MODULE 2

1. Informed (Heuristic) Search Strategies

- Greedy best first search
- A* search
- Optimality of A*
- Memory-bounded heuristic search

2. Local Search Algorithms and Optimization Problems

- Hill-climbing search
- Simulated annealing
- Local beam search
- Genetic algorithms

3. Logical Agents

- Knowledge-based agents
- The Wumpus world
- Logic
- Propositional logic
- Propositional theorem proving

INFORMED (HEURISTIC) SEARCH STRATEGIES

Informed search strategy:-one that uses problem-specific knowledge beyond the definition of the problem itself and can find solutions more efficiently than an uninformed strategy.

- Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This knowledge helps agents to explore less to the search space and find more efficiently the goal node.
- Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics Function

- Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal.
- Heuristic function is represented by h(n), and it calculates the cost of an optimal path between the pair of states.
- The value of the heuristic function is always **positive.**

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 **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 2.1

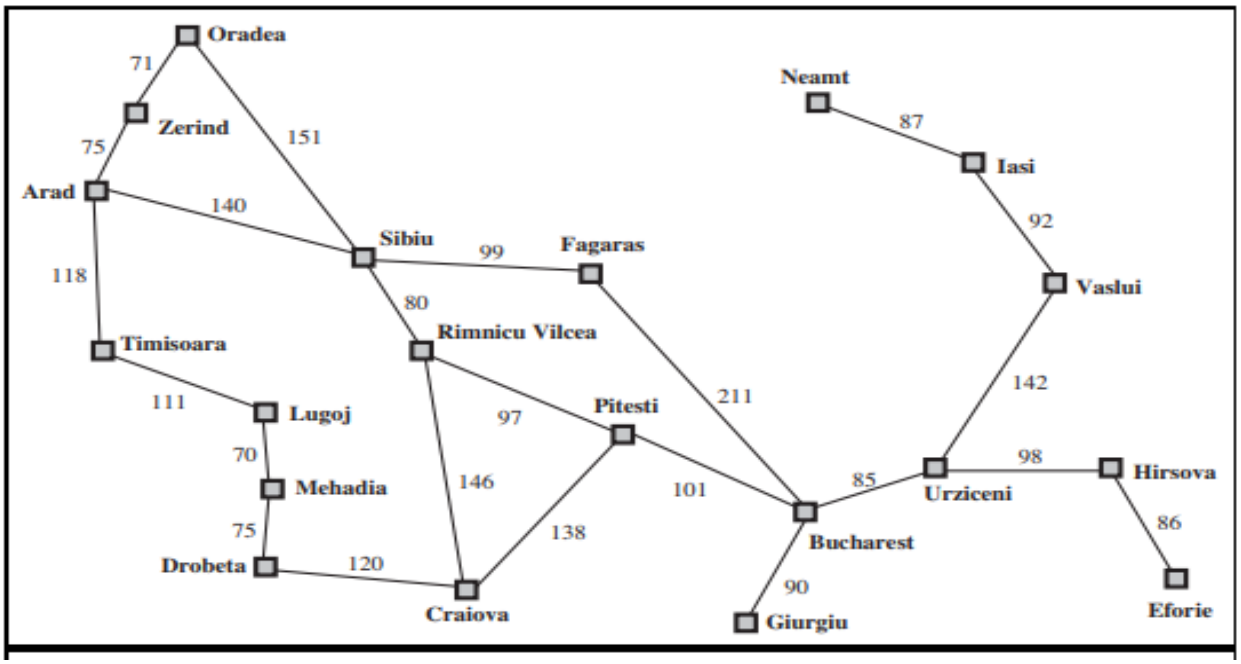


Figure 2: A simplified road map of part of Romania

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 2.1: Values of h_{SLD} straight-line distances to Bucharest.

Greedy best-first search example

Figure 2.2 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.

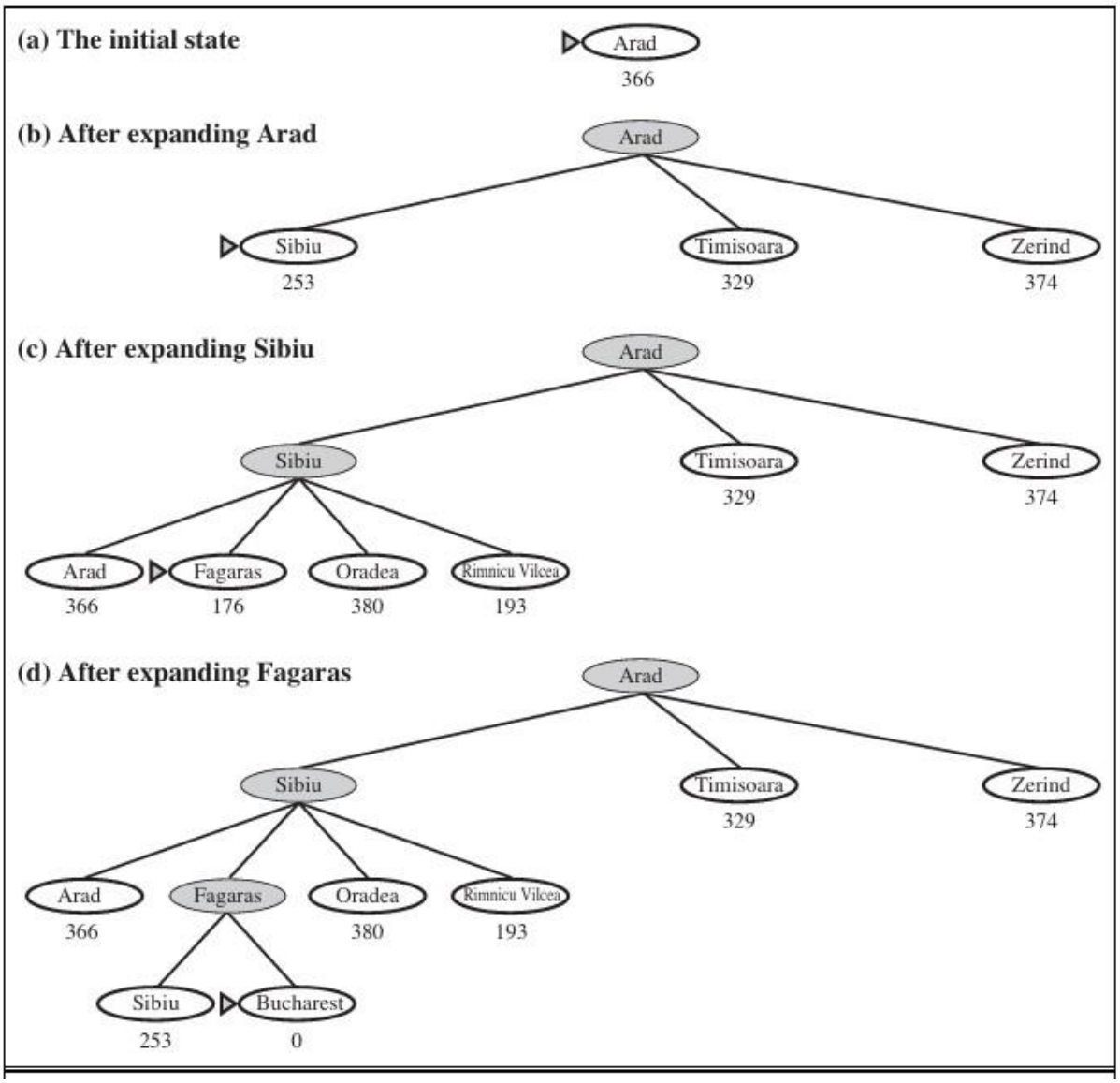
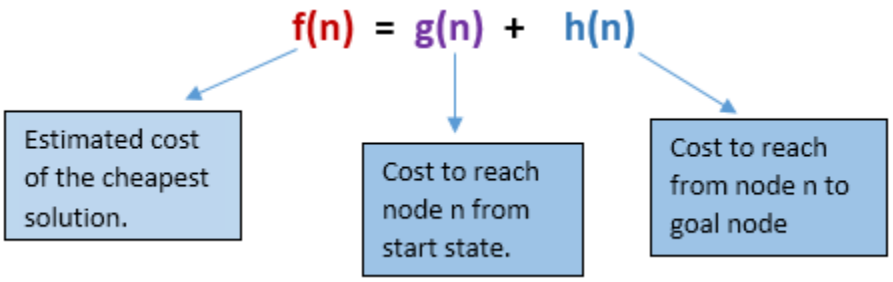


Figure 2.2: Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is **not optimal.**
- **Time and space complexity** for the tree version is **$O(b^m)$,** where m is the maximum depth of the search space.

A* Search

- A* search is the most commonly known form of best-first search.
- It uses heuristic function h(n), and cost to reach the node n from the start state g(n).
- It has combined features of UCS (Uniform Cost Search) and Greedy Best-First Search, by which it solves the problem efficiently.
- A∗ search is both **complete and optimal**



Conditions for optimality: Admissibility and consistency

- The first condition we require for **optimality** is that heuristic function h(n) be an **admissible heuristic** for every node n, h(n)<=h*(n).
  where, h*(n) is true cost to reach the goal state from n.
- An **admissible heuristic** is one that never overestimates the cost to reach the goal.
  Example: $h_{SLD}(n)$ (never overestimate the actual road distance)
- A second condition, heuristic function h(n) is said to be **consistent** if for every node n, every successor n' of n generated by any action a

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

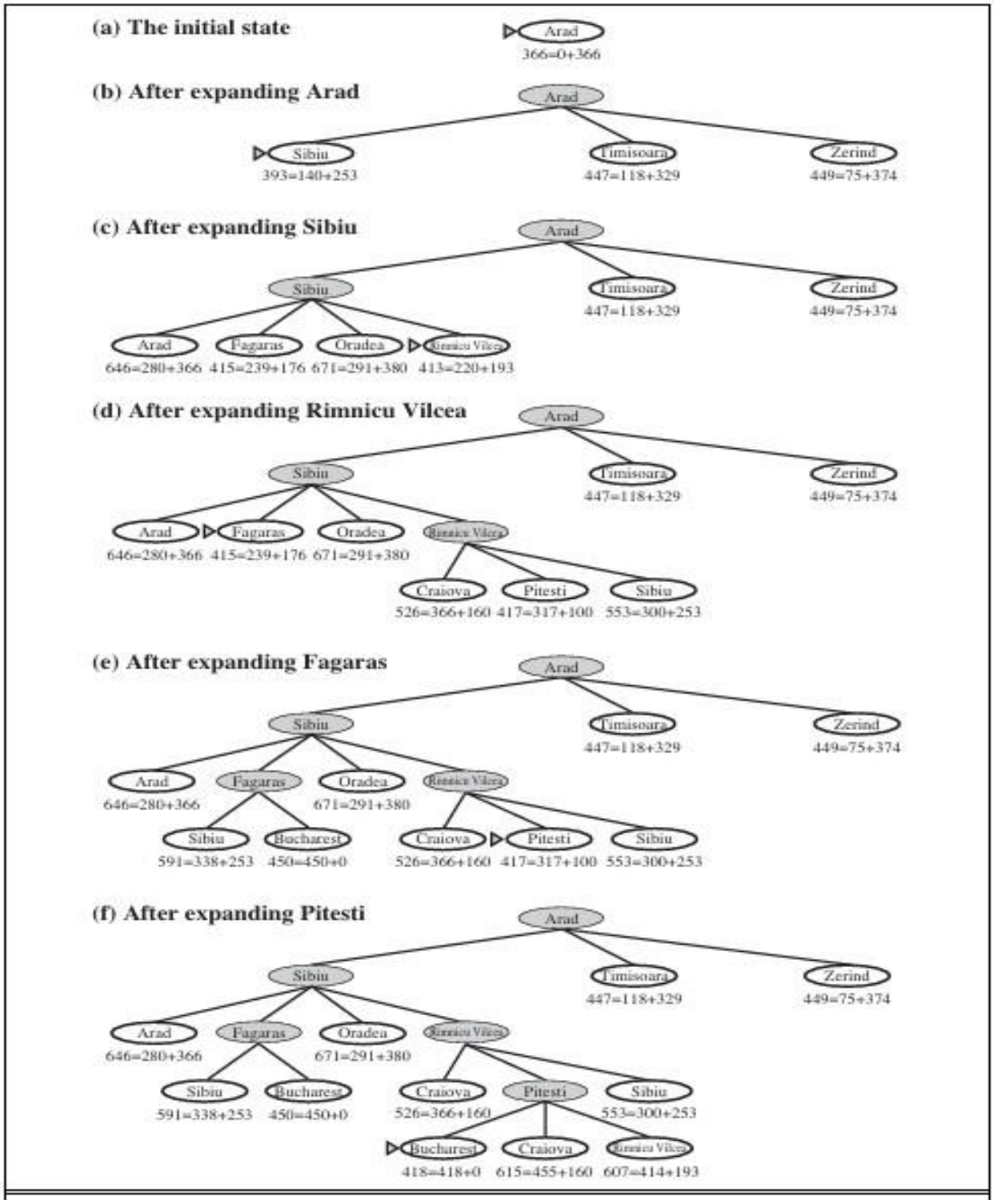


Figure 2.3: Stages in an A∗ search for Bucharest. Nodes are labeled with **f = g+h.** The h values are the straight-line distances to Bucharest taken from Figure 2.1

Memory-bounded heuritic 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.

Two other memory-bounded algorithms: -

1. RBFS
2. MA∗

1) 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.

```
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 2.4: The algorithm for recursive best-first search.

- 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.

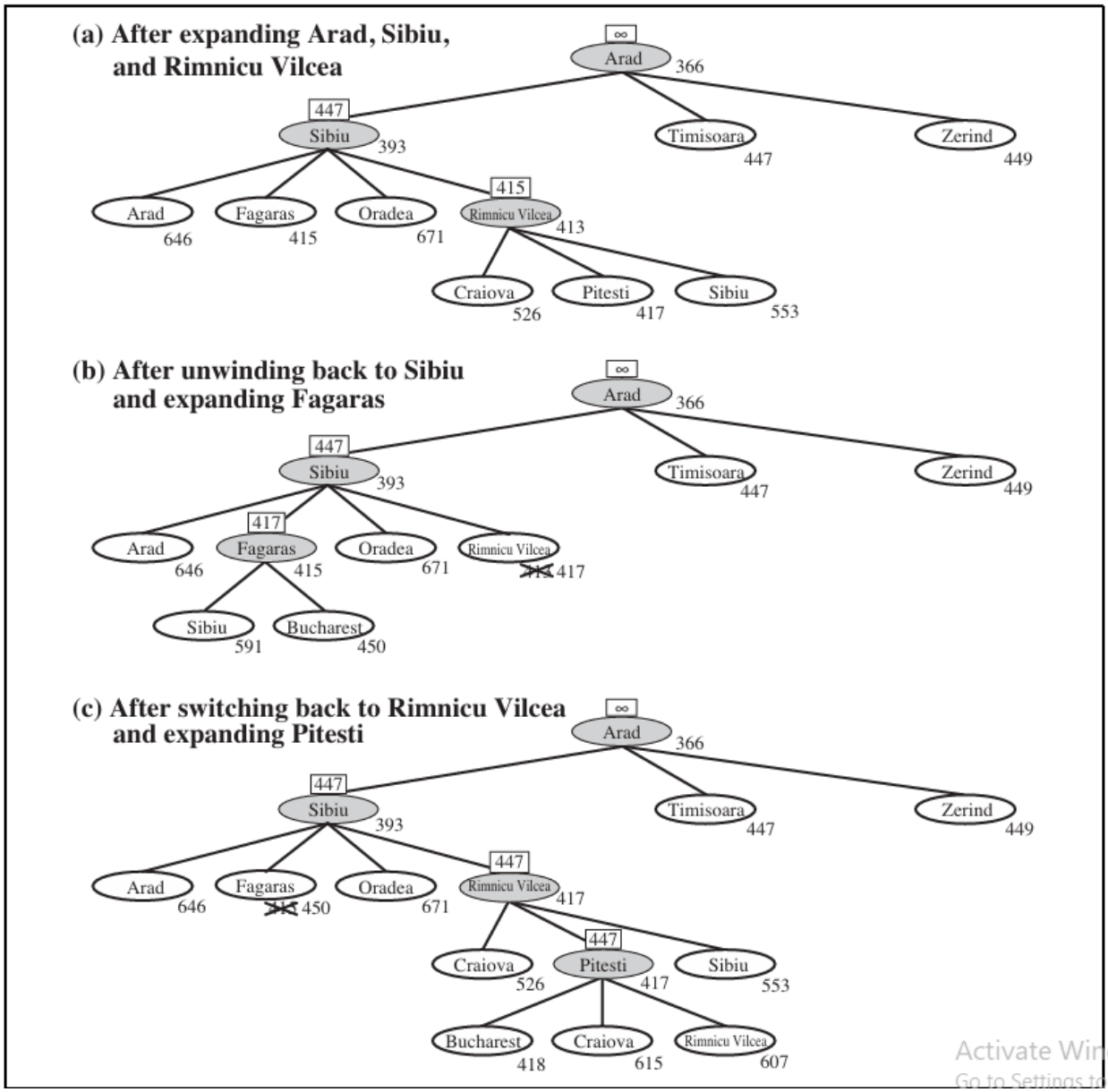


Figure 2.5: Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. **(a)** The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is 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 upto Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs atleast 447, the expansion continues to Bucharest.

2) 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.
- In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree.
- With this information, SMA∗ regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten.

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

Local search algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node.

Although local search algorithms are not systematic, they have two key advantages:

(1) they use very little memory—usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

1) Hill-climbing search (steepest-ascent version)

- It is simply a loop that continually moves in the direction of increasing value— uphill.
- It terminates when it reaches a "peak" where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.
- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.

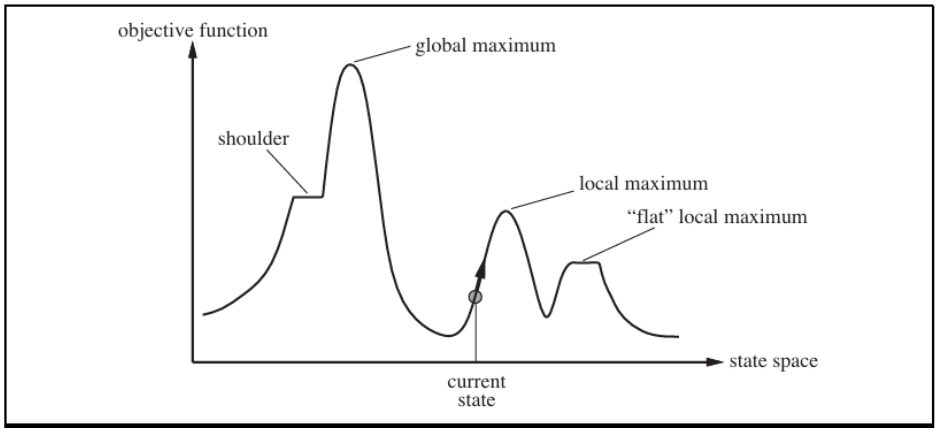


Figure 2.6: A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor ← a highest-valued successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
        current ← neighbor
```

Figure 2.7: The hill-climbing search algorithm

Unfortunately, hill climbing often gets stuck for the following reasons:

- **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
- **Ridges:** Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux:** a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible.

To illustrate hill climbing, we will use the 8-queens problem

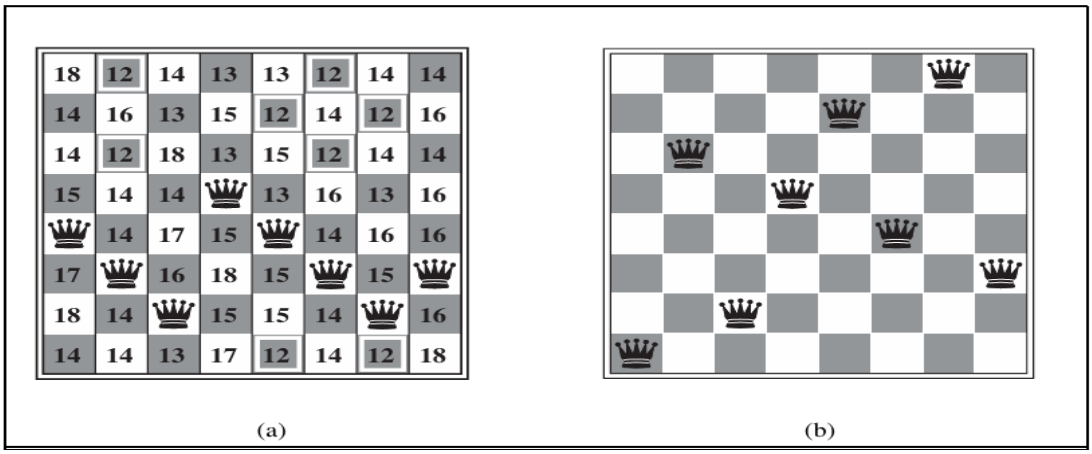


Figure 2.8: (a) An 8-queens state with heuristic cost estimate h=17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has h=1but every successor has a higher cost.

- Local search algorithms typically use a complete-state formulation, where each state has 8 queens on the board, one per column.
- The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8×7=56successors).
- The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.
- The global minimum of this function is zero, which occurs only at perfect solutions.

Many variants of hill climbing have been invented.

1. **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
2. **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
3. **Random-restart hill climbing** adopts the well-known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.

2) Simulated annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

Figure 2.9: The simulated annealing algorithm

- In metallurgy, annealing is the process used to temper or harden metals and glass by

heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.

- Imagine the task of getting a **ping-pong ball** into the deepest crevice in a bumpy surface.
- If we just let the ball roll, it will come to rest at a local minimum.
- If we shake the surface, we can bounce the ball out of the local minimum.
- The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum.
- The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.

3) Local Beam Search

- The local beam search algorithm keeps track of k states rather than just one. It begins with k randomly generated states.
- At each step, all the successors of all k states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.
- In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing.
- A variant called stochastic beam search, analogous to stochastic hill climbing, helps alleviate this problem.
- Instead of choosing the best k from the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.
- Stochastic beam search bears some resemblance to the process of natural selection, whereby the “successors” (offspring) of a “state” (organism) populate the next generation according to its “value” (fitness).

4) Genetic Algorithms

- A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining *two* parent states rather than by modifying a single state.
- GAs begin with a set of k randomly generated states, called the population.
- The production of the next generation of states is rated by the objective function, or (in GA terminology) the **fitness function**.
- For each pair to be mated, a **crossover point** is chosen randomly from the positions in the string.
- Finally, each location is subject to random mutation with a small independent probability.
- One digit was mutated in the first, third, and fourth offspring.
- In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

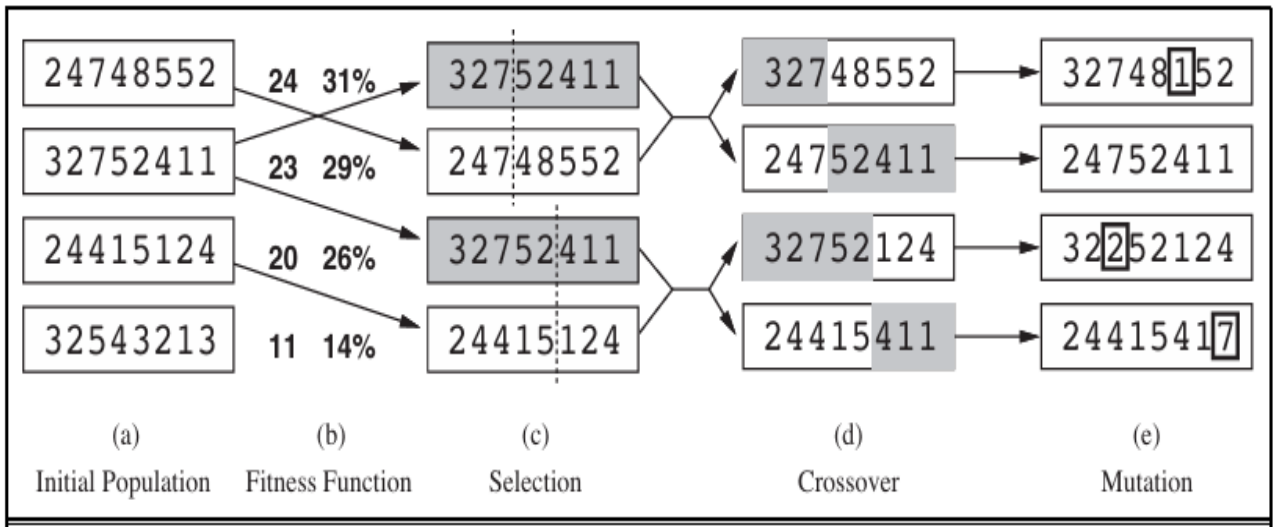


Figure 2.10: The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in **(a)** is ranked by the fitness function in **(b),** resulting in pairs for mating in **(c)** They produce offspring in **(d),** which are subject to mutation in **(e).**

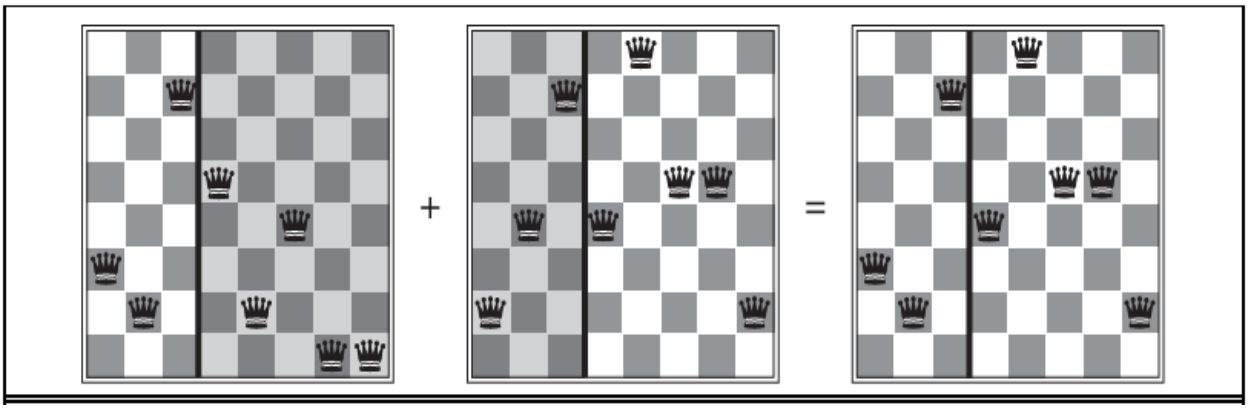


Figure 2.11: The 8-queens states corresponding to the first two parents in Figure 2.10(c) and the first offspring in Figure 2.10(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 2.12: A genetic algorithm.

LOGICAL AGENTS

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.
- Each sentence is expressed in a language called a knowledge representation language and represents some assertion about the world.
- 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.

```
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 2.13: A generic knowledge-based agent.

- 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.
- Third, the agent program **TELLs** the knowledge base which action was chosen, and the agent executes the action.
- **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 Wumpus World

- The wumpus world is a cave consisting of rooms connected by passageways.
- 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 goal is to find a heap of gold.

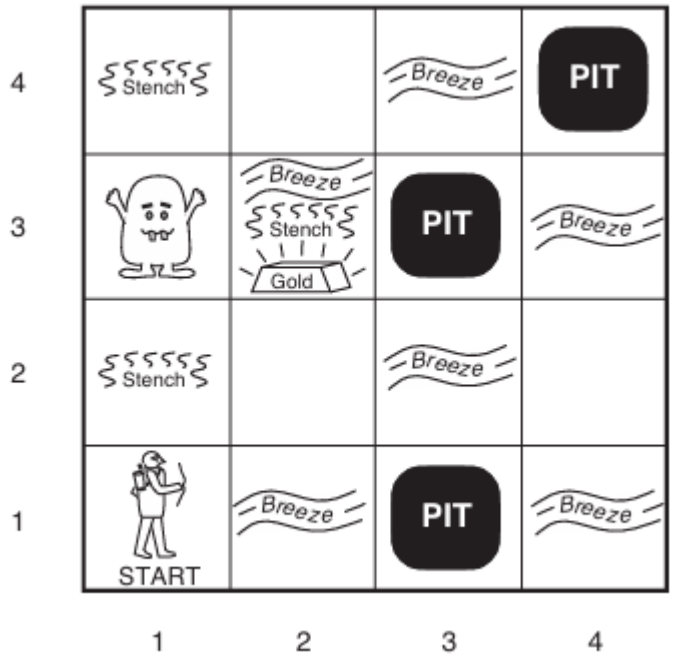


Figure 2.14: A typical wumpus world. The agent is in the bottom left corner, facing right.

PEAS description:

1. **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.

2. **Environment:** A 4×4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
3. **Actuators:**
    - The agent can move Forward, TurnLeft by 90◦, orTurnRight 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 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].
4. **Sensors:** The agent has five sensors, each of which gives a single bit of information: –
    - In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a Stench.
    - In the squares directly adjacent to a pit, the agent will perceive a Breeze.
    - In the square where the gold is, the agent will perceive a Glitter.
    - When an agent walks into a wall, it will perceive a Bump.
    - When the wumpus is killed, it emits a woeful Scream that can be perceived anywhere in the cave.

The percepts will be given to the agent program in the form of a list of five symbols.
for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get **[Stench, Breeze, None, None, None].**

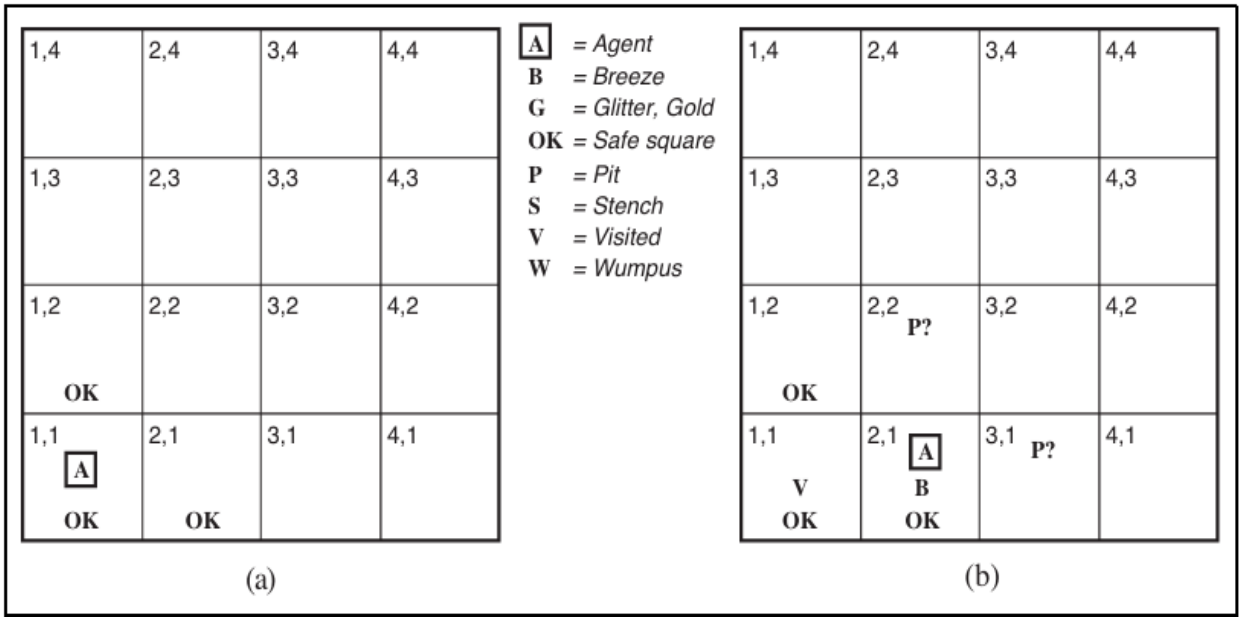


Figure 2.15: The first step taken by the agent in the wumpus world.

(a) The initial situation, after percept [None, None, None, None, None].

(b) After one move, with percept [None, Breeze, None, None, None].

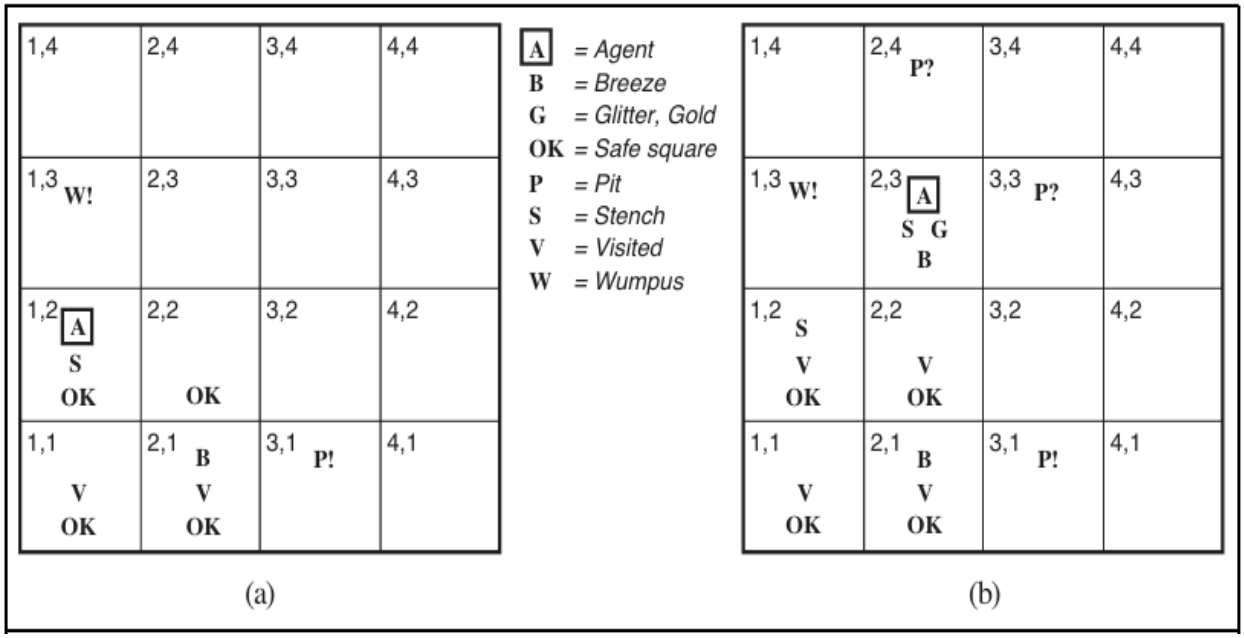


Figure 2.16: Two later stages in the progress of the agent.

(a) After the third move, with percept [Stench, None, None, None, None].

(b) After the fifth move, with percept [Stench, Breeze, Glitter, None, None].

Logic

- The 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.

When we need to be precise, we use the term **model** in place of "possible world."

- 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.
- 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).

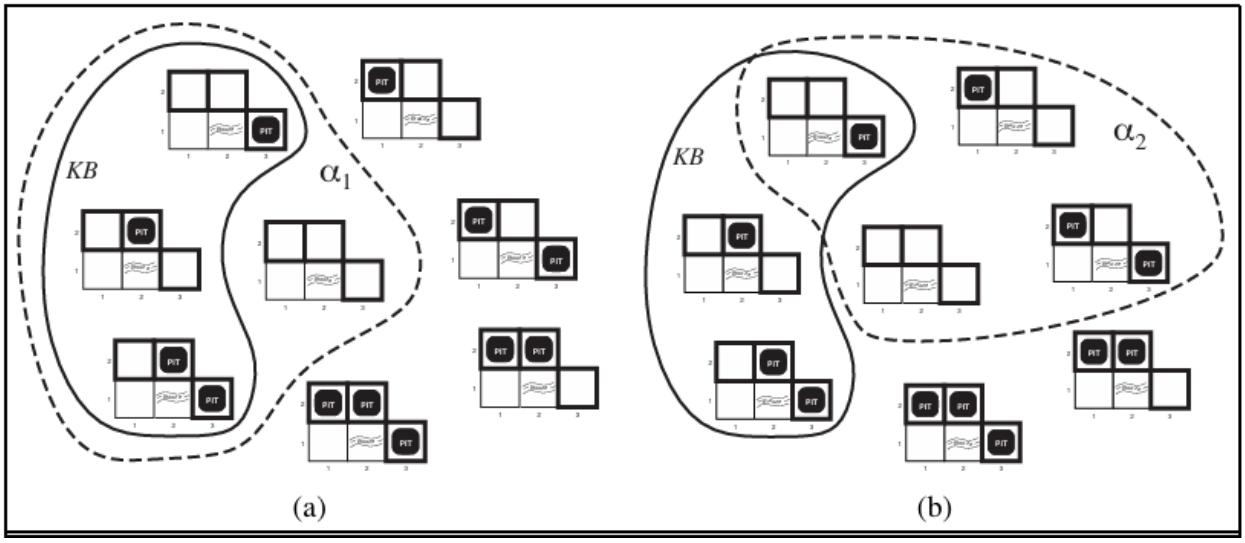


Figure 2.17: Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. **(a)** Dotted line shows models of α1 (no pit in [1,2]). **(b)** Dotted line shows models of α2(no pit in [2,2]).

Propositional Logic

Syntax

The **syntax** of propositional logic defines the allowable sentences.

1. 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.
   - W1,3 to stand for the proposition that the wumpus is in [1,3].
2. **Complex sentences** are constructed from simpler sentences, using parentheses and logical connectives. There are five connectives in common use:
   - ¬ (not). A sentence such as ¬W1,3 is called the negation of W1,3.
   - ∧ (and). A sentence whose main connective is ∧, suchasW1,3 ∧ P3,1, is called a con junction; its parts are the conjuncts.
   - ∨ (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.
   - ⇒ (implies). A sentence such as (W1,3∧P3,1) ⇒¬W2,2 is called an implication.
   - ⇔ (if and only if). The sentence W1,3 ⇔¬W2,2 is a biconditional.

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

Figure 2.18: 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.

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} .

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

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 2.19: Truth tables for the five logical connectives.

A simple knowledge base

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].

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

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

2. A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square;

   R2 : B1,1 ⇔ (P 1,2 ∨ P 2,1).

   R3 : B2,1 ⇔ (P 1,1 ∨ P 2,2 ∨ P 3,1).

3. The preceding sentences are true in all wumpus worlds.

   R4 : ¬ B1,1

   R5 : B2,1

A simple inference procedure

- Our goal now is to decide whether **KB |= α** for some sentence α.
- 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 2^7 =128 possible models; in three of these, KB is true.
- 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.
- If KB and α contain n symbols in all, then there are 2^n models. Thus, the **time complexity** of the algorithm is $O(2^n)$.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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 2.20: A truth table constructed for the knowledge base given in the text. KB is true if R1through R5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, P1,2 is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

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

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

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

Figure 2.21: 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 key word "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 2.22: Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

Propositional Theorem Proving

- The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as α ≡ β.
- i.e., P ∧ Q and Q ∧ P are logically equivalent;
- An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other: α ≡ β if and only if α |= β and β |= α.
- The second concept we will need is **validity**. A sentence is valid if it is true in all models.
- For example, the sentence P ∨ ¬ P is valid. Valid sentences are also known as **tautologies**—they are necessarily true.
- From our definition of entailment, we can derive the deduction theorem, which was known to the ancient Greeks: For any sentences α and β, α |= β if and only if the sentence (α ⇒ β) is valid.
- The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, some model. For example, the knowledge base given earlier, (R1 ∧ R2 ∧ R3 ∧R4 ∧R5), is satisfiable because there are three models in which it is true.

Inference and proofs

Inference rules can be applied to derive a proof—a chain of conclusions that leads to the desired goal. The best-known rule is called **Modus Ponens** and is written

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

The notation means that, whenever any sentences of the form α ⇒ β and α are given, then the sentence β can be inferred.

For example, if (WumpusAhead ∧ WumpusAlive) ⇒ Shoot and (WumpusAhead ∧ WumpusAlive) are given, then Shoot can be inferred.

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}$$

For example, from (WumpusAhead ∧ WumpusAlive), WumpusAlive can be inferred.

For example, the equivalence for **biconditional elimination** yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

Not all inference rules work in both directions like this.

Let us see how these inference rules and equivalences can be used in the WUMPUS world.

We start with the knowledge base containing **R1 through R5** and show how to prove ¬ P1,2, that is, there is no pit in [1,2].

- First, we apply biconditional elimination to R2 to obtain

  R6 : (B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .
- Then we apply And-Elimination to R6 to obtain

  R7 : ((P1,2 ∨ P2,1) ⇒ B1,1) .
- Logical equivalence for contrapositives gives

  R8 : (¬ B1,1 ⇒ ¬ (P1,2 ∨ P2,1)) .
- Now we can apply Modus Ponens with R8 and the percept R4 (i.e., ¬B1,1), to obtain

  R9 : ¬ (P1,2 ∨ P2,1) .
- Finally, we apply De Morgan's rule, giving the conclusion

  R10 : ¬ P1,2 ∧ ¬ P2,1 .

That is, neither [1,2] nor [2,1] contains a pit.

We just need to define a proof problem as follows: •

- INITIAL STATE : the initial knowledge base.
- ACTIONS : the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove.

One final property of logical systems is **monotonicity,** which says that the set of entailed sentences can only increase as information is added to the knowledge base.

For any sentences α and β, **if KB |= α then KB ∧ β | = α.**

Proof by resolution

The current section introduces a single inference rule, resolution, that yields a complete inference algorithm when coupled with any complete search algorithm.

We begin by using a simple version of the resolution rule in the **WUMPUS** world.

- The agent returns from [2,1] to [1,1] and then goes to [1,2], where it perceives a stench, but no breeze. We add the following facts to the knowledge base:

  R11 : ¬ B1,2

  R12 : B1,2 ⇔ (P1,1 ∨ P2,2 ∨ P1,3) .

- By the same process that led to R10 earlier, we can now derive the absence of pits in [2,2] and [1,3] (remember that [1,1] is already known to be pit less):

  R13 : ¬ P2,2

  R14 : ¬ P1,3 .

- We can also apply biconditional elimination to R3, followed by Modus Ponens with R5, to obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

  R15 : P1,1 ∨ P2,2 ∨ P3,1 .

- Now comes the first application of the resolution rule: the literal ¬ P2,2 in R13 resolves with the literal P2,2 in R15 to give the **resolvent**

  R16 : P1,1 ∨ P3,1 .

- In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1] or [3,1]. Similarly, the literal ¬ P1,1 in R1 resolves with the literal P1,1 in R16 to give

  R17 : P3,1 .

- In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]. These last two inference steps are examples of the unit resolution inference rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k},$$

  where each l is a literal and li and m are complementary literals

Note that a single literal can be viewed as a disjunction of one literal, also known as a unit clause.

The unit resolution rule can be generalized to the full resolution rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

where li and mj are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses except the two complementary literals. For example, we have

$$\frac{P_{1,1} \vee P_{3,1}, \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}} .$$

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.9 The removal of multiple copies of literals is called **factoring.** For example, if we resolve (A ∨ B) with (A ∨ ¬ B), we obtain (A ∨ A), which is reduced to just A.

Conjunctive normal form

Every sentence of propositional logic is logically equivalent to a conjunction of clauses.

A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form or CNF**

We illustrate the procedure by converting the sentence B1,1 ⇔ (P1,2 ∨ P2,1) into CNF. The steps are as follows:

1. Eliminate ⇔, replacing α ⇔ β with (α ⇒ β) ∧ (β ⇒ α).

   (B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .

2. Eliminate ⇒, replacing α ⇒ β with ¬ α ∨ β:

   (¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬ (P1,2 ∨ P2,1) ∨ B1,1) .

3. CNF requires ¬ to appear only in literals, so we "move ¬ inwards" by repeated application of the following equivalences from **Figure 2.22**

   ¬ (¬ α) ≡ α (double-negation elimination)

   ¬ (α ∧ β) ≡(¬ α ∨ ¬ β) (DeMorgan)

   ¬ (α ∨ β) ≡(¬ α ∧ ¬ β) (DeMorgan)

   In the example, we require just one application of the last rule:

   (¬ B1,1 ∨ P1,2 ∨ P2,1) ∧ ((¬ P1,2 ∧¬ P2,1) ∨ B1,1) .

4. Now we have a sentence containing nested ∧ and ∨ operators applied to literals. We apply the distributivity law from **Figure 2.22**, distributing ∨ over ∧ wherever possible.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) .$$

The original sentence is now in CNF, as a conjunction of three clauses.