



RV Institute of Technology
and Management®

AI-ML

Module 3



RV Institute of Technology
and Management®

Outline

- Informed Search Strategies
 - Greedy best-first search
 - A* search
 - Heuristic functions



RV Institute of Technology
and Management®

Module-3

Problem-solving: Informed Search Strategies, Heuristic functions

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

Chapter 3 - 3.5, 7.6

Chapter 7 - 7.1, 7.2, 7.3, 7.4



RV Institute of Technology
and Management®

INFORMED (HEURISTIC) SEARCH STRATEGIES

One that uses problem-specific knowledge beyond the definition of the problem itself can find solutions more efficiently than an uninformed strategy



BEST-FIRST SEARCH

Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, $f(n)$.

The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.

The implementation of best-first graph search is identical to that for uniform-cost search

The choice of f determines the search strategy.

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.

if n is a goal node, then $h(n)=0$.

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.



Greedy best-first search

Greedy best-first search tries to **expand the node that is closest to the goal**, on the grounds that this is likely to lead to a solution quickly.

It evaluates nodes by using just the heuristic function; that is, **$f(n) = h(n)$** .



RV Institute of Technology
and Management®

Greedy best-first search- Route finding problems in Romania

straight line distance heuristic, H_{sld}

If the goal is Bucharest, we need to know the straight-line

distances to Bucharest h_{SLD} is correlated with actual road distances

and is, therefore, a useful heuristic

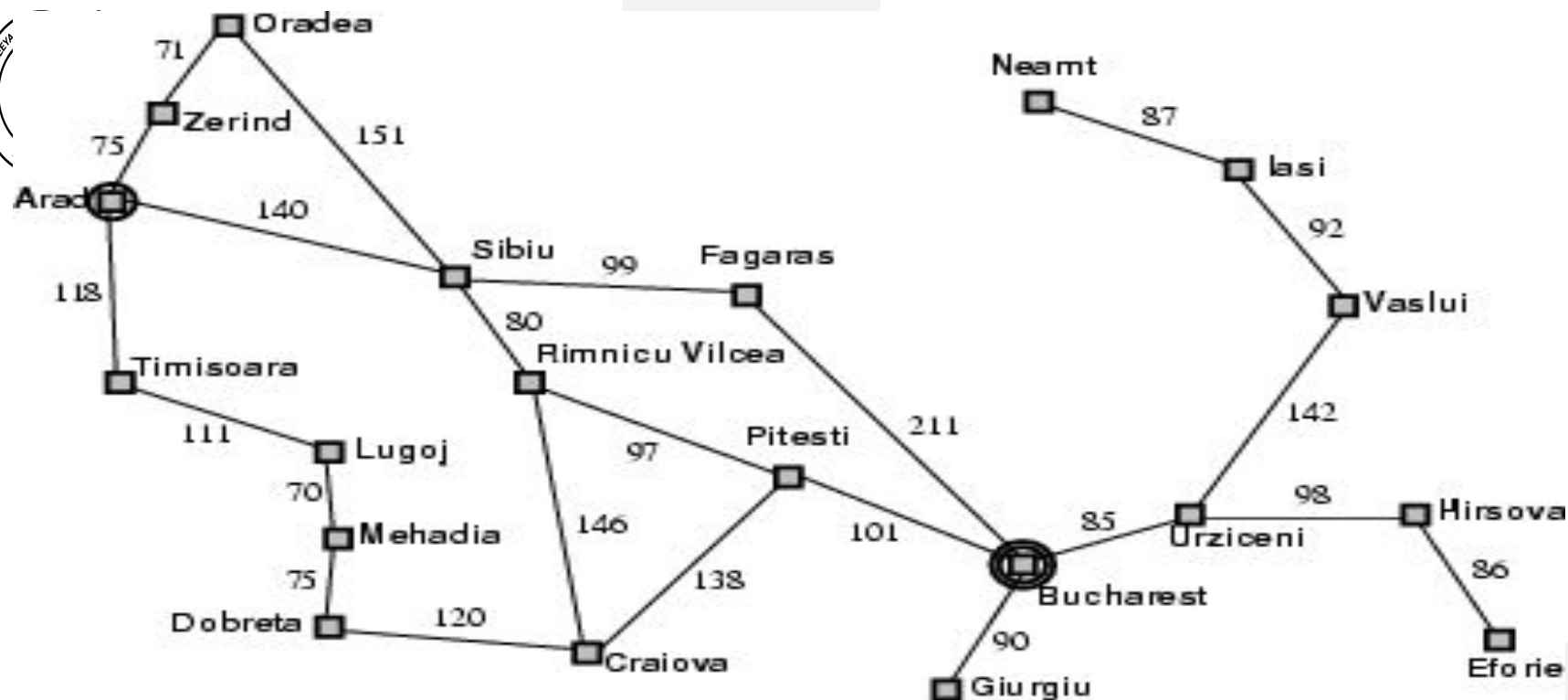
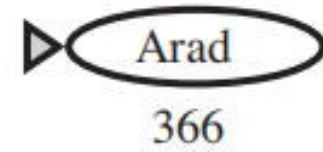


Figure 3.22 Values of h_{SLD} —straight-line distances to Bucharest.



RV Institute of Technology
and Management®

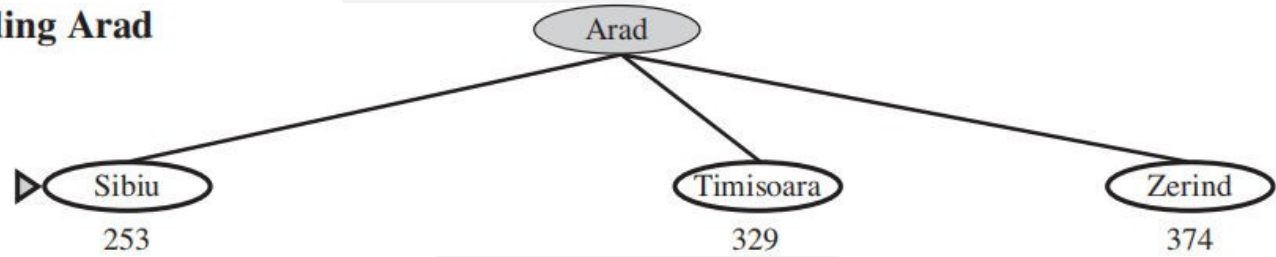
(a) The initial state





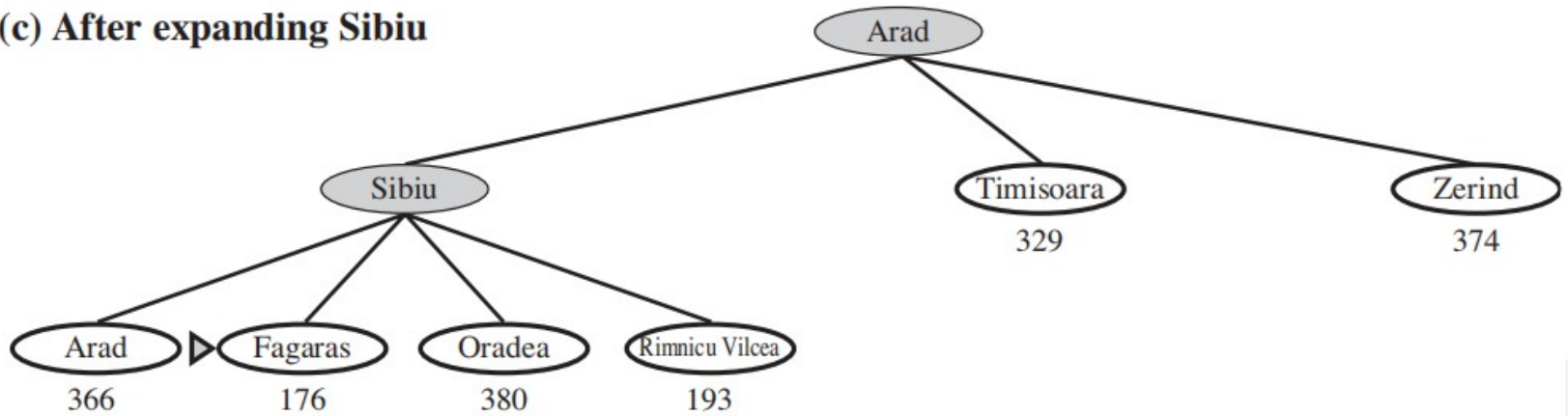
RV Institute of Technology
and Management®

(b) After expanding Arad





(c) After expanding Sibiu



(d) After expanding Fagaras

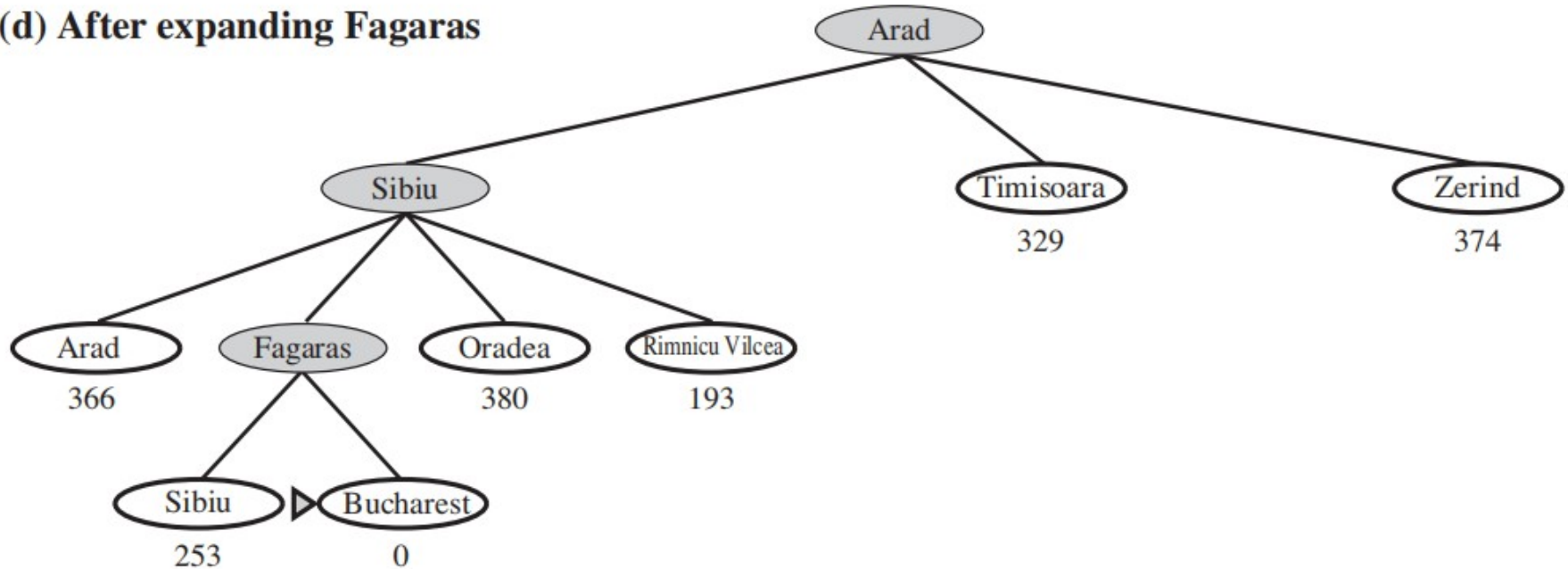
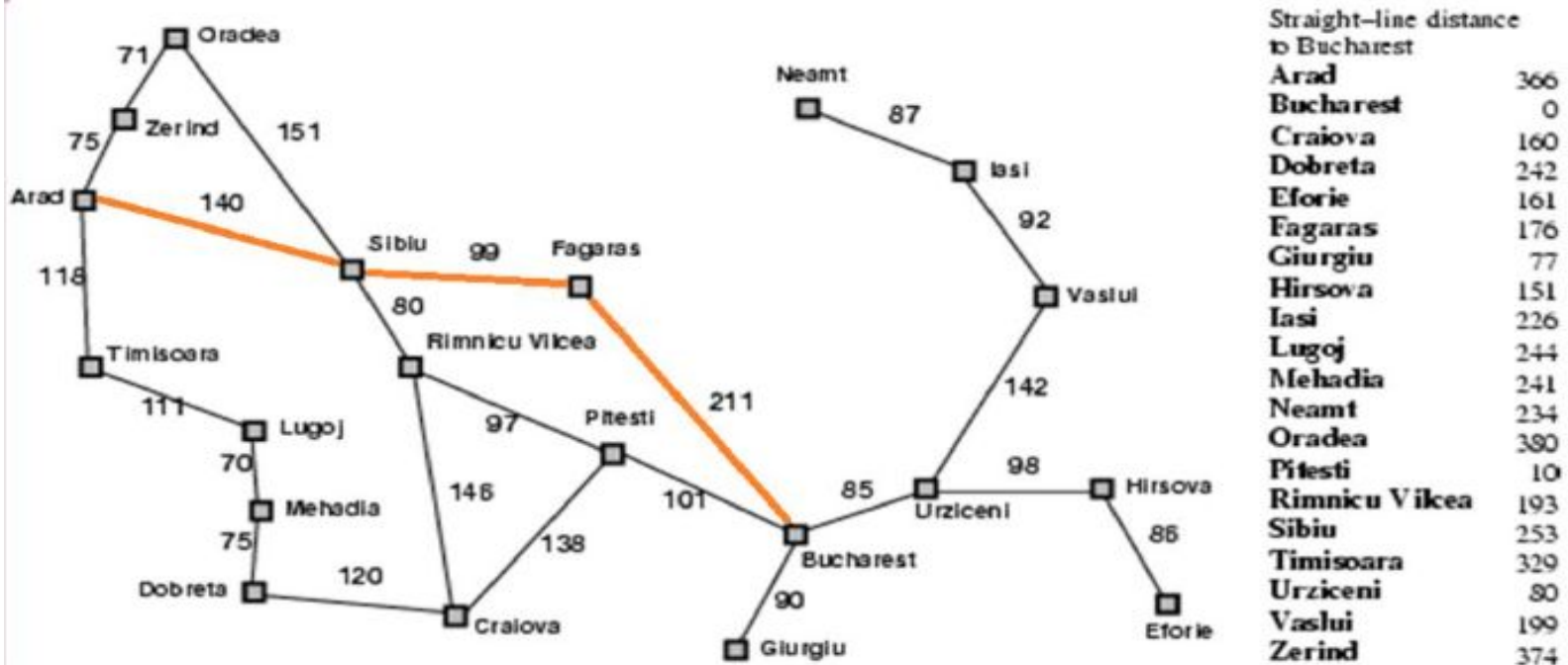


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD} . Nodes are labeled with their h -values.



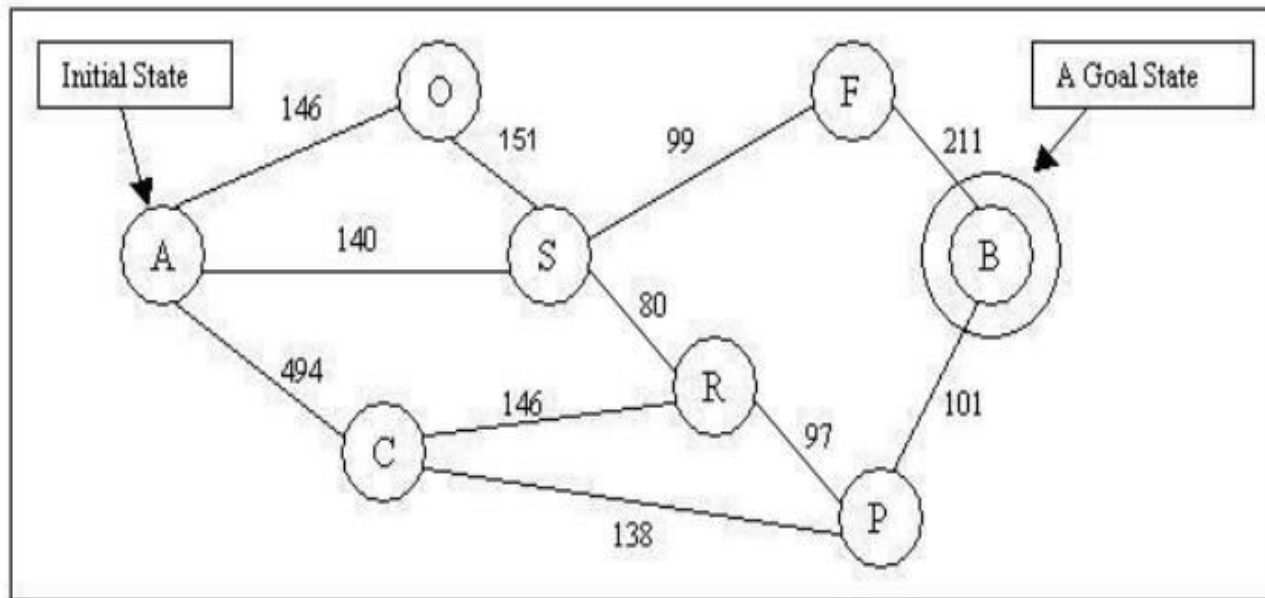
Stages in a greedy search for Bucharest, using the straight-line distance to Bucharest



Arad → Sibiu → Fagaras → Bucharest



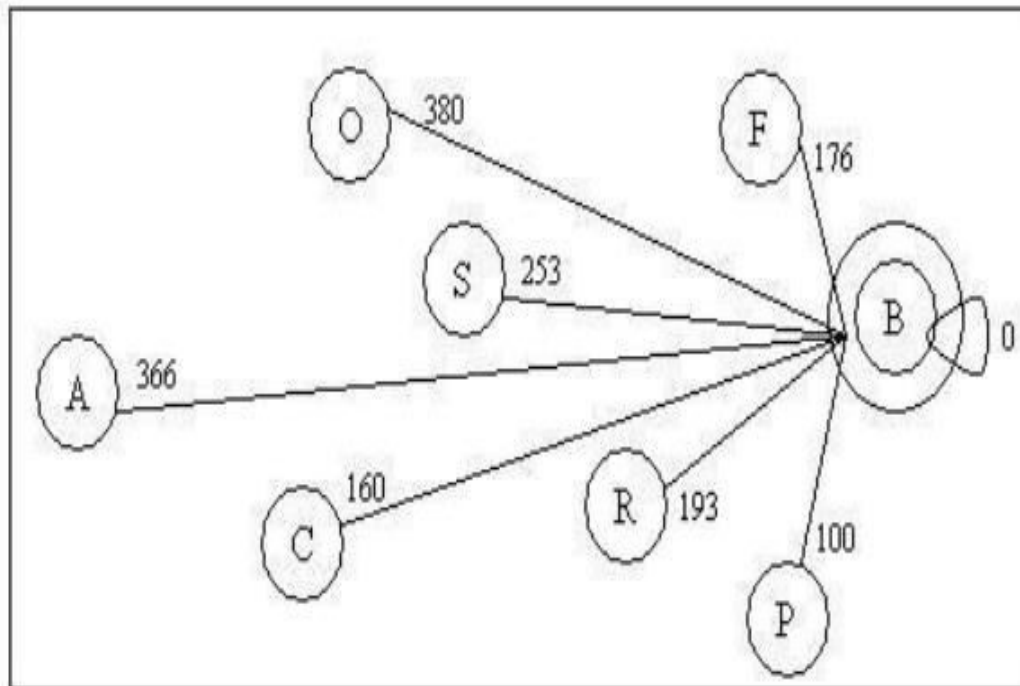
Now let's use an example to see how greedy best-first search works. Below is a map that we are going to search the path on. For this example, let the valuation function $f(n)$ simply be equal to the heuristic function $h(n)$.





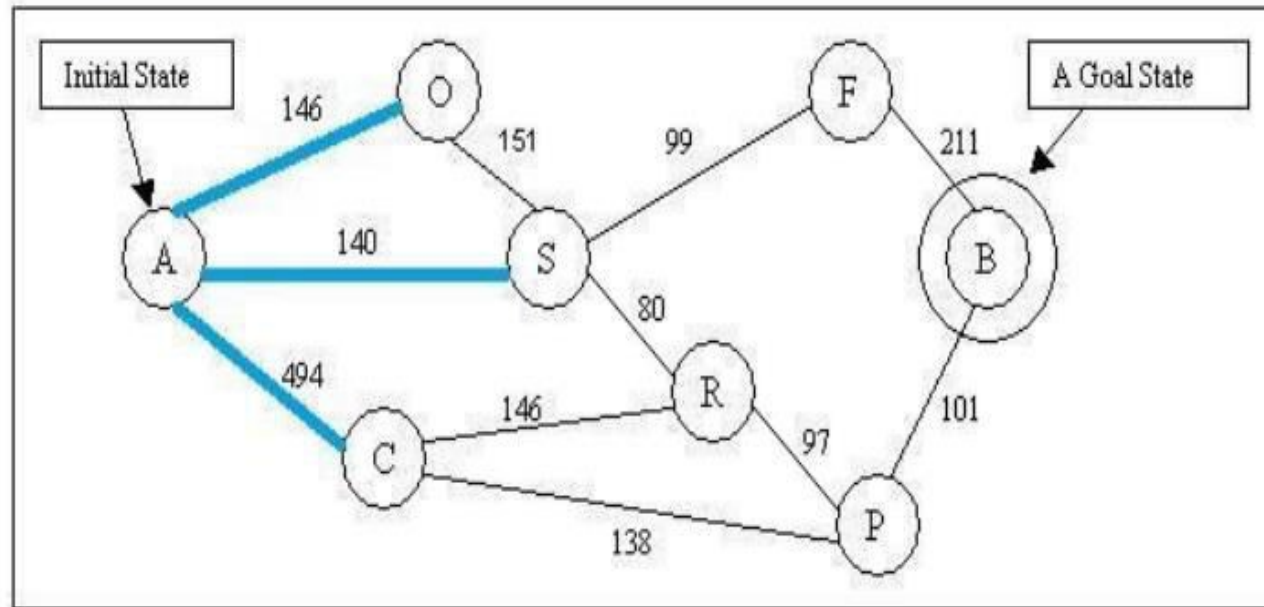
The heuristic function; that is, $f(n) = h(n)$.

This is a map of a city. We are going to find out a path from city A to city B.



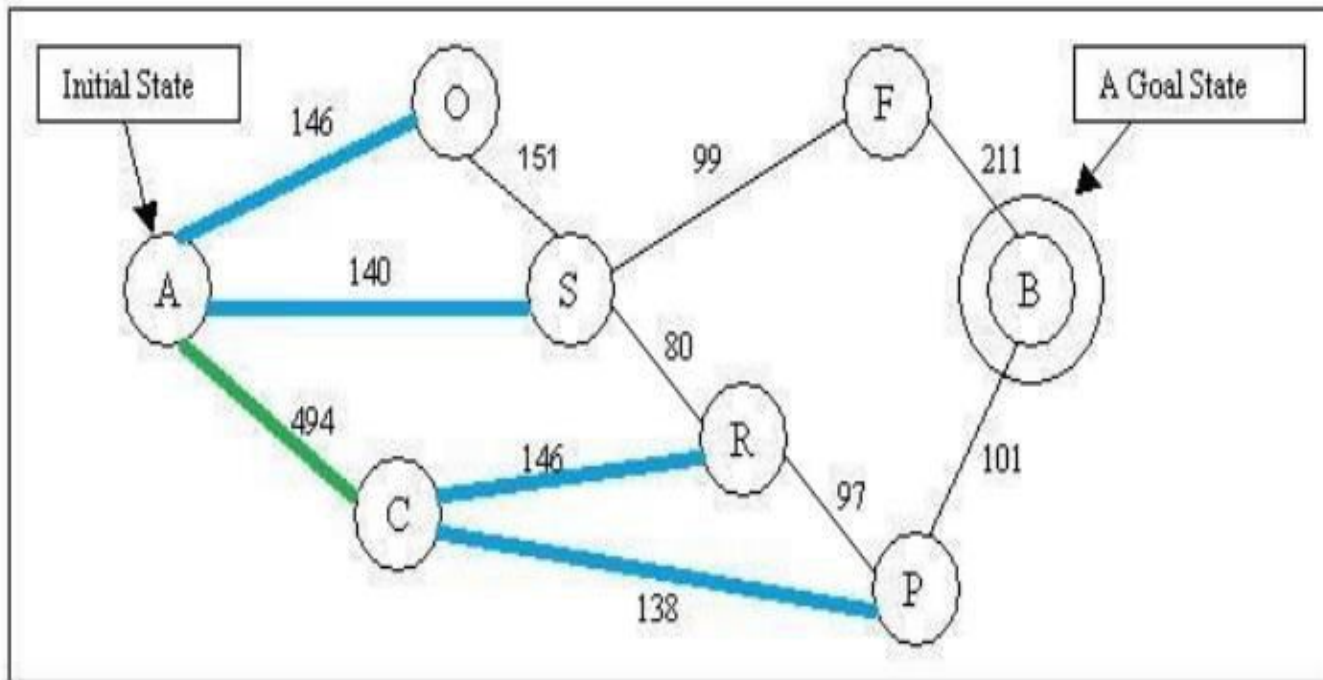


This graph shows the straight distance from each city to city B. The straight distances serve as each city's $h(n)$ in this example. Because $f(n) = h(n)$ our algorithm will at every stage choose to explore the node that we know is closest to our goal. Let's assume cities on the map as nodes and path between cities as edges. If you start exploring at node A, we have node O, S and C reachable now.



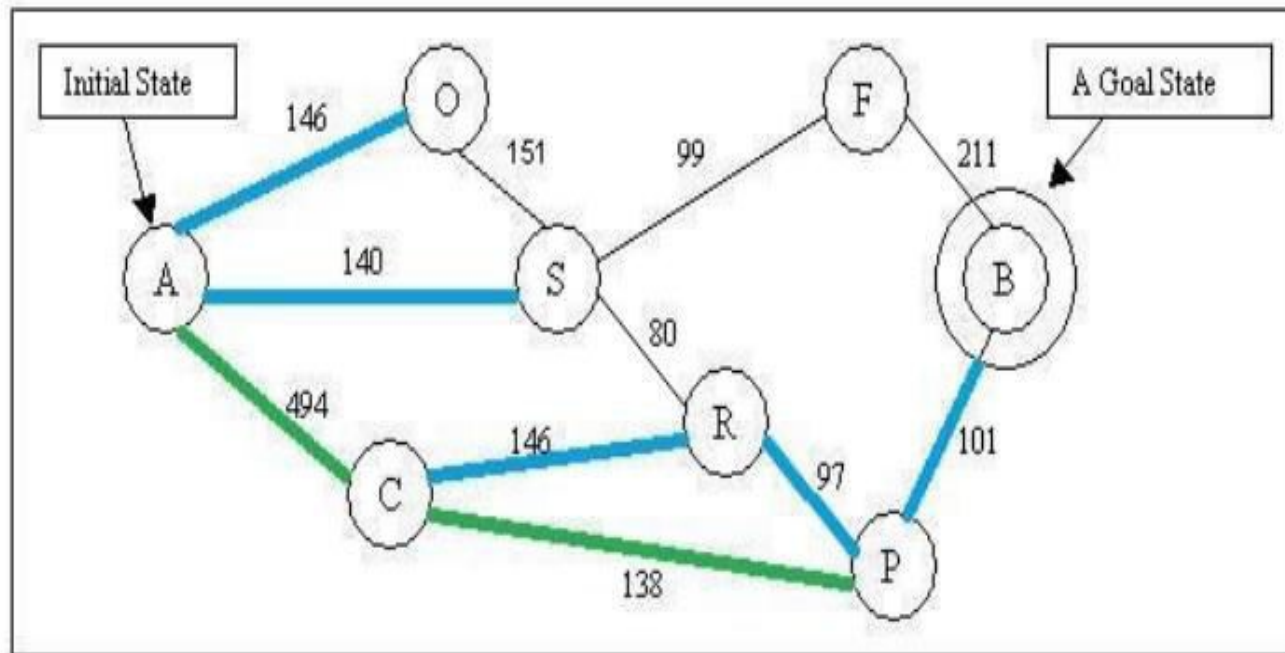


According to the estimate function $f(n) = h(n)$, and given by the graph of $h(n)$ values above: Node O's $f(n) = 380$, Node S' $f(n) = 253$, Node C's $f(n) = 160$. The algorithm will choose to explore node C. After explored C, we have new node R and P reachable now.

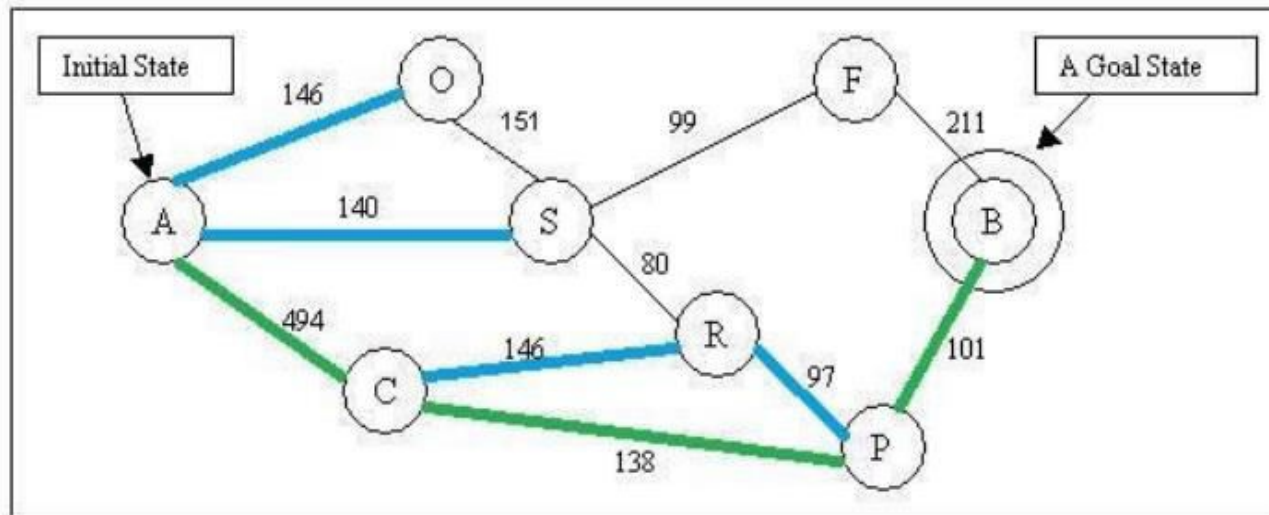




According to the estimate function $f(n)$: Node O's $f(n) = 380$, Node S' $f(n) = 253$, Node R's $f(n) = 193$, Node P's $f(n) = 100$. The algorithm will choose node P to explore. After explored C, we have new node B reachable now.



According to the estimate function $f(n)$: Node O's $f(n) = 380$, Node S' $f(n) = 253$, Node R's $f(n) = 193$, Node B's $f(n) = 0$. The algorithm will choose node B to explore. After explored B, we have reached our goal state. The algorithm will be stopped and the path is found.



Note that in this example, the distance between the current node and next node does NOT inform our next step, only the heuristic given by the distance between potential nodes to the goal.

One can generalize the evaluation function of a target node to be a weighted sum of the heuristic function and the distance from the current node to that target, which could produce a different result.



Properties of Greedy Best First Search

- Greedy search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end.
- It suffers from the same defects as depth-first search i.e.,
 - ✓ It is not optimal
 - ✓ It is incomplete because it can start down an infinite path and never return to try other possibilities.



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

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



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.



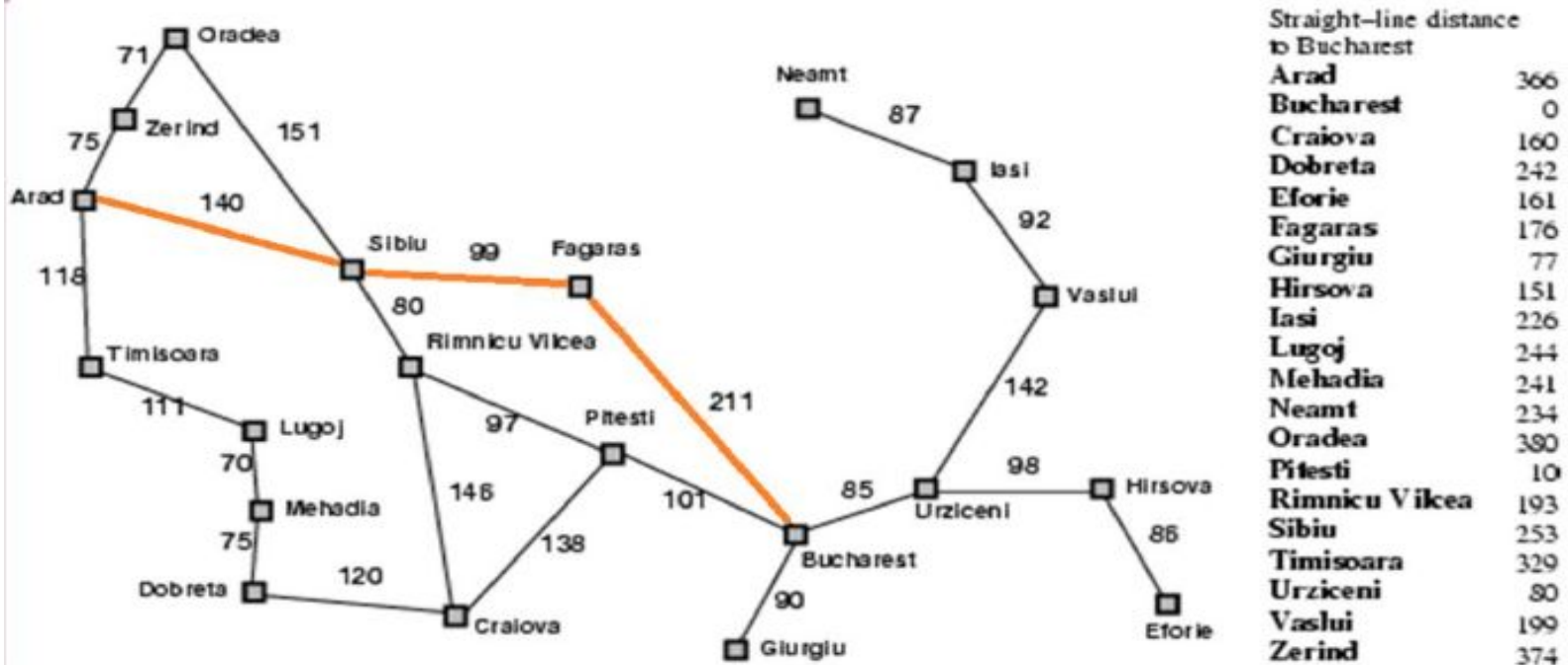
A* search strategy

A* SEARCH

- The most widely known form of best-first search is called A* search ("A-star search").
- Idea: **avoid expanding paths that are already expensive**
- **Evaluation function $f(n) = g(n) + h(n)$**
 - $g(n)$ = cost so far to reach n
 - $h(n)$ = estimated cost from n to goal
 - $f(n)$ = estimated total cost of path (i.e. cheapest solution) through n to goal



Progress of an A* tree search



Arad → Sibiu → Fagaras → Bucharest

Progress of an A* tree search

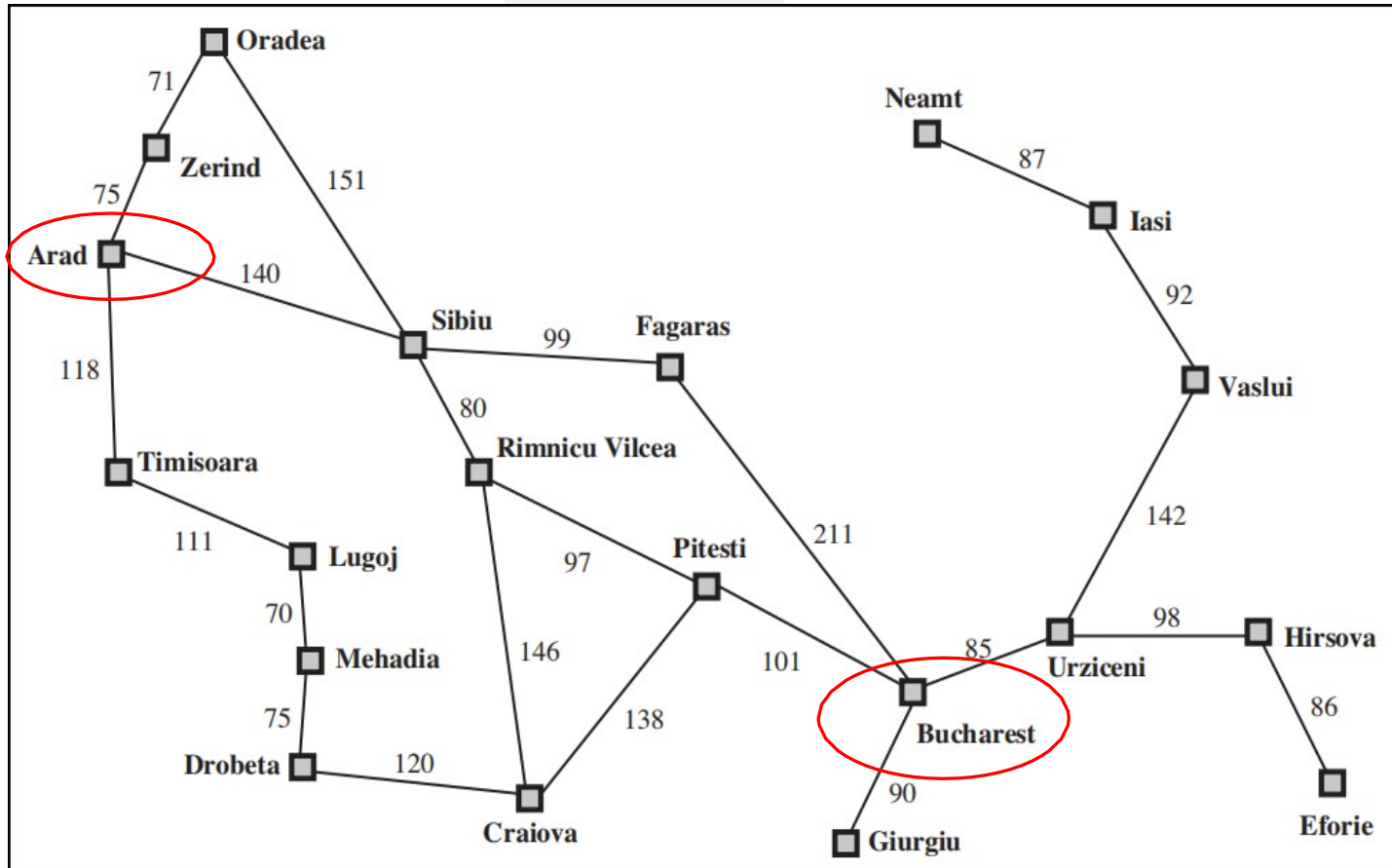


Figure 3.2 A simplified road map of part of Romania.

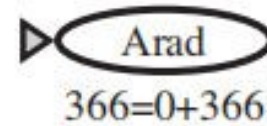


RV Institute of Technology
and Management®

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:

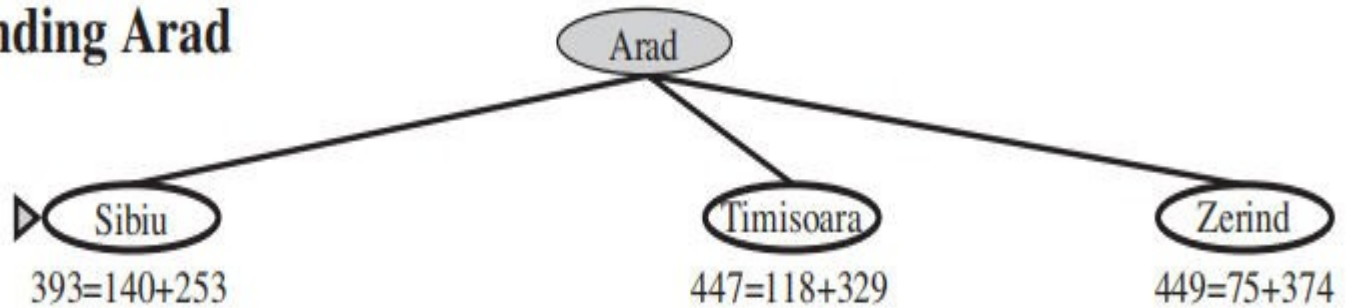
$$f(n) = g(n) + h(n) .$$

(a) The initial state



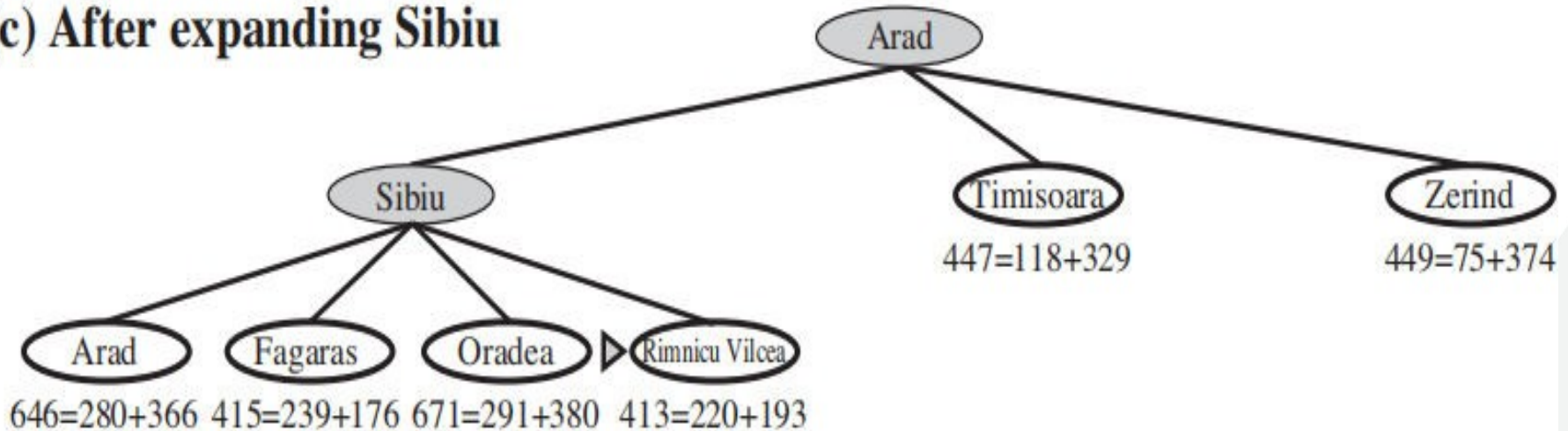


(b) After expanding Arad



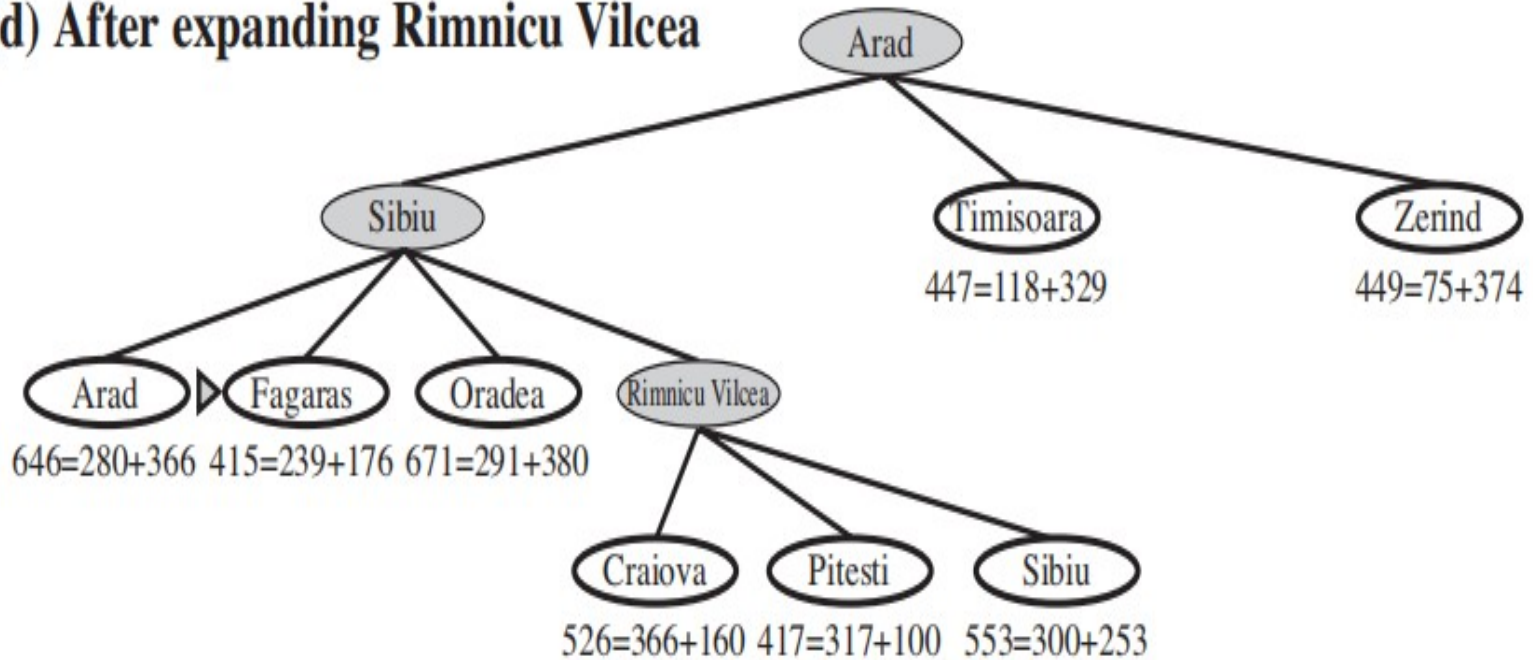


(c) After expanding Sibiu

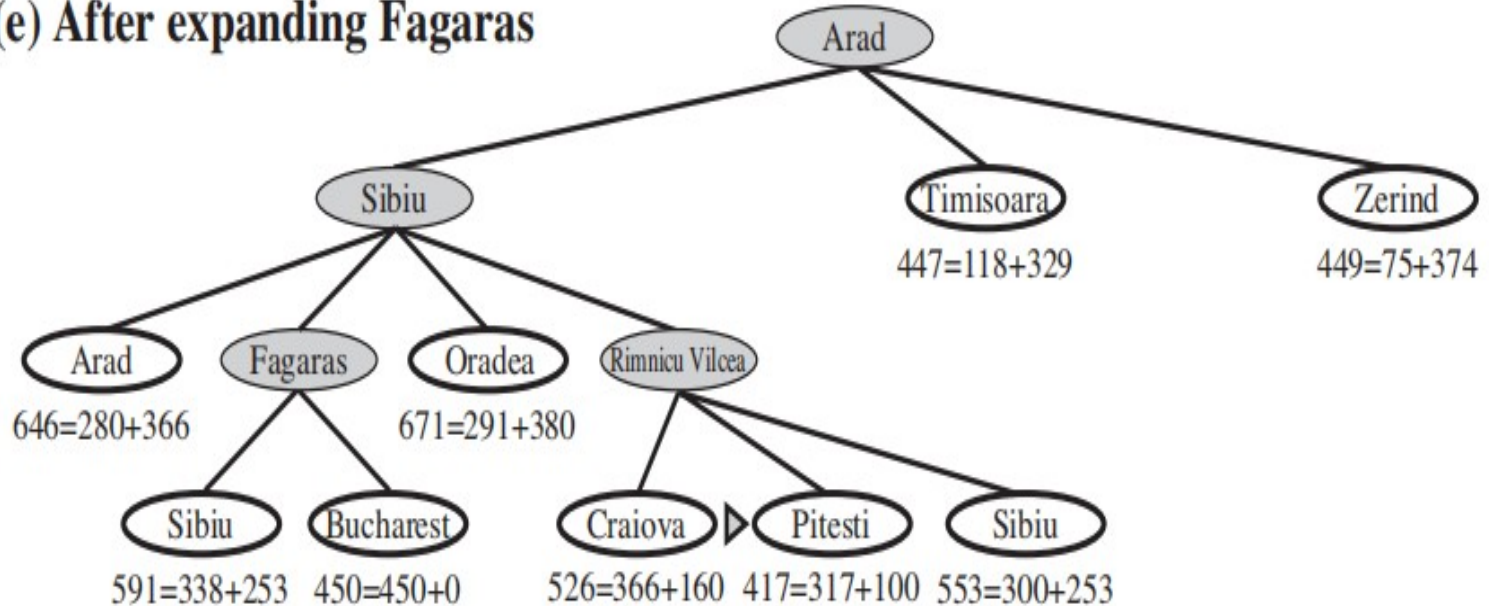




(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



Bucharest first appears on the frontier here 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.

(f) After expanding Pitesti

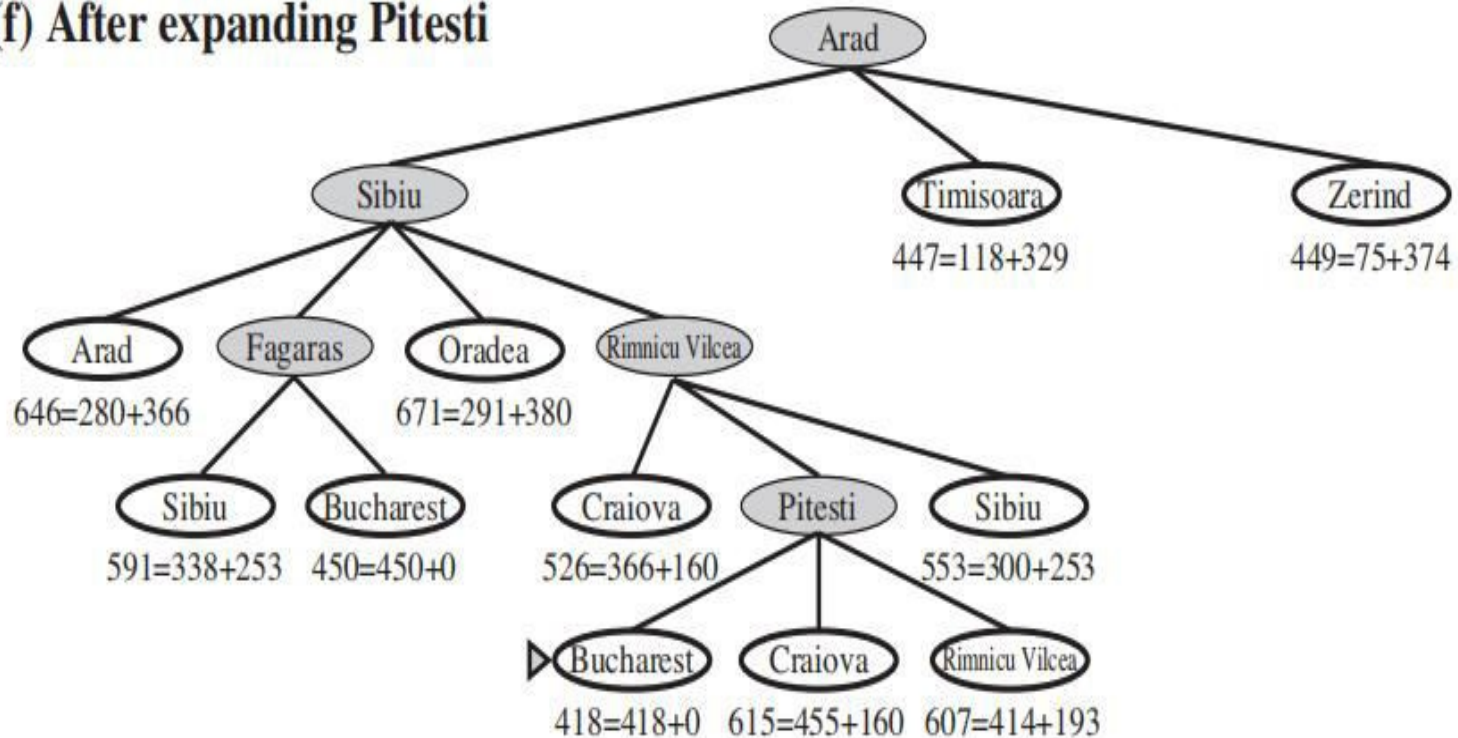
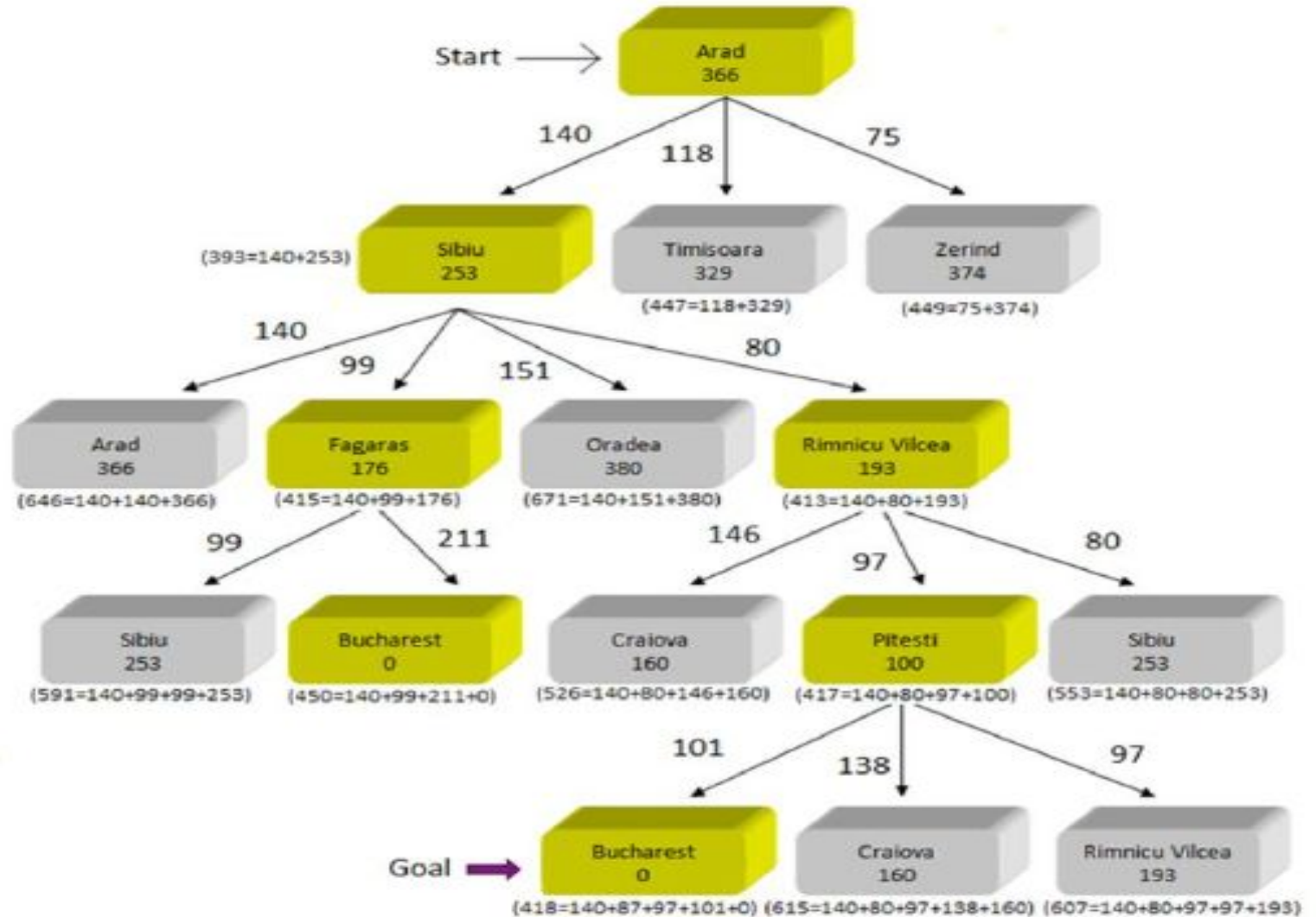


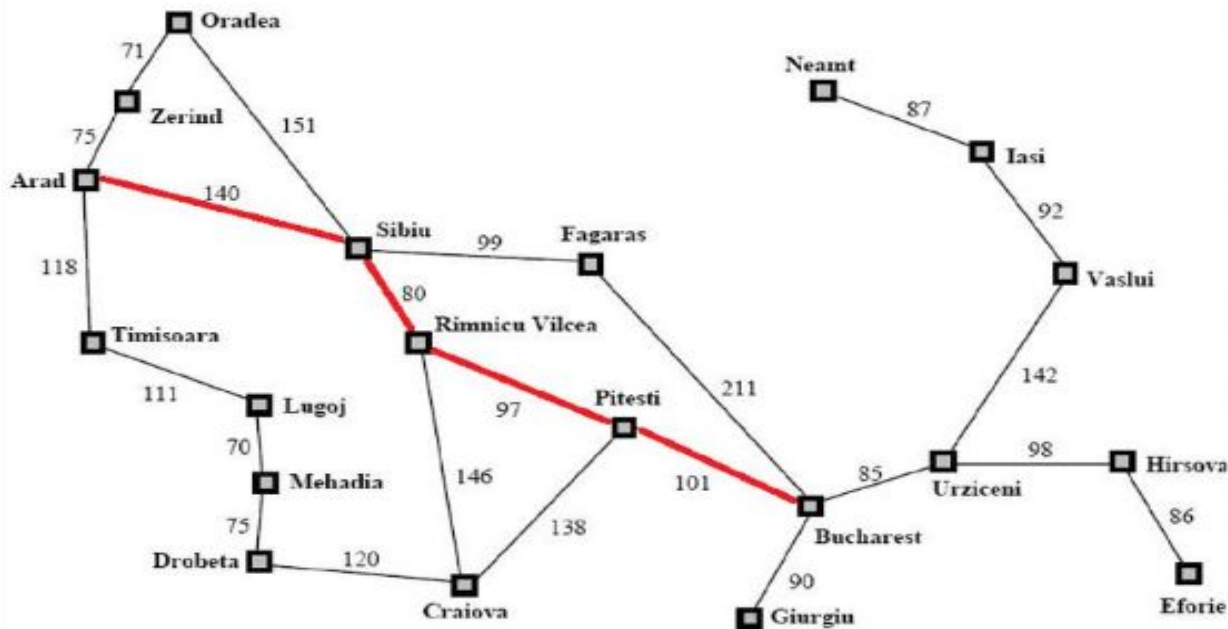
Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

A* SEARCH EXAMPLE





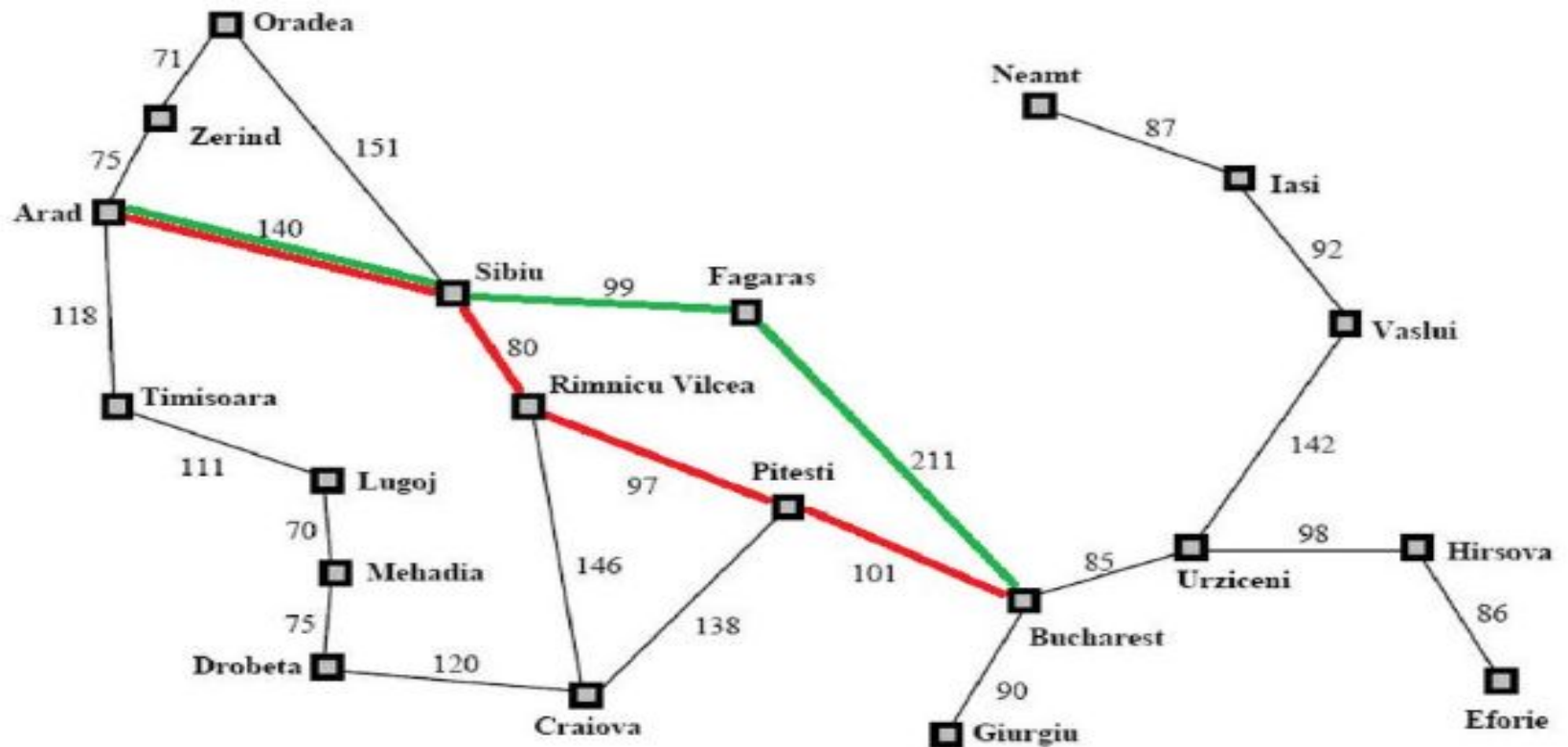
A* SEARCH EXAMPLE



Straight-line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

COMPARING A* AND GREEDY SEARCH



— greedy search path → $140 + 99 + 211 = 450$

— A* search path → $140 + 80 + 97 + 101 = 418$



Conditions for optimality: **Admissibility**

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. $g(n)$ 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

- Eg, **straight-line distance** h_{SLD} that we used in getting to Bucharest is **an admissible heuristic** because the shortest path between any two points is a straight line
- Straight line cannot be an **overestimate**



Conditions for optimality: Consistency

It is **required** only for applications of **A*** to graph search

A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' :

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

This is a form of the general **triangle inequality**, which stipulates that each side of a **triangle cannot be longer than the sum of the other two sides**

- Here, the triangle is formed by n , n' , and the goal 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' 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 .



RV Institute of Technology
and Management®

Optimality of A*

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**.*



Optimality of A^*

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.*

A^* expands no nodes with $f(n) > C^*$ —for example, Timisoara is not expanded in even though it is a child of the root

the subtree below Timisoara is **pruned**; because h SLD is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality

Pruning eliminates possibilities from consideration without having to examine them A^* is **optimally efficient** for any given consistent heuristic.

- That is, no other optimal algorithm is guaranteed to expand fewer nodes than A^*
- This is because any algorithm that *does not* expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution.



If $h(n)$ is consistent, then the values of $f(n)$ along any path are nondecreasing.

The proof follows directly from the definition of consistency.

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

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



whenever A^* selects a node n for expansion,
the optimal path to that node has been found.

- Were this not the case, there would have to be another frontier node n on the optimal path from the start node to n
- because f is nondecreasing along any path, n would have lower f -cost than n and would have been selected first



RV Institute of Technology
and Management®

the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing order of $f(n)$.

Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have $h = 0$) and all later goal nodes will be at least as expensive.

Contours in the state space

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.

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.

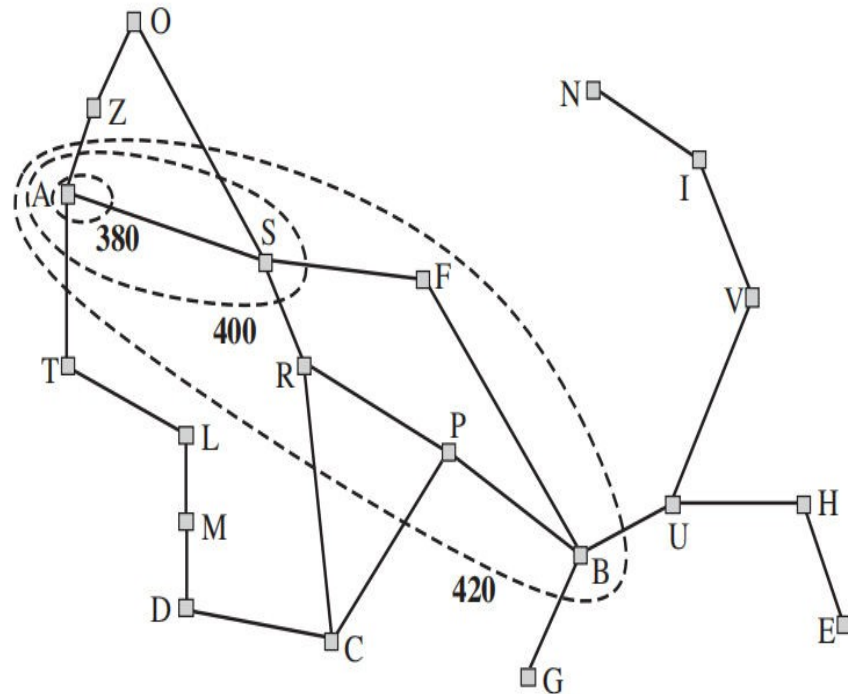


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f -costs less than or equal to the contour value.



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.



RV Institute of Technology
and Management®

Disadvantages of A* Algorithm

The number of states within the goal contour search space is still exponential in the length of the solution.



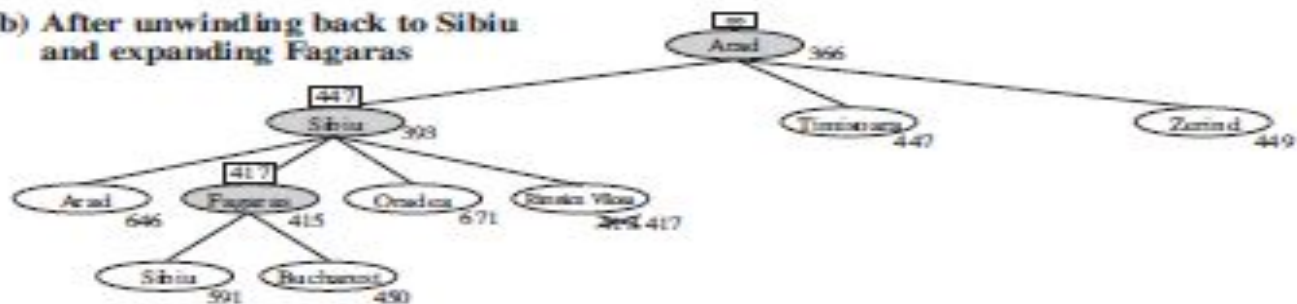
Memory-bounded heuristic search

- 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.
- **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.

(a) After expanding Arad, Sibiu, and Rimnicu Vilcea



(b) After unwinding back to Sibiu and expanding Fagaras



(c) After switching back to Rimnicu Vilcea and expanding Pitesti

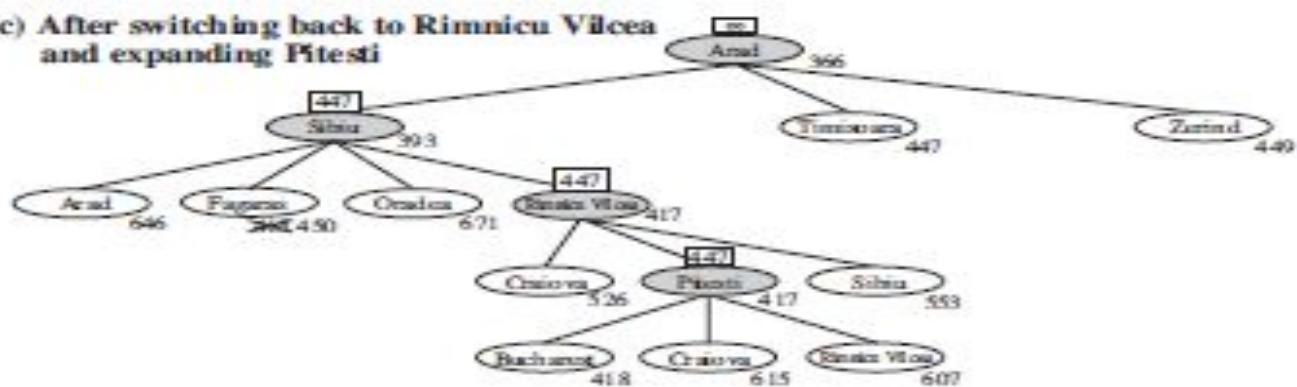
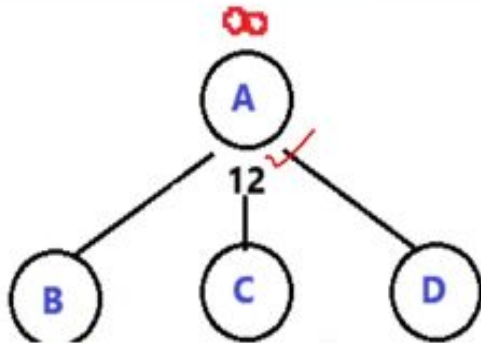


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f -limit value for each recursive call is shown on top of each current node, and every node is labeled with its f -cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is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

Recursive Best First Search Solved Example 1 ⁱ



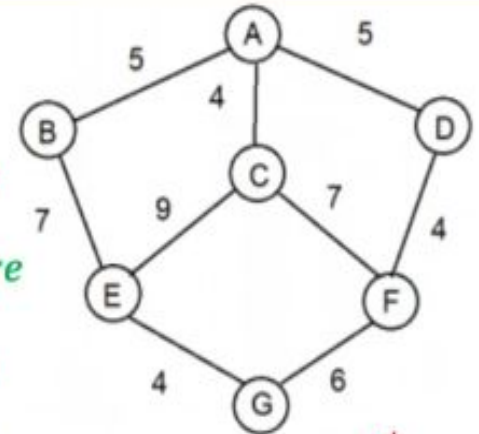
$$s.f = \max(s.g + s.h, \text{node}.f)$$

Best = best successor node

Best.f > f_limit return failure

Alt = second best succ node

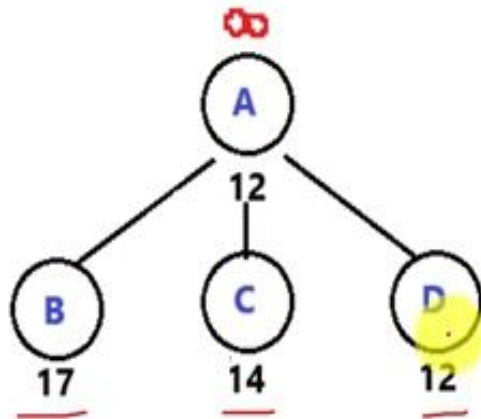
$$f_limit = \min(f_limit, \text{alt}.f)$$



A	12 ✓
B	12
C	10
D	7
E	2
F	6
G	0



Recursive Best First Search Solved Example 1



$$s.f = \max(s.g + s.h, \text{node}.f)$$

Handwritten notes: 14, 12

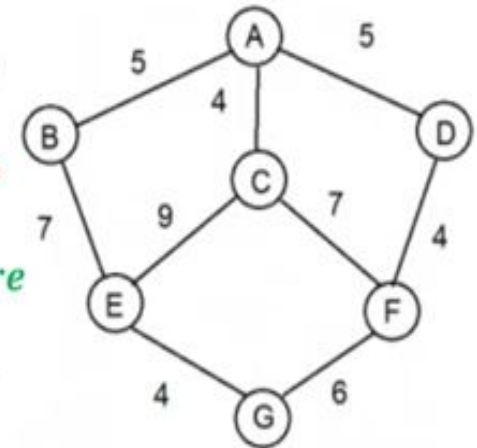
Best = best successor node

Best.f ¹² > f_limit return failure

Alt = second best succ node

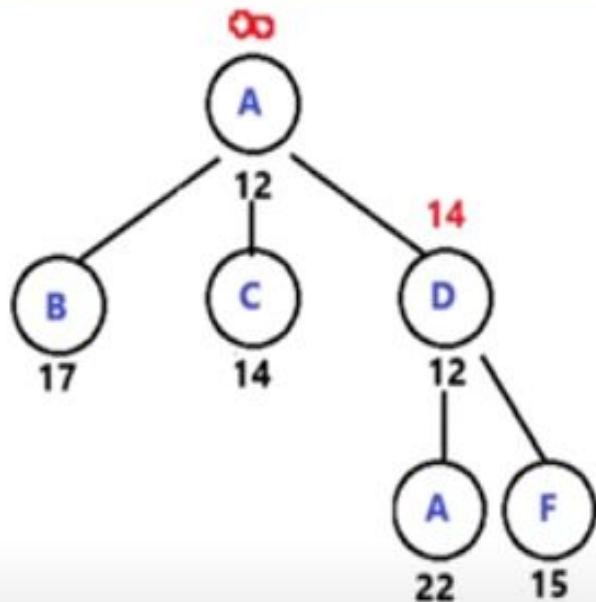
$$\text{f_limit} = \min(\text{f_limit}, \text{alt.f})$$

Handwritten notes: ∞, 14



A	12
B	12
C	10
D	7
E	2
F	6
G	0

Recursive Best First Search Solved Example 1



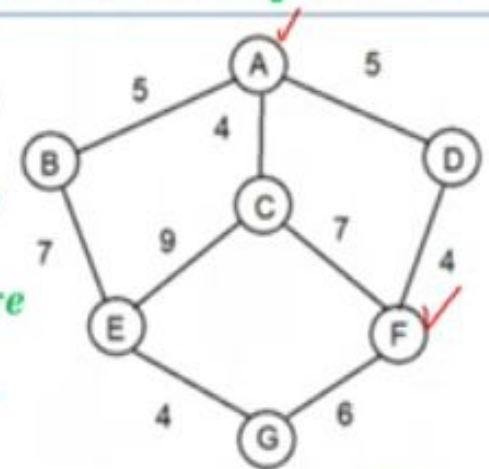
$$s.f = \max(s.g + s.h, \text{node}.f)$$

Best = best successor node

Best.f > f_limit return failure

Alt = second best succ node

$$f_limit = \min(f_limit, \text{alt}.f)$$



A	12
B	12
C	10
D	7
E	2
F	6
G	0



Learning to search better

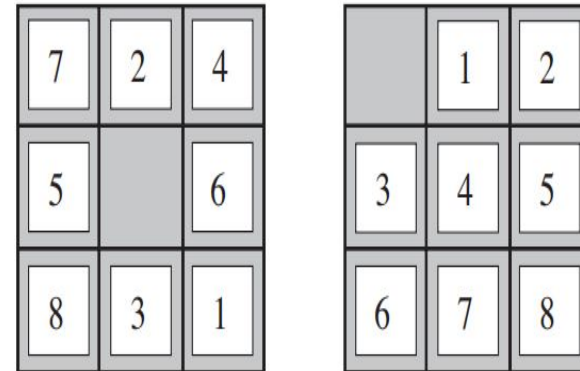
- presented several fixed strategies—breadth-first, greedy best-first, and so on—that have been designed by computer scientists.
- Could an agent *learn* how to search better? The answer is yes, and the method rests on an important concept called the metalevel state space.
- Each state in a metalevel state space captures the internal (computational) state of a program that is searching in an object-level state space such as Romania.
- For example, the internal state of the A* algorithm consists of the current search tree. Each action in the metalevel state space is a computation step that alters the internal state; for example, each computation step in A* expands a leaf node and adds its successors to the tree.
- Thus, Figure 3.24, which shows a sequence of larger and larger search trees, can be seen as depicting a path in the metalevel state space where each state on the path is an object-level search tree.
- Now, the path in Figure 3.24 has five steps, including one step, the expansion of Fagaras, that is not especially helpful. For harder problems, there will be many such missteps, and a metalevel learning algorithm can learn from these experiences to avoid exploring unpromising subtrees. The techniques used for this kind of learning are described in Chapter 21. The goal of learning is to minimize the total cost of problem solving, trading off computational expense and path cost.



HEURISTIC FUNCTIONS

•8-puzzle search space

- Typical solution length: 20 steps
- Average branching factor: 3
- Exhaustive search to depth 22: $3^{22} = 3.1 \times 10^{10}$ 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.



Start State

Goal State

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



7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

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



Admissible Heuristics

- h_1 = the number of misplaced tiles. For Figure 3.28, all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.
- h_2 = 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**. h_2 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
 - $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$



Heuristic 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,

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



Experimental Results on 8-puzzle problems

	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 . The results suggest that h_2 is better than h_1 , and is far better than using iterative deepening search.



Quality of Heuristics

Since A^* expands all nodes whose f value is less than that of an optimal solution, it is always better to use a heuristic with a higher value as long as it does not over-estimate.

Therefore h_2 is uniformly better than h_1 , or h_2 dominates h_1 .

A heuristic should also be easy to compute, otherwise the overhead of computing the heuristic could outweigh the time saved by reducing search (e.g. using full breadth-first search to estimate distance wouldn't help)



Inventing Heuristics

Many good heuristics can be invented by considering relaxed versions of the problem (abstractions). A problem with fewer restrictions on the actions is called a **relaxed problem**.

For 8-puzzle: A tile can move from square A to B if A is horizontally or vertically adjacent to B **and** B is blank

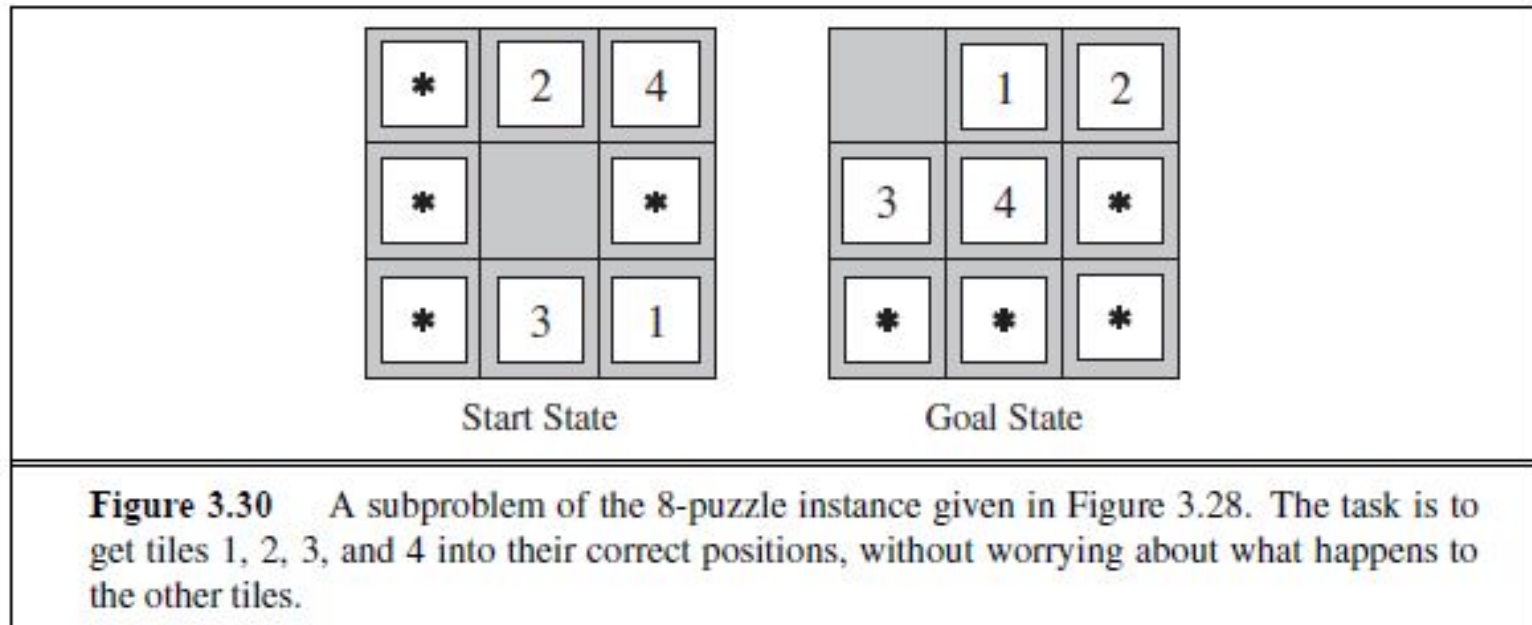
we can generate three relaxed problems by removing one or both of the conditions:

- (a) A tile can move from square A to B if A is adjacent to B.
- (b) A tile can move from square A to B if B is blank.
- (c) A tile can move from square A to B.

If there are a number of features that indicate a promising or unpromising state, a weighted sum of these features can be useful. Learning methods can be used to set weights.



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 in Figure 3.28.
- The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions.
- 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.



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?
- One solution was given in the preceding sections—namely, to devise relaxed problems for which an optimal solution can be found easily.
- Another solution is to learn from experience. “Experience” here means solving lots of 8-puzzles, for instance.
- Each optimal solution to an 8-puzzle problem provides examples from which $h(n)$ can be learned.
- Techniques for doing just this using neural nets, decision trees, and other methods are demonstrated in later chapters



Logical Agents

- Humans can know “things” and “reason”
 - Representation: How are the things stored?
 - Reasoning: How is the knowledge used?
 - To solve a problem...
 - To generate more knowledge...
- Knowledge and reasoning are important to artificial agents because they enable successful behaviors difficult to achieve otherwise
 - Useful in partially observable environments
- Can benefit from knowledge in very general forms, combining and recombining information



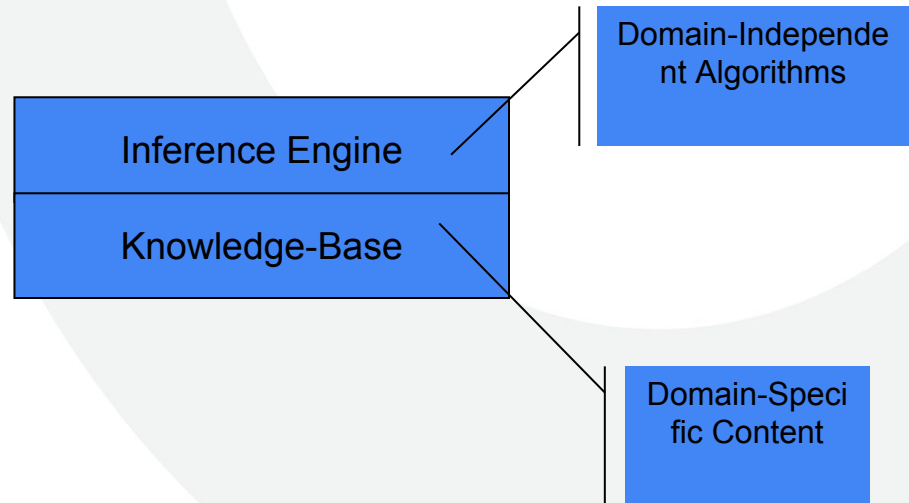
Knowledge-Based Agents

- Central component of a Knowledge-Based Agent is a **Knowledge-Base**
 - A set of sentences in a formal language
- Sentences are expressed using a knowledge representation language (represents some assertion about the world). Sometimes we dignify a sentence with the name axiom, when the sentence is taken as given without being derived from other sentences.
- Two generic functions:
 - TELL - add new sentences (facts) to the KB
 - “Tell it what it needs to know”
 - ASK - query what is known from the KB
 - “Ask what to do next”
- Both operations may involve **inference**—that is, deriving new sentences from old.



Knowledge-Based Agents

- The agent must be able to:
 - Represent states and actions
 - Incorporate new percepts
 - Update internal representations of the world
 - Deduce hidden properties of the world
 - Deduce appropriate actions





Knowledge-Based Agents

```
function KB-AGENT(percept) returns an action
  static: 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
```

First, it TELLS the knowledge base what it perceives. Second, it ASKS the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on.

Third, the agent program TELLS the knowledge base which action was chosen, and the agent executes the action.



- 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.
- MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed.
- The details of the inference mechanisms are hidden inside TELL and ASK



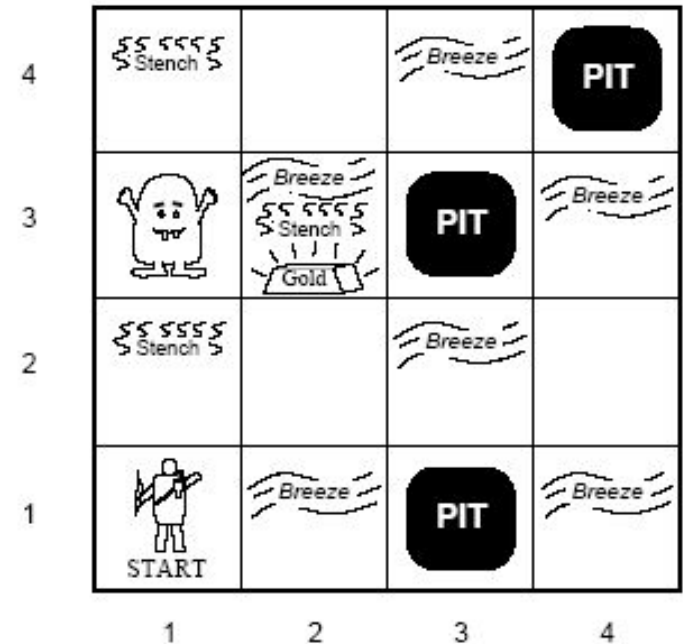
Knowledge-Based Agents

- Declarative
 - You can build a knowledge-based agent simply by “TELLing” it what it needs to know
- Procedural
 - Encode desired behaviors directly as program code
- A successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code



Wumpus World

- Performance Measure
 - Gold +1000, Death – 1000
 - Step -1, Use arrow -10
- Environment
 - Square adjacent to the Wumpus are **smelly**
 - Squares adjacent to the 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 the gold if in the same square
 - Releasing drops the gold in the same square
- Actuators
 - Left turn, right turn, forward, grab, release, shoot
- Sensors
 - Breeze, glitter, and smell
- See page 197-8 for more details!





Wumpus World

PEAS description

Performance measure:

- +1000 for gold
- 1000 for being eaten or falling down pit
- 1 for each action
- 10 for using the arrow

Environment:





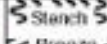




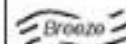




4×4 grid of "rooms", each "room" can be empty, with gold, occupied by Wumpus, or with a pit.

Actuators:

Move forward, turn left 90°, turn right 90°
Grab, shoot

Sensors:

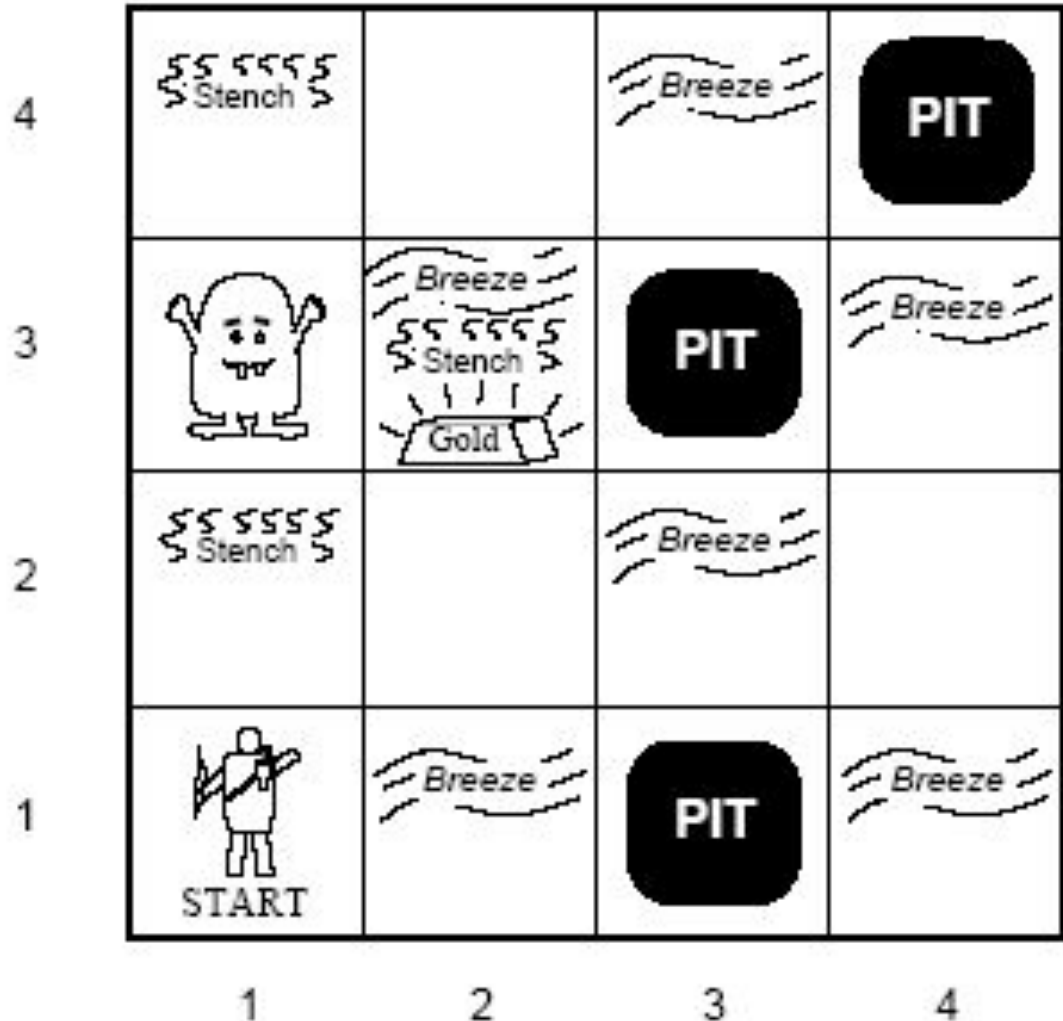
Olfactory – stench from Wumpus
Touch – breeze (pits) & hardness (wall)
Vision – see gold
Auditory – hear Wumpus scream when killed

4	 Stench		 Breeze	
3		 Stench  Gold		 Breeze
2	 Stench		 Breeze	
1		 Breeze		 Breeze
	1	2	3	4

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} \text{stench} \\ \text{breeze} \\ \text{wall} \\ \text{glitter} \\ \text{scream} \end{pmatrix}, x_i \in \{0,1\}$$



- **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.





Environment:

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



Actuators:

- The agent can move *Forward*, *TurnLeft* by 90° , or *TurnRight* by 90° .
- The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead 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**, **no bump**, or **no scream**, the agent program will get

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



RV Institute of Technology
and Management®

Wumpus world characterization

- Fully Observable
- Deterministic
- Episodic
- Static
- Discrete
- Single-agent?



RV Institute of Technology
and Management®

Fully Observable – does sensors give access to full state of the environment?



Wumpus world characterization

- Fully Observable No – only local perception
- Deterministic
- Episodic
- Static
- Discrete
- Single-agent?



RV Institute of Technology
and Management®

Deterministic — is the next state completely determined by the current state and the action executed by the agent or is there some uncertainty?



Wumpus world characterization

- Fully Observable No – only local perception
- Deterministic Yes – outcomes exactly specified
- Episodic
- Static
- Discrete
- Single-agent?



RV Institute of Technology
and Management®

Episodic (or sequential) – episodic means divided into independent atomic episodes (like an assembly line robot where decisions about the current part does not affect the decision on the next part) or Sequential – where current action affects future environments.



Wumpus world characterization

- Fully Observable No – only local perception
- Deterministic Yes – outcomes exactly specified
- Episodic No – sequential at the level of actions
- Static
- Discrete
- Single-agent?



RV Institute of Technology
and Management®

Static or dynamic? Does the world change as time goes by?



Wumpus world characterization

- Fully 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
- Single-agent?



Wumpus world characterization

- Fully 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?



Wumpus world characterization

- Fully 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



Wumpus World

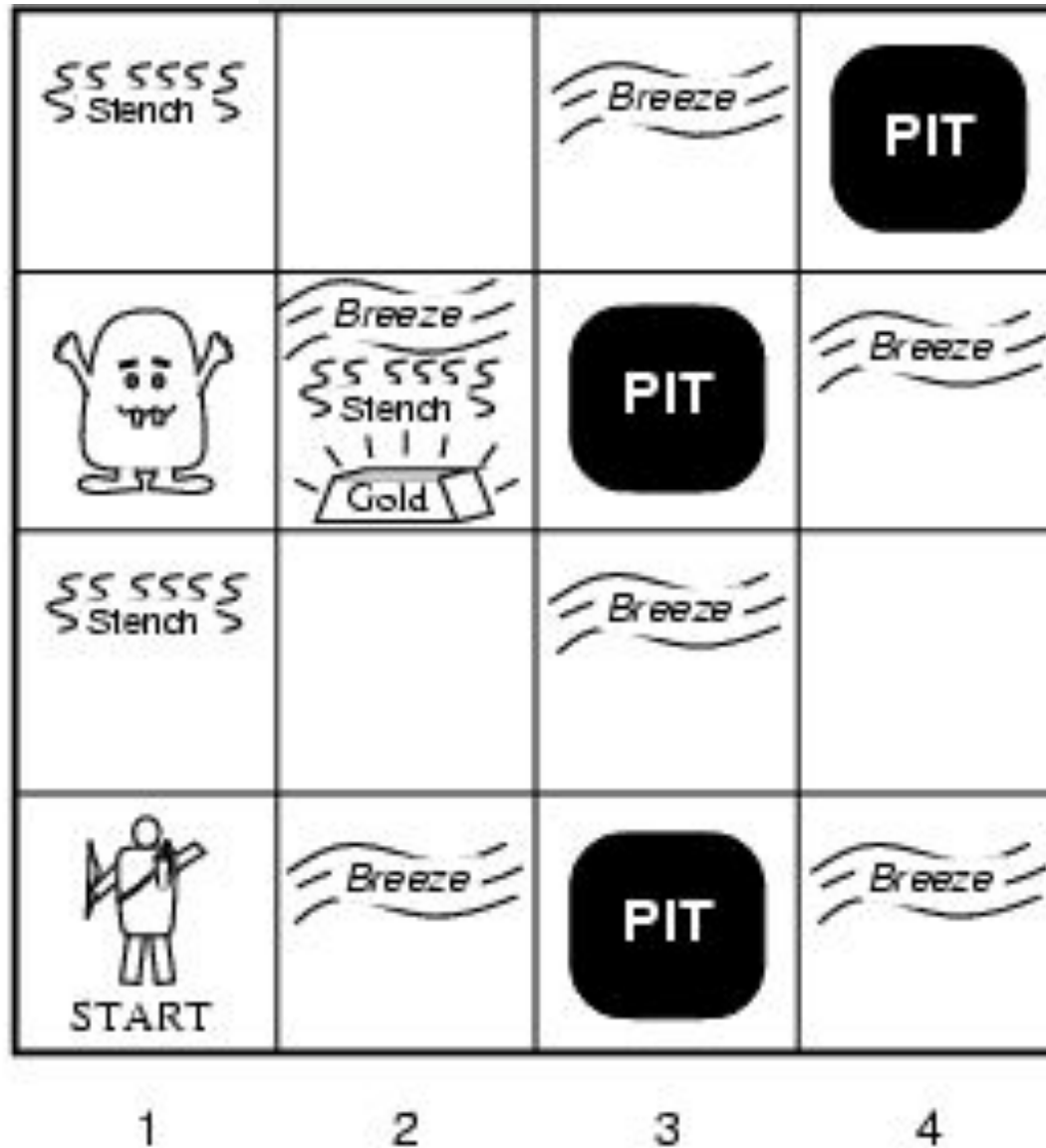
- Characterization of Wumpus World
 - Observable
 - partial, only local perception
 - Deterministic
 - Yes, outcomes are specified
 - Episodic
 - No, sequential at the level of actions
 - Static
 - Yes, Wumpus and pits do not move
 - Discrete
 - Yes
 - Single Agent
 - Yes



3

2

1



Principle Difficulty: agent is initially ignorant of the configuration of the environment – going to have to reason to figure out where the gold is without getting killed!



Exploring the Wumpus World

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 OK	2,2	3,2	4,2
1,1 A OK	2,1 OK	3,1	4,1

(a)

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

Initial situation:

**Agent in 1,1 and percept is
[None, None, None, None, None]**

**From this the agent can infer the
neighboring squares are safe
(otherwise there would be a breeze
or a stench)**

(b)

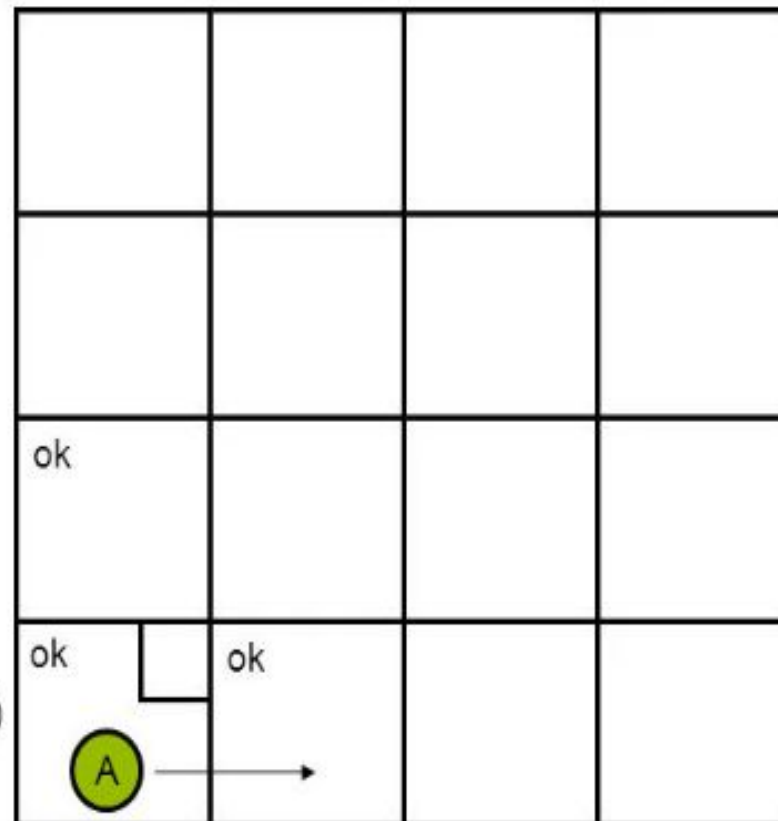


Exploring the Wumpus world



$$\mathbf{x}_{1,1} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Agent senses nothing
(no breeze, no smell,..)



[1,1] The KB initially
contains the rules of the
environment.

The first percept is [*none*,
none,none,none,none],

move to safe cell e.g. 2,1

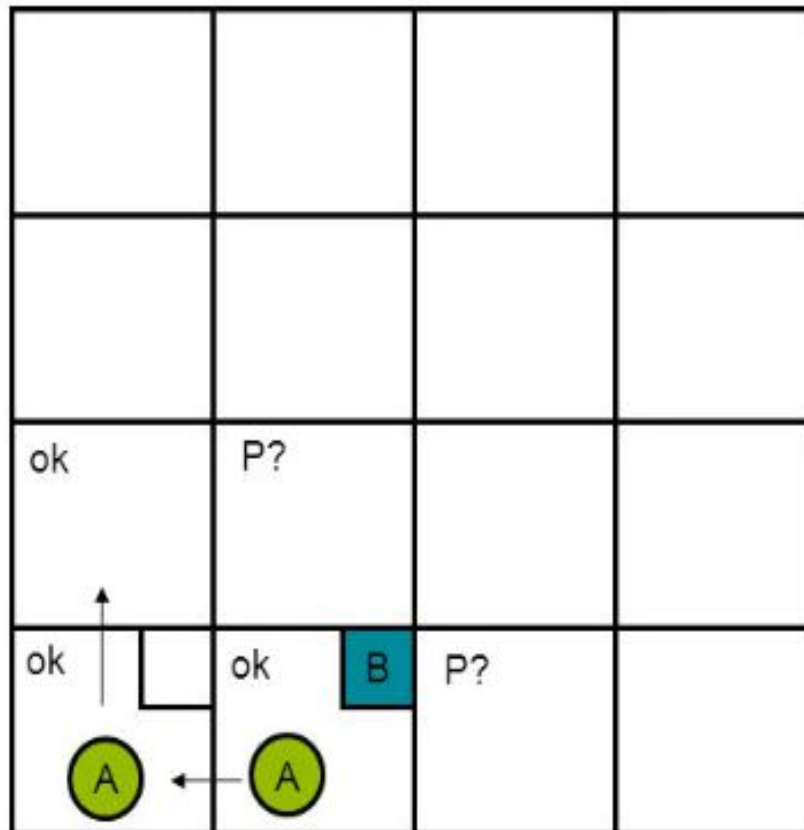
[stench, breeze, glitter, bump, scream]



Exploring the Wumpus world

$$\mathbf{x}_{1,2} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Agent feels a breeze



[2,1] = breeze

indicates that there is a
pit in [2,2] or [3,1],

return to [1,1] to try next
safe cell

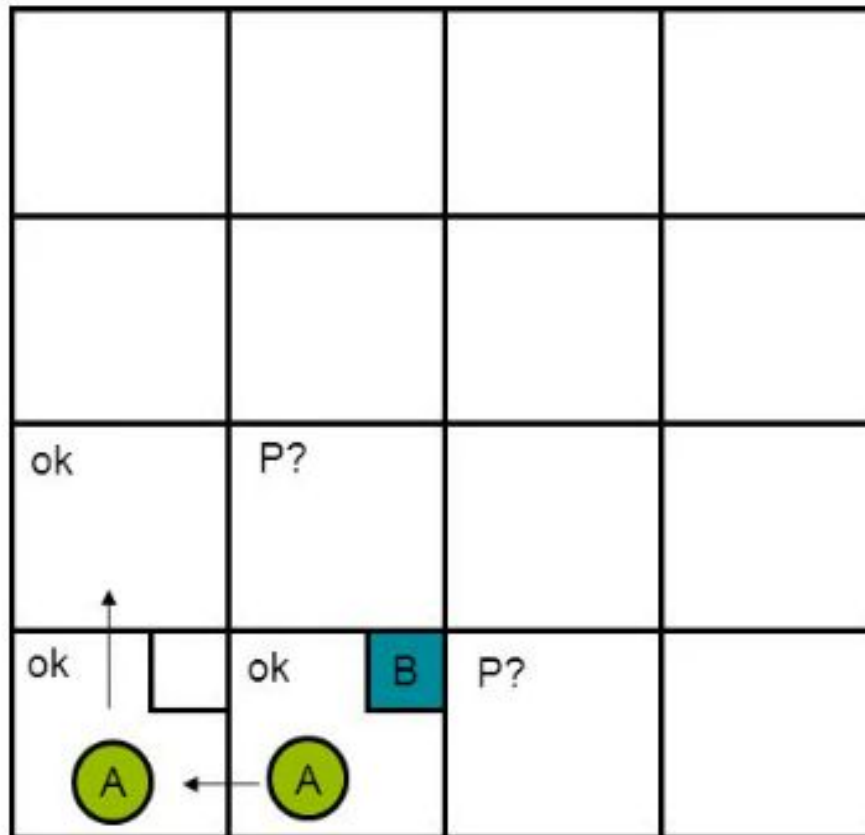
[stench, breeze, glitter, bump, scream]



Exploring the Wumpus world

$$\mathbf{x}_{1,2} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Agent feels a breeze



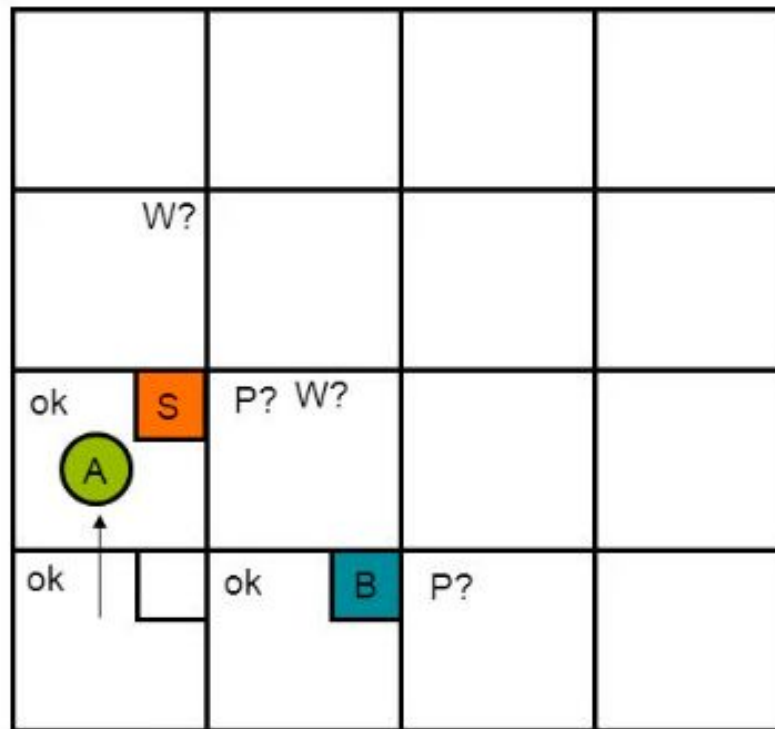
[stench, breeze, glitter, bump, scream]



Exploring the Wumpus world

Agent feels a foul smell

$$\mathbf{x}_{2,1} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

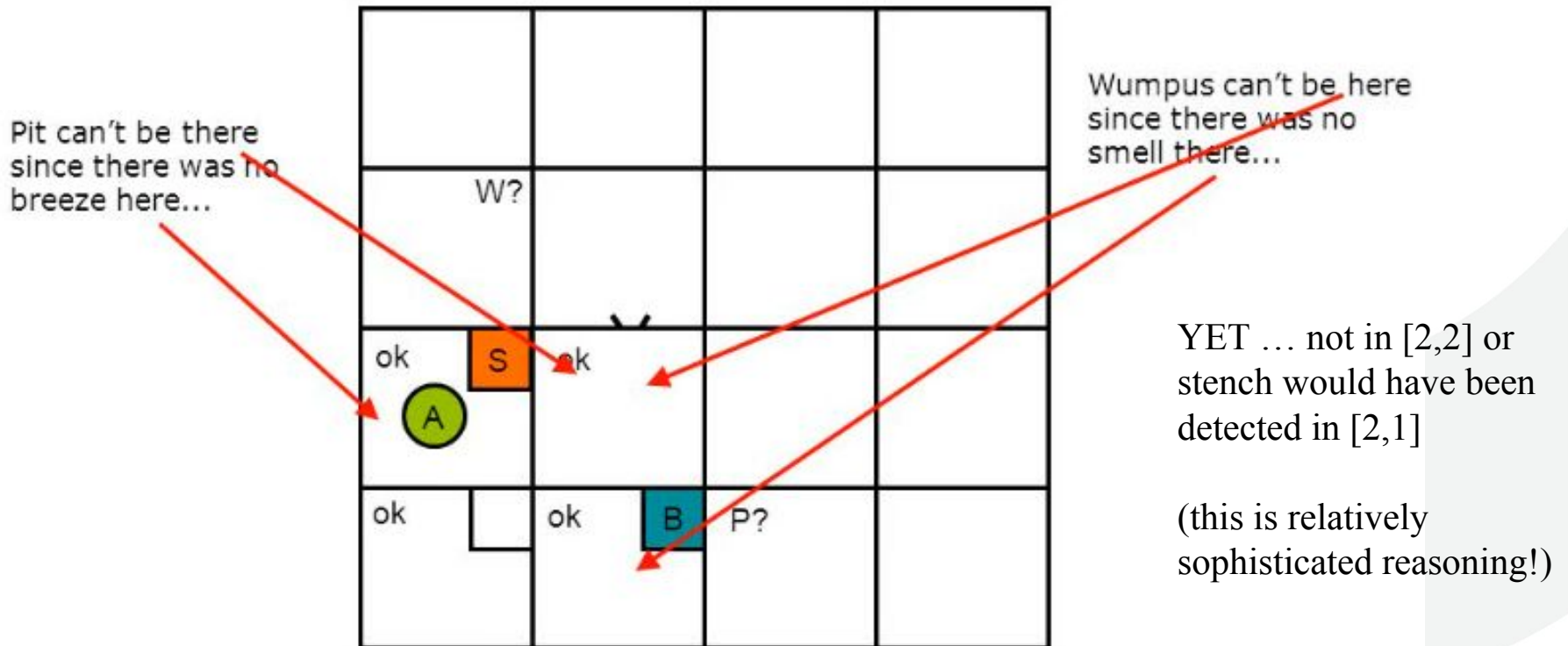


[1,2] Stench in cell which
means that wumpus is in
[1,3] or [2,2]

YET ... not in [1,1]

[stench, breeze, glitter, bump, scream]

Exploring the Wumpus world

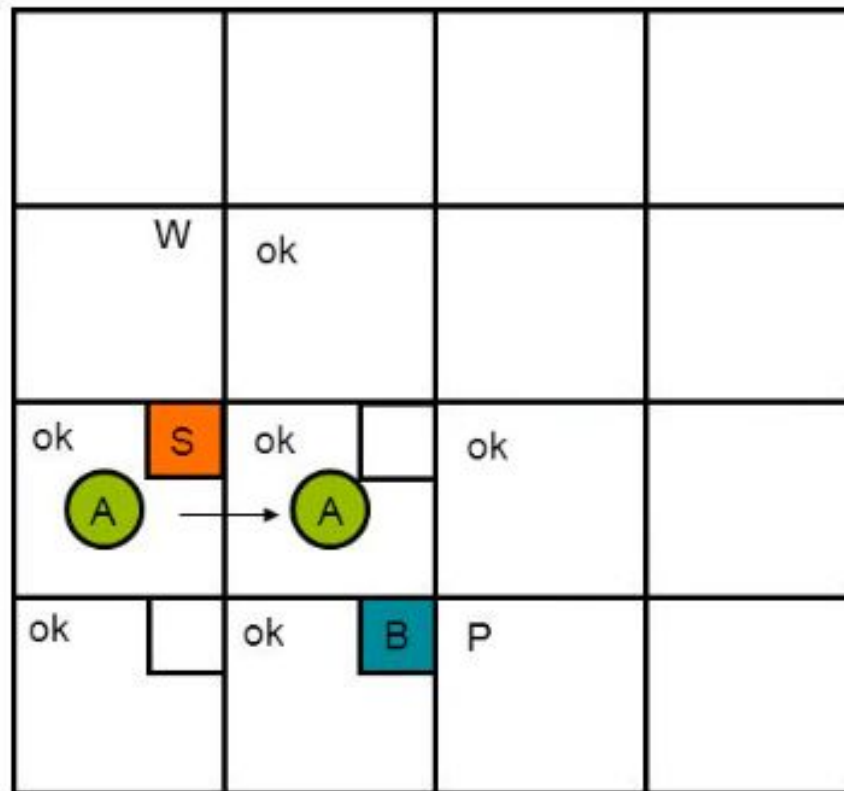




Exploring the Wumpus world

Agent senses nothing
(no breeze, no smell,..)

$$\mathbf{x}_{2,2} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

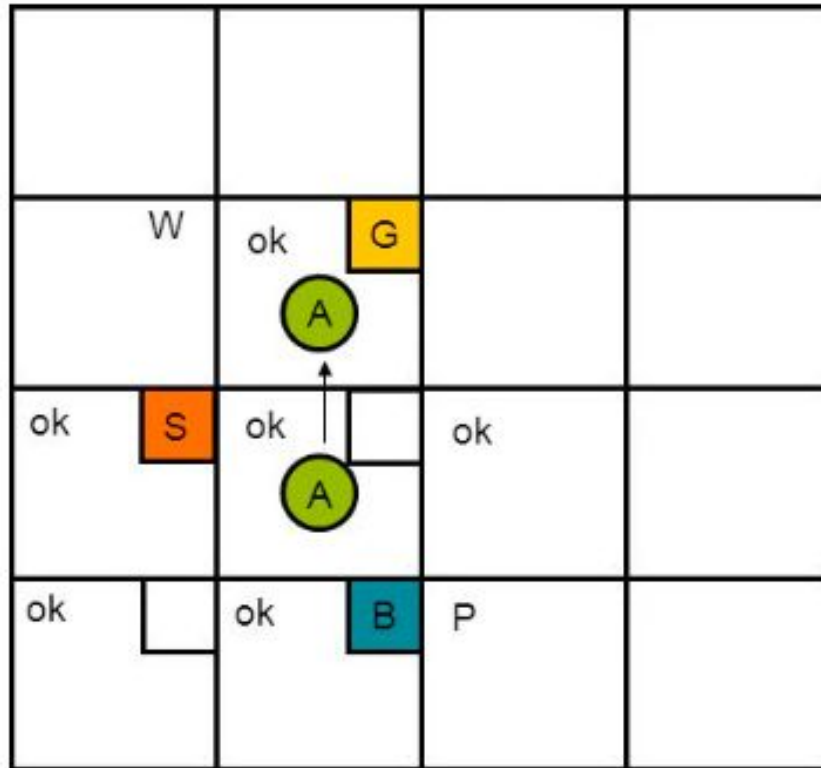




Exploring the Wumpus world

Agent senses breeze,
smell, and sees gold!

$$\mathbf{x} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$



THUS ... wumpus is in
[1,3]

THUS [2,2] is safe
because of lack of breeze
in [1,2]

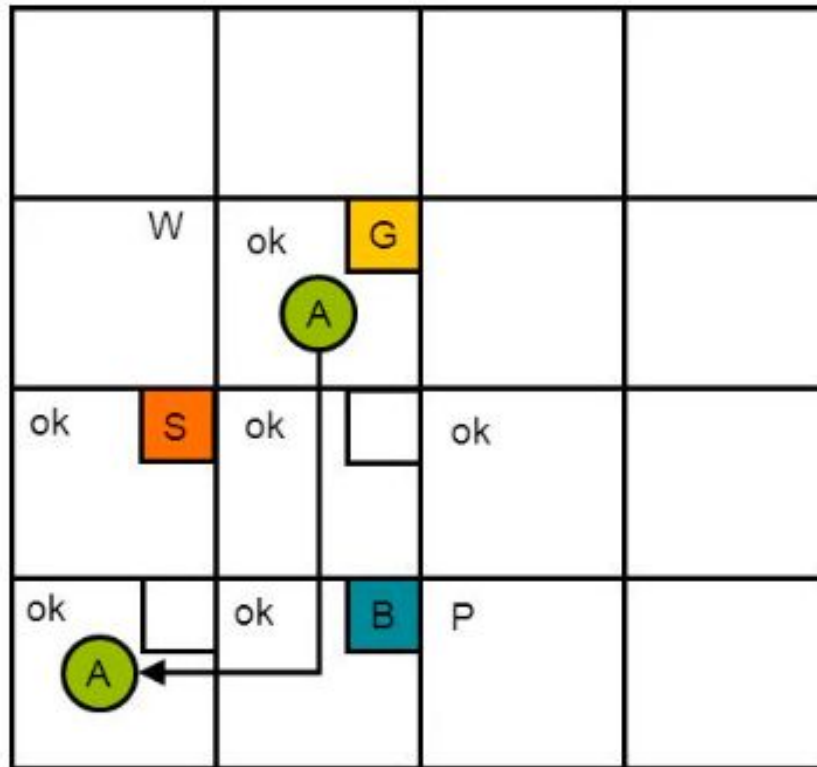
THUS pit in [1,3] (again
a clever inference)

move to next safe cell
[2,2]

[stench, breeze, glitter, bump, scream]



Exploring the Wumpus world



[2,2] move to [2,3]

[2,3] detect glitter ,
smell, breeze

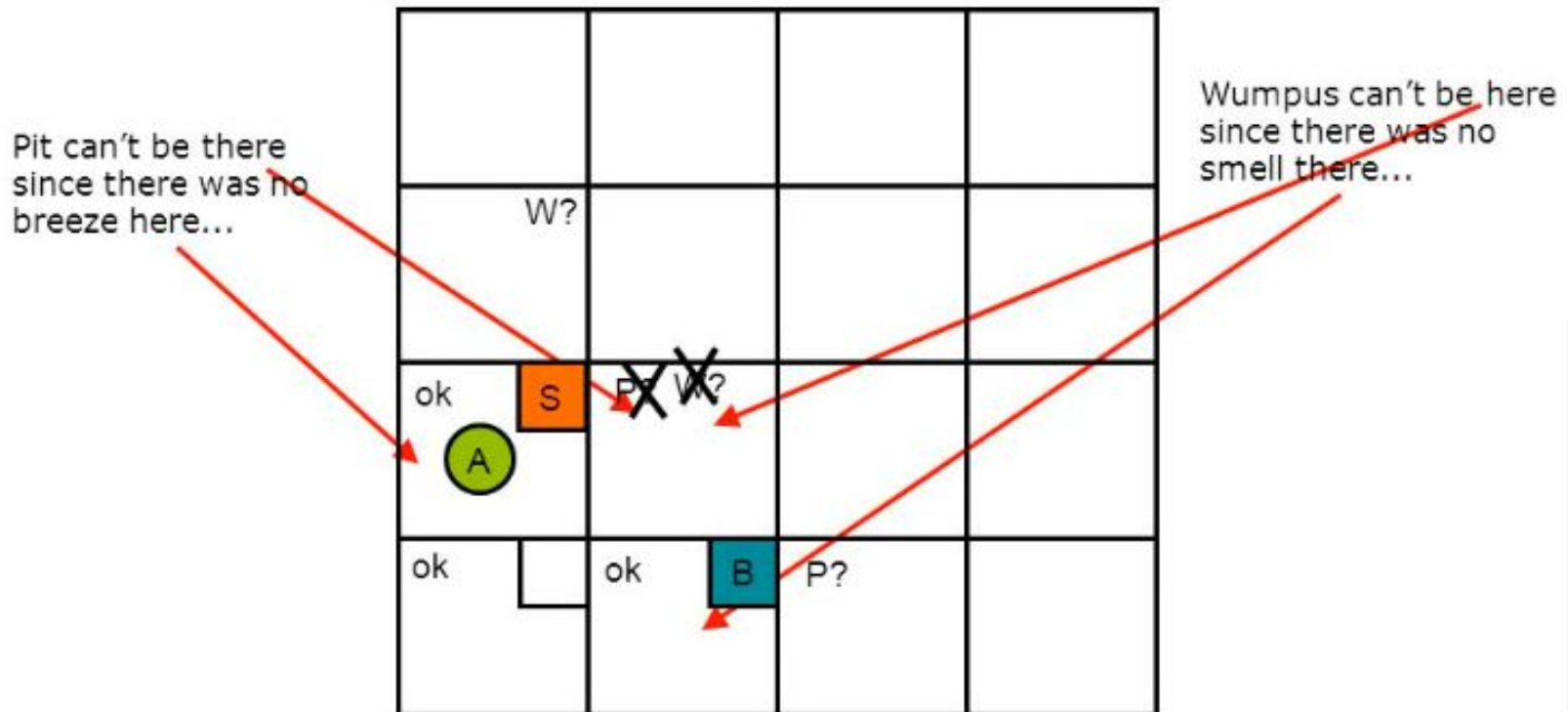
THUS pick up gold

THUS pit in [3,3] or
[2,4]

Grab the gold and
get out!

[stench, breeze, glitter, bump, scream]

Exploring the Wumpus world



How do we automate this kind of reasoning?
(How can we make these inferences automatically?)

[stench, breeze, glitter, bump, scream]



1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1	2,1	3,1	4,1
A			
OK	OK		

(a)

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK	P?		
1,1	2,1	3,1	4,1
V	A	P?	
OK	B		
	OK		

(b)

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].

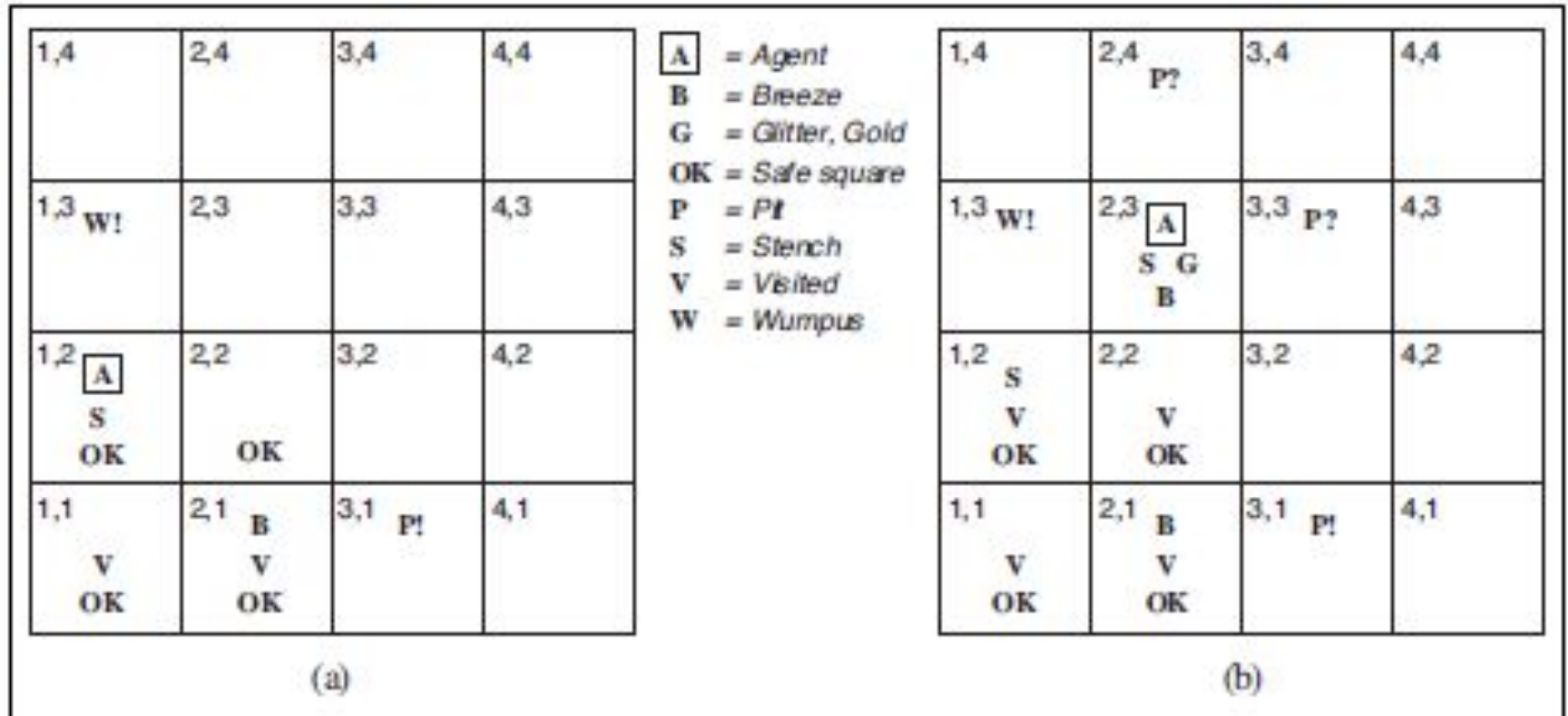


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [Stench, None, None, None, None]. (b) After the fifth move, with percept [Stench, Breeze, Glitter, None, None].





RV Institute of Technology
and Management®

What our example has shown us

Can represent general knowledge about an environment by a set of rules and facts

Can gather evidence and then infer new facts by combining evidence with the rules

The conclusions are guaranteed to be correct if

- The evidence is correct

- The rules are correct

- The inference procedure is correct

- > logical reasoning

The inference may be quite complex

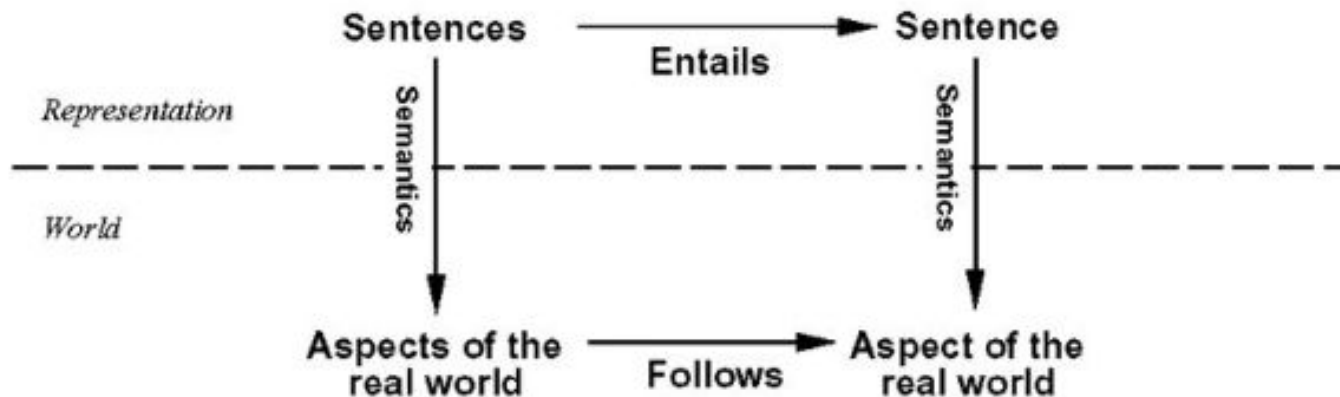
- E.g., evidence at different times, combined with different rules, etc

Logic

Logic is a formal language for representing information such that conclusions can be drawn

A logic has

- **Syntax** that specifies symbols in the language and how they can be combined to form *sentences*
- **Semantics** that specifies what facts in the world a semantics refers to. Assigns truth values to sentences based on their meaning in the world.
- **Inference procedure**, a mechanical method for computing (deriving) new (true) sentences from existing sentences





Logic

- Knowledge bases consist of sentences in a formal language
 - Syntax
 - defines how symbols can be put together to form the sentences in the language
 - Semantics
 - The “meaning” of the sentence
 - ***The truth of each sentence with respect to each possible world (model)***
- Example:
 - $x + 2 \geq y$ is a sentence
 - $x^2 + y > \{\}$ is not a sentence
 - $x + 2 \geq y$ is true iff $x + 2$ is no less than the number y
 - $x + 2 \geq y$ is true in a world where $x = 7, y = 1$
 - $x + 2 \geq y$ is false in world where $x = 0, y = 6$



Logic

- **Entailment** means that one thing **follows logically from** another
 $\alpha \models \beta$ or $KB \models \alpha$
- $\alpha \models \beta$ iff in every model in which α is true, β is also true
- if α is true, then β must be true
- the truth of β is “contained” in the truth of α



- Example:
 - A KB containing
 - “Cleveland won”
 - “Dallas won”
 - Entails...
 - “Either Cleveland won or Dallas won”

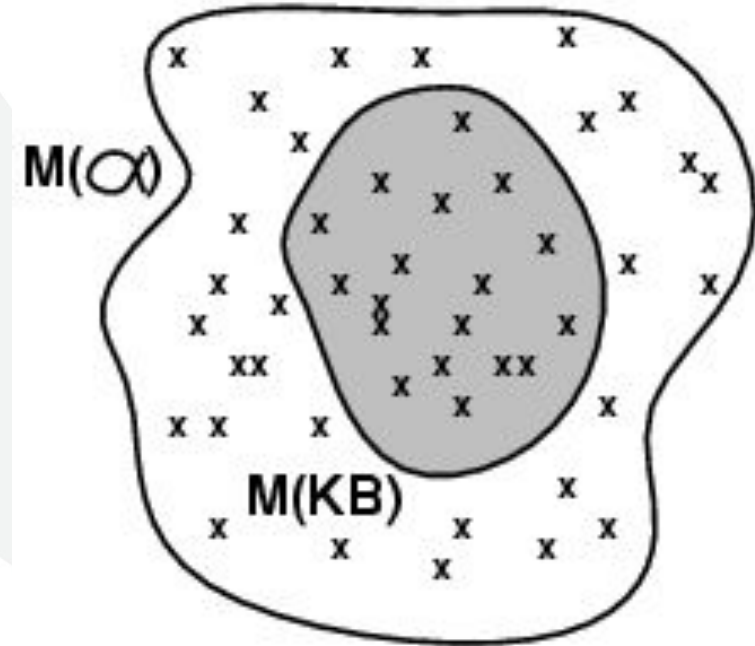
- Example:

$$x + y = 4 \text{ entails } 4 = x + y$$

Entailment is a relationship between sentences (i.e.,
syntax) that is based on **semantics**

Logic

- A model is a formally structured world with respect to which truth can be evaluated
 - We say, M is a model of sentence α if α is true in m
 - $M(\alpha)$ is the set of all models of α
 - Then $KB \models \alpha$ if $M(KB) \subseteq M(\alpha)$

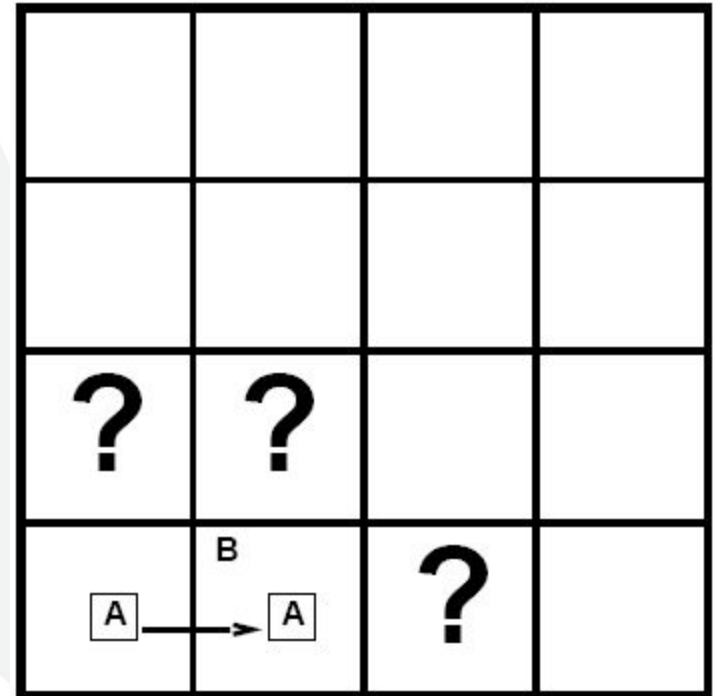


E.g. KB = Cleveland won and
Dallas won α = Cleveland won



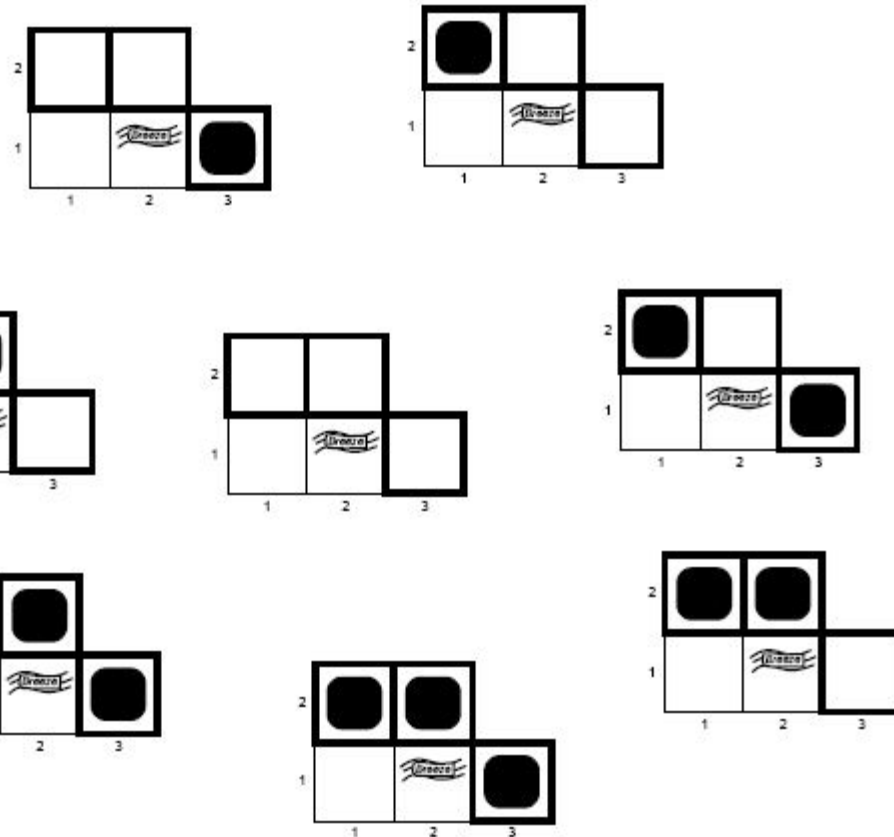
Logic

- Entailment in Wumpus World
- Situation after detecting nothing in [1,1], moving right, breeze in [2,1]
- Consider possible models for ? assuming only pits
- 3 Boolean choices \Rightarrow 8 possible models



The percepts and the agent's knowledge of the rules of the game (from the PEAS description) constitute the KB. The agent wants to know if the adjacent squares contain pits. Each may or may not contain a pit.

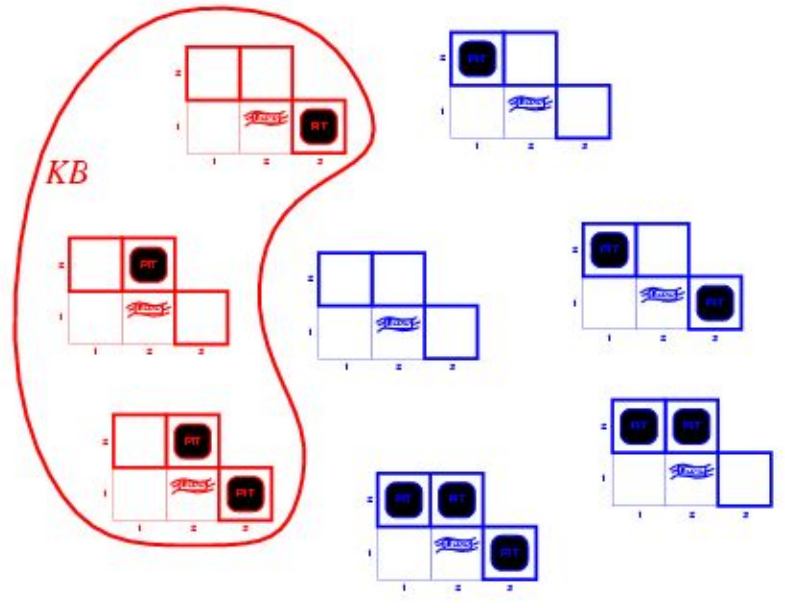
Wumpus possible models



Although the figure shows possible wumpus worlds, they are really nothing more than assignments of true and false to the sentences “there is a pit in ...” [1,2], [2,2], [3,1]



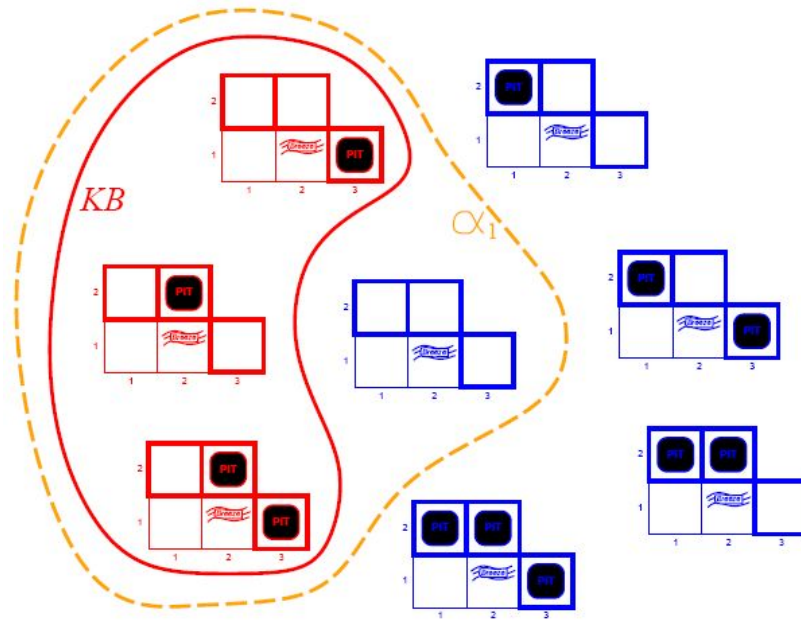
Wumpus models



$KB = \text{wumpus-world rules} + \text{observations}$

The KB is false in models that contradict what the agent knows – there are just 3 models in which the KB is true.

e.g., KB is false in any models in which [1,2] contains a pit, because there is no breeze in [1,1]

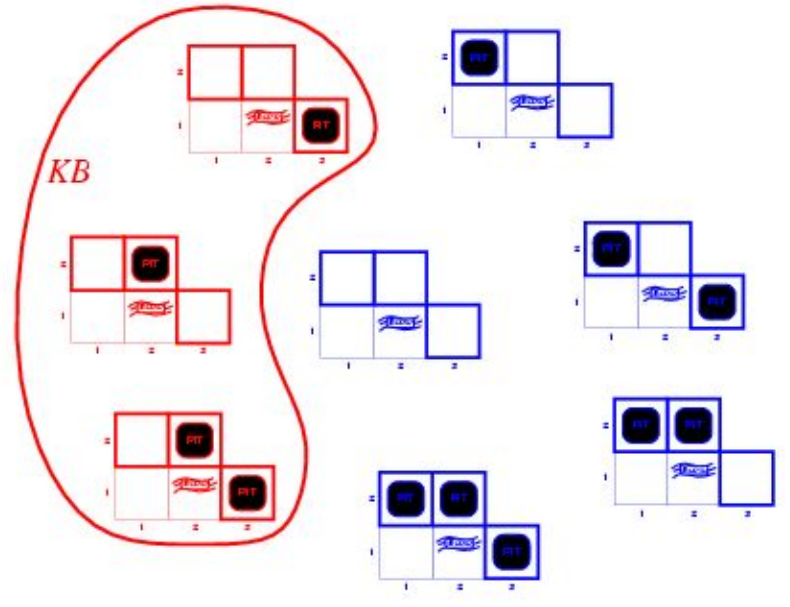


- $KB = \text{wumpus world rules} + \text{observations}$
- $\alpha_1 = \text{"there is no pit in } [1,2]\text{"}, KB \models \alpha_1$, proved by **model checking**

Model checking – enumerate all possible models, and check to see that alpha is true in all models in which KB is true (and then it must be the case that the KB entails alpha).

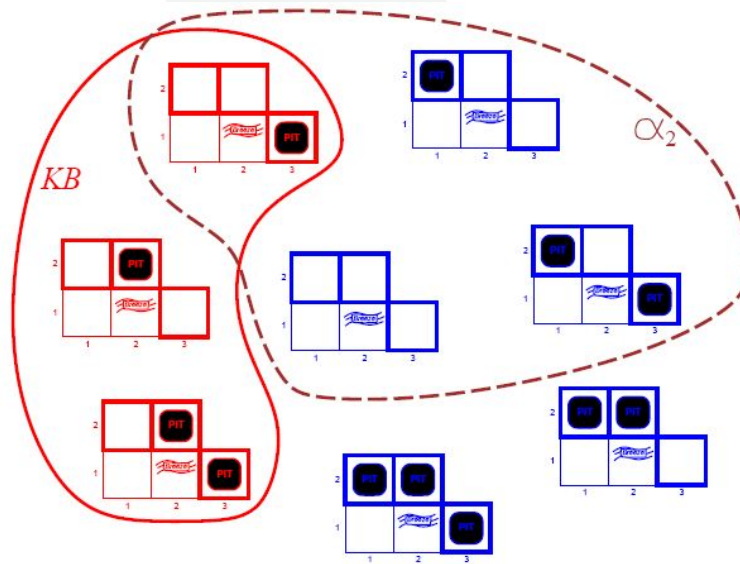


Wumpus models



- KB = wumpus-world rules + observations

Logic



- KB = wumpus world rules + observations
- α_2 = “[2,2] is safe”, $KB \not\models \alpha_2$ proved by model checking

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].)



Logic

- **Inference** is the process of deriving a specific (new) sentence from a KB (where the sentence must be entailed by the KB)
 - $KB \vdash_i \alpha$ = sentence α can be derived from KB by procedure i
- Understanding inference and entailment: think of
 - Set of all consequences of a KB as a haystack
 - α as the needle
- Entailment is like the needle being in the haystack
- Inference is like finding it
- “KB’s are a haystack”
 - Entailment = needle in haystack
 - Inference = finding it



Inference

- $KB \vdash_i \alpha$ = sentence α can be derived from KB by inference procedure I
- Soundness: i is sound if whenever $KB \vdash_i \alpha$, it is also true that $KB \models \alpha$
- Completeness: i is complete if whenever $KB \models \alpha$, it is also true that $KB \vdash_i \alpha$
- Preview: we will define a logic (first-order logic) which is expressive enough to say almost anything of interest, and for which there exists a sound and complete inference procedure.
- That is, the procedure will answer any question whose answer follows from what is known by the KB .



Propositional Logic : Syntax

- AKA Boolean Logic (False and True)
- Proposition symbols $P1, P2$, etc are (atomic) sentences
- NOT: If $S1$ is a sentence, then $\neg S1$ is a sentence (**negation**)
- AND: If $S1, S2$ are sentences, then $S1 \wedge S2$ is a sentence (**conjunction**)
- OR: If $S1, S2$ are sentences, then $S1 \vee S2$ is a sentence (**disjunction**)
- IMPLIES: If $S1, S2$ are sentences, then $S1 \Rightarrow S2$ is a sentence (**implication**)
- IFF: If $S1, S2$ are sentences, then $S1 \Leftrightarrow S2$ is a sentence (**biconditional**)



Propositional logic: Semantics

Each model specifies true/false for each proposition symbol

E.g. $P_{1,2}$ $P_{2,2}$ $P_{3,1}$
false true false

With these symbols, 8 possible models, can be enumerated automatically.

Rules for evaluating truth with respect to a model m :

$\neg S$	is true iff	S is false	
$S_1 \wedge S_2$	is true iff	S_1 is true and	S_2 is true
$S_1 \vee S_2$	is true iff	S_1 is true or	S_2 is true
$S_1 \Rightarrow S_2$	is true iff	S_1 is false or	S_2 is true
i.e.,	is false iff	S_1 is true and	S_2 is false
$S_1 \Leftrightarrow S_2$	is true iff	$S_1 \Rightarrow S_2$ is true and	$S_2 \Rightarrow S_1$ is true

Simple recursive process evaluates an arbitrary sentence, e.g.,

$$\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1}) = \text{true} \wedge (\text{true} \vee \text{false}) = \text{true} \wedge \text{true} = \text{true}$$



Truth tables for connectives

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>

John likes football and John likes baseball.

John likes football or John likes baseball.

(English or is a bit different...)



Truth tables for connectives

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

John likes football and John likes baseball.

John likes football or John likes baseball.

If John likes football then John likes baseball.

(Note different from English – if John likes football maps to false, then the sentence is true.)

(Implication seems to be if antecedent is true then I claim the consequence is, otherwise I make no claim.)



Wumpus World Sentences

Let $P_{i,j}$ be true if there is a pit in $[i, j]$.
Let $B_{i,j}$ be true if there is a breeze in $[i, j]$.

$$\neg P_{1,1}$$

$$\neg B_{1,1}$$

$$B_{2,1}$$

"Pits cause breezes in adjacent squares"

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

A square is breezy if and only if there is an adjacent pit



Simple Inference Procedure

- $KB \models \alpha?$
- Model checking – enumerate the models, and check if α is true in every model in which KB is true. Size of truth table depends on # of atomic symbols.
- Remember – a model is a mapping of all atomic symbols to true or false – use semantics of connectives to come to an interpretation for them.

Truth tables for inference

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	KB	α_1
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<u><i>true</i></u>	<u><i>true</i></u>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<u><i>true</i></u>	<u><i>true</i></u>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>	<u><i>true</i></u>	<u><i>true</i></u>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>

This is taken from the text – meant to show how you need to enumerate all possibilities – then look in the lines of the table where the KB is true.

A Simple Knowledge Base

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	KB	α_1
false	false	false	false	false	false	false	false	true
false	false	false	false	false	false	true	false	true
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
false	true	false	false	false	false	false	false	true
false	true	false	false	false	false	true	<u>true</u>	<u>true</u>
false	true	false	false	false	true	false	<u>true</u>	<u>true</u>
false	true	false	false	false	true	true	<u>true</u>	<u>true</u>
false	true	false	false	true	false	false	false	true
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
true	true	true	true	true	true	true	false	false

- $R1: \neg P_{1,1}$
- $R2: B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
- $R3: B_{2,1} (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
- $R4: \neg B_{1,1}$
- $R5: B_{2,1}$

- KB consists of sentences R_1 thru R_5
- $R1 \wedge R2 \wedge R3 \wedge R4 \wedge R5$



A Simple Knowledge Base

```
function TT-ENTAILS?( $KB, \alpha$ ) returns true or false  
     $symbols \leftarrow$  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  
    else do  
         $P \leftarrow$  FIRST( $symbols$ );  $rest \leftarrow$  REST( $symbols$ )  
        return TT-CHECK-ALL( $KB, \alpha, rest, EXTEND(P, true, model)$ ) and  
            TT-CHECK-ALL( $KB, \alpha, rest, EXTEND(P, false, model)$ )
```

- Every known inference algorithm for propositional logic has a worst-case complexity that is exponential in the size of the input. (co-NP complete)



Logical equivalence

- Two sentences are logically equivalent iff true in same models: $\alpha \equiv \beta$ iff $\alpha \models \beta$ and $\beta \models \alpha$

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$

$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$

$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$

$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$

$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$

$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition}$$

$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$

$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}$$

$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{de Morgan}$$

$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{de Morgan}$$

$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$

$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$



RV Institute of Technology
and Management®

End of Mod 3