1,1], and [2,1] are visited by the agent, so we will use symbol V to represent the visited squares.

- At the third step, now agent will move to the room [1,2] which is OK. In the room [1,2] agent perceives a stench which means there must be a Wumpus nearby. But Wumpus cannot be in the room [1,1] as by rules of the game, and also not in [2,2] (Agent had not detected any stench when he was at [2,1]). Therefore agent infers that Wumpus is in the room [1,3], and in current state, there is no breeze which means in [2,2] there is no Pit and no Wumpus. So it is safe, and we will mark it OK, and the agent moves further in [2,2].
- At room [2,2], here no stench and no breezes present so let's suppose agent decides to move to [2,3]. At room [2,3] agent perceives glitter, so it should grab the gold and climb out of the cave.

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].

- The agent perceives a stench in [1,2], resulting in the state of knowledge. 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. The lack of a breeze in [1,2] implies that there is no pit in [2,2].
- 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. assume that the agent turns and moves to [2,3]. In [2,3], the agent detects a glitter, so it should grab the gold and then return home.



3.7 <u>Logic</u>

The fundamental concepts of logical representation and reasoning.

- Knowledge bases consist of sentences.
- Sentences are expressed according to the **syntax** of the representation language.
  Example: “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 (model)**.
  Example: 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.

- 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, say that m **satisfies** α or sometimes m **is a model of** α.
- The notation: M(α) to mean the set of all models of α.
- Notion of truth involves the relation of logical entailment between sentences—the idea that a sentence follows logically from another sentence.
  Mathematical notation: α |= β (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.
  α |= β if and only if M(α) ⊆ M(β)

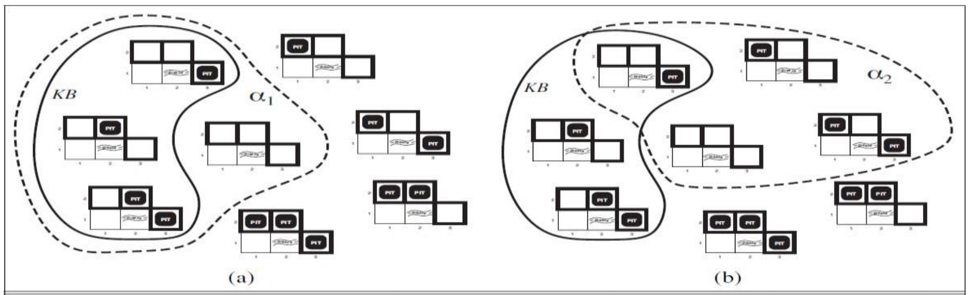


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in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

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.

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.

Let us consider two possible conclusions:

α1 = "There is no pit in [1,2]."
α2 = "There is no pit in [2,2]."

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].
- 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].

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).

Formal notation:

If an inference algorithm i can derive α from KB, we write $\boxed{KB \vdash_i \alpha\,,}$
which is pronounced "α is derived from KB by i" or "i derives α from KB."

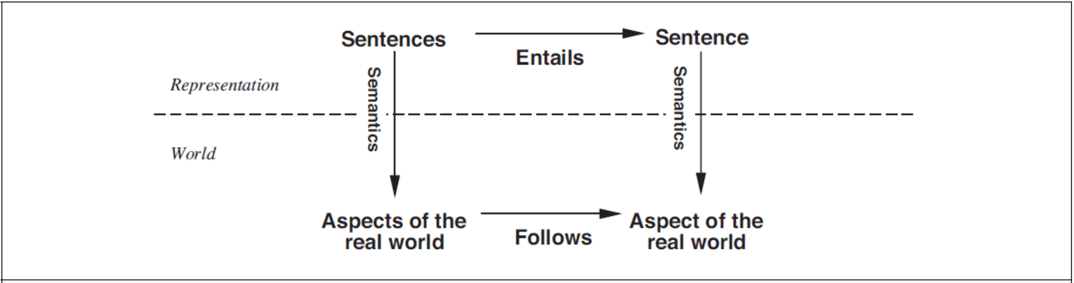


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations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

Sound or truth- preserving

An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving**.

Completeness

The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed.

Grounding

The **grounding**—the connection between logical reasoning processes and the real environment in which the agent exists.

This correspondence between world and representation is illustrated in Figure 7.6

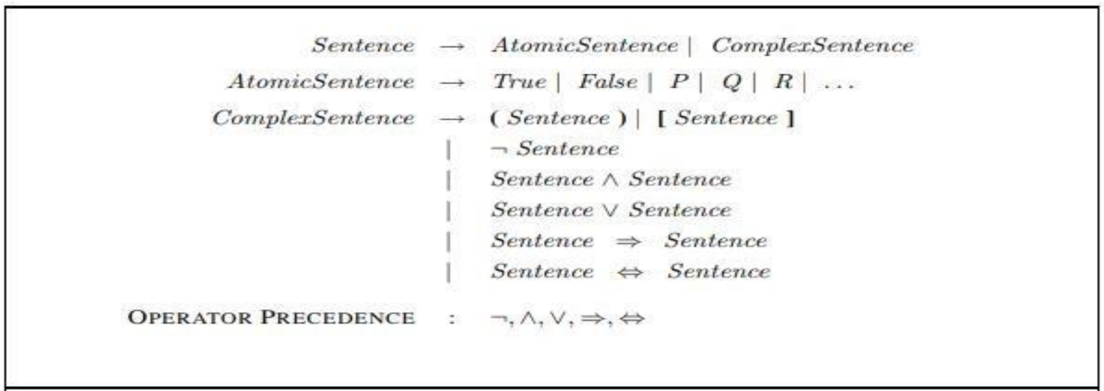
3.8 PROPOSITIONAL LOGIC:A VERY SIMPLE LOGIC

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. 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*.
- **Complex sentences** are constructed from simpler sentences, using parentheses and **logical connectives**.
- There are five connectives in common use:
  - ¬ (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**).
  - ∧ (and). A sentence whose main connective is ∧, such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction**.
  - ∨ (or). A sentence using ∨, such as (W1,3∧P3,1)∨W2,2, is a disjunction of the **disjunction**

(W1,3 ∧ P3,1) and W2,2.

- ⇒ (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** .Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as ⊃ or →.
- ⇔ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**. Some other books write this as ≡.



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.

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

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

The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model.

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"):

- $\neg P$ 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*.

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 knowledge base

- To construct a knowledge base for the wumpus world.
- Focus first on the *immutable* aspects of the wumpus world, mutable aspects are focused later.
- 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].

- We label each sentence Ri so that we can refer to them:

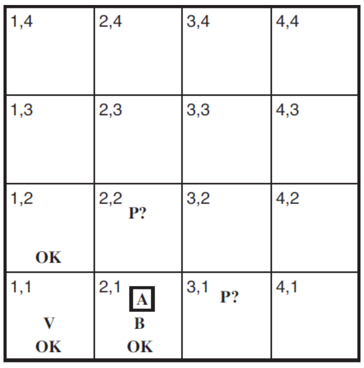
  There is no pit in [1,1]: R1 : ¬P1,1

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

  $R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

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

- The preceding sentences are true in all wumpus worlds.
- Include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation given in Figure.



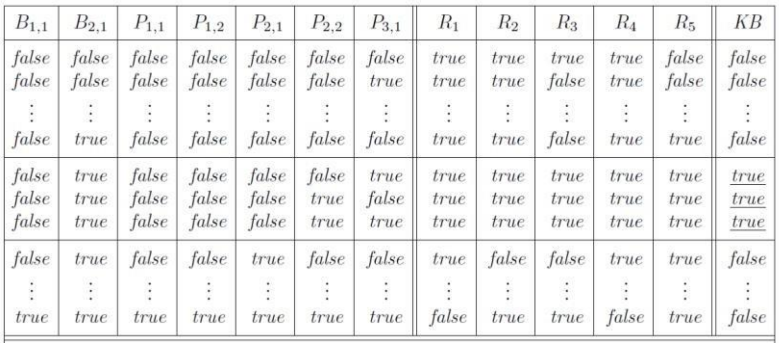
R4 : ¬B1,1
R5 : B2,1

A simple inference procedure

- Our goal now is to decide whether KB |= α for some sentence α.
- Our first algorithm for inference is a model-checking approach:
    - enumerate the models, and
    - check that α is true in every model in which KB is true.
- Models are assignments of true or false toevery proposition symbol.

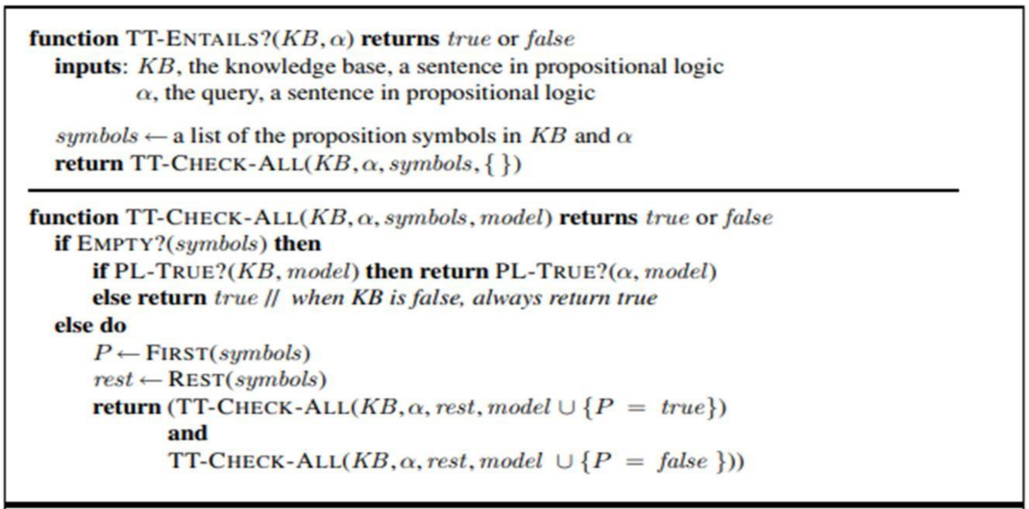
Wumpus-world example:

- The relevant proposition symbols are $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$, and $P_{3,1}$.
- there are $2^7 = 128$ possible models
- In three of these, *KB* is true



$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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*	<u>*true*</u>
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	<u>*true*</u>
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	<u>*true*</u>
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?($KB, \alpha$) returns true or false
  inputs: $KB$, the knowledge base, a sentence in propositional logic
          $\alpha$, the query, a sentence in propositional logic

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

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

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment.

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. If *KB* and *α* contain *n* symbols in all, then there are 2^n models. Thus, the time complexity of the algorithm is $O(2^n)$. The space complexity is only $O(n)$ because the enumeration is depth-first.

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

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