Unit-02: Advanced Search and Basic Knowledge Representation & Reasoning

Artificial Intelligence

Dr. Mohammad Fayazur Rahaman
Associate Professor, mfrahaman_ece@mgit.ac.in

Dept. of Electronics and Communications Engineering,
Mahatma Gandhi Institute of Technology, Gandipet, Hyderabad-75

Mar-Jun 2022

Unit-02: Advanced Search and Basic Knowledge Representation & Reasoning [1]

Adversarial Search and Games

I. In this chapter we cover **competitive environments**, in which **two or more** agents have **conflicting goals**, giving rise to **adversarial search** problems
II. We begin with a restricted **class of games**, and define the **optimal move** and **an algorithm** for finding it: **minimax search**
III. We show that **pruning** makes the search **more efficient** by **ignoring portions** of the search tree that makes no difference to the optimal move

Where are we ?

Example: Game Tree

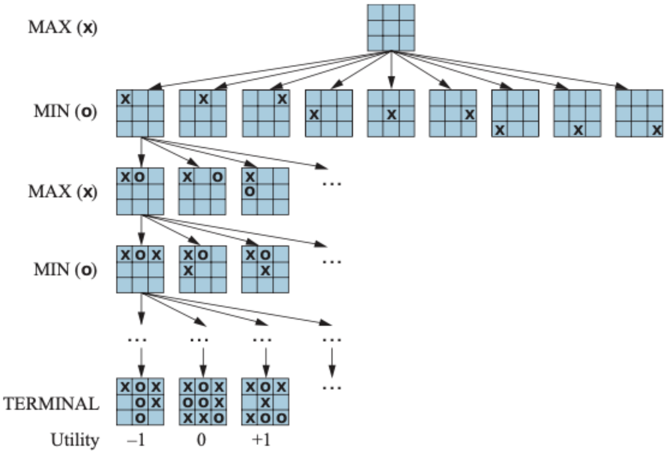


Figure 6.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Constructing Search Trees

Two-player zero-sum games

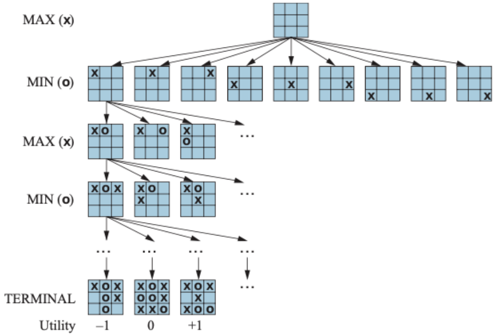


Figure 6.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

i. Let us consider a game that is **deterministic**, **two-player**, **turn-taking**, **perfect information** and **zero-sum**
   - Note that **perfect information** is a synonym for **fully observable**
   - **Zero-sum** means that what is **good** for one player is just as **bad** for the other

ii. For games, we use the term **move** as a synonym for **action** and **position** as a synonym for **state**

iii. We will call our two players `MAX` and `MIN`
   - `MAX` moves first, and then the players take turns moving until the game is over
   - At the end of the game, **points** are awarded to the winning player and **penalties** are given to the loser

iv. A game can be formally defined with the following elements
- S_0: The **initial state**, which specifies how the game is set up at the start
- `TO-MOVE`(s): The player **whose trun** it is to move in state s
- `ACTIONS`(s): The set of **legal** moves in state s
- `RESULT`(s, a): The **transition model**, which defines the state resulting from taking **action** a in state s
- `IS-TERMINAL`(s): A **terminal test**, which is true when the game is over and false otherwise. State where the game has ended are called **terminal states**
- `UTILITY`(s, p): A **utility function** which defines the final numeric value to player p when the game ends in terminal state s
  - In **chess**, the outcome is a win, loss, or draw, with values $+1$, -1, or 0

v. As before, the **initial state**, `ACTIONS` function, and `RESULT` function define the **state space graph**
- A **graph** is where the vertices are **states**, the **edges** are moves and a **state** might be reached by **multiple paths**

vi. Further, we can superimpose a **search tree** over part of that graph to **determine what move** to make
- We define the complete **game tree** as a **search tree** that follows every **sequence of moves** all the way to the **terminal state**

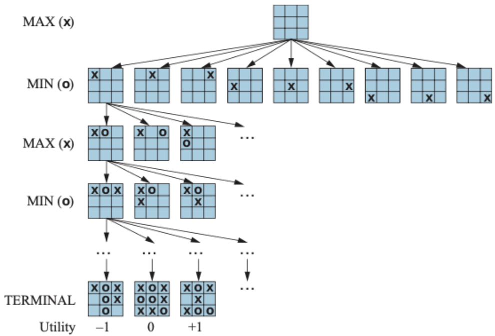


Figure 6.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

vii. The figure shows part of the game three for tic-tac-toe (noughts and crosses)
- From the initial state, `MAX` has 9 possible moves
- Play alternate between `MAX` placing an X and `MIN` placing an O until we reach leaf nodes corresponding to the terminal states, such that one player has three squares in a row or all the squares are filled
- The number on each leaf node indicates the **utility value** of the terminal state from the point of view of `MAX`

viii. For **Tic-Tac-Toe** the **game tree** is relatively small, i.e., fewer than $9! = 362,880$ terminal nodes
- But for **chess** there are over 10^{40} nodes, so the **game tree cannot be realised** in the physical world

Where are we ?

Example: Game Tree

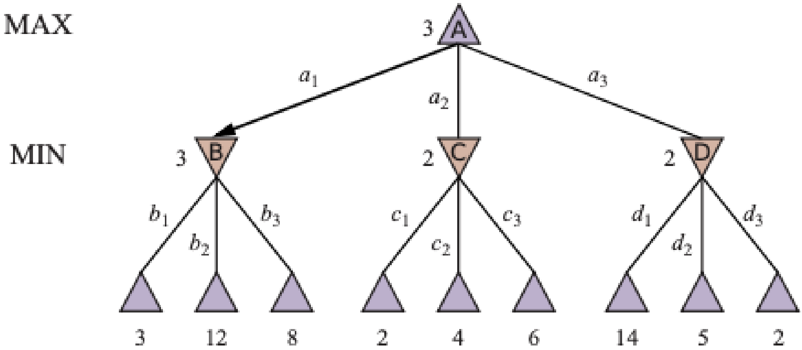


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Minimax Search

Optimal Decisions in Games

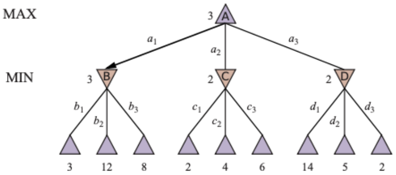


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

i. `MAX` wants to find a sequence of actions leading to a win, but `MIN` has something to say about it
   - This means that `MAX` strategy must be a **conditional plan** - contingent strategy specifying a response to each of `MIN`'s possible moves

ii. For games with **multiple outcome** scores, we need a slightly more general algorithm called **minimax search**

iii. Consider the trivial game in the adjacent figure;
   - The possible moves for `MAX` at the root node are labelled a_1, a_2, and a_3
   - The possible replies to a_1 for `MIN` are b_1, b_2, b_3, and so on
   - This particular game ends after one move each by `MAX` and `MIN`. (Note that word '**Ply**' is used to mean '**move**' sometimes)
   - The **utilities** of the terminal states in this game range from 2 to 14

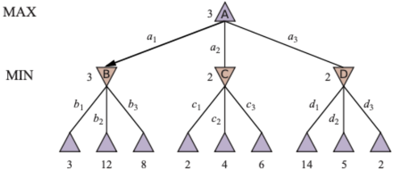


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

iv. Given a **game tree**, the **optimal strategy** can be determined by working out the **minimax value** of each state in the tree, which are written as `MINIMAX`(s)
- The **minimax value** is the **utility** of `MAX` for being in that state, assuming both players play **optimally** from there to the end of the game
- The minimax value of a **terminal state** is just its utility
- In a non-terminal stage,
  - `MAX` prefers to move to a state of **maximum value** where it is `MAX` turn to move, and
  - `MIN` prefers the state of **minimum value** for `MAX`
- So we have

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

v. In the fig, the **terminal nodes** on the bottom level get their utility values from the games `UTILITY` function
- The first `MIN` node, labelled B, has three successor states with values 3, 12, and 8, so it's **minimax value** is 3
  - Similarly, the other two `MIN` nodes have minimax value of 2
- The root node is a `MAX` node, its successor states have **minimax values** of 3,2, and 2; so it has a **minimax value** of 3.
  - We can also identify the minimax decision at the root; action a_1 is the optimal choice for `MAX` because it leads to the state with the **highest minimax value**

The minimax search algorithm

vii. Now that we can compute $\texttt{MINIMAX}(s)$, we can turn that into a **search algorithm** that finds the best move for $\texttt{MAX}$ by trying all actions and choosing the one whose resulting state has the **highest minimax value**
   - It is a **recursive algorithm** that proceeds all the way down to the **leaves** of the tree and then **backs up** the **minimax values** through the tree as the recursion unwinds

viii. The **minimax algorithm** performs a complete **depth first exploration** of the game tree
   - If the **maximum depth** of the tree is m and there are b **legal move** at each point, then the **time complexity** of the minimax algorithm is $\mathcal{O}(b^m)$ and the
   - Space complexity is $\mathcal{O}(b\,m)$

ix. The **exponential complexity** makes MINIMAX **impractical** for complex games;
   - For example in chess, this has a branching factor of about 35 and the average game has depth of about 80 ply; and it is not feasible to search $35^{80} \approx 10^{123}$ states

x. By **approximating** the minimax analysis in various ways, we can derive **more practical** algorithms

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 6.3 An algorithm for calculating the optimal move using minimax—the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

Example: Multiplayer Game Tree

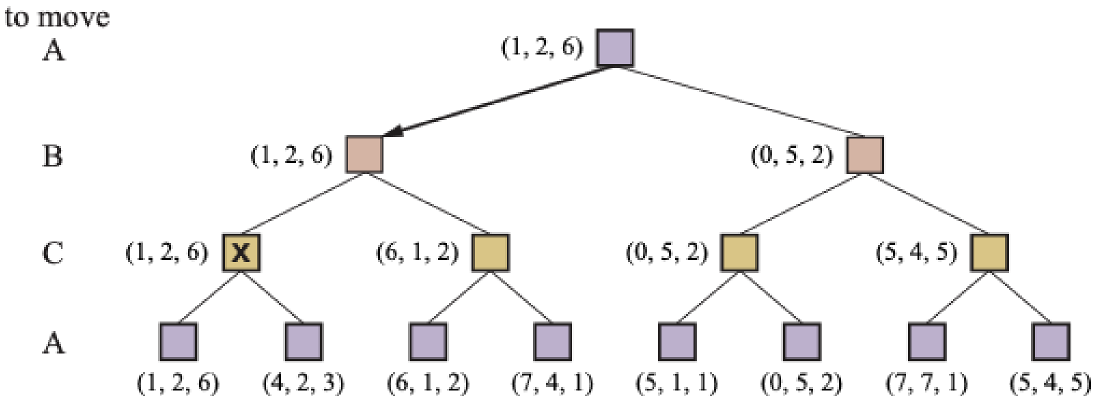


Figure 6.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Where are we ?

Example: Alpha-Beta Pruning

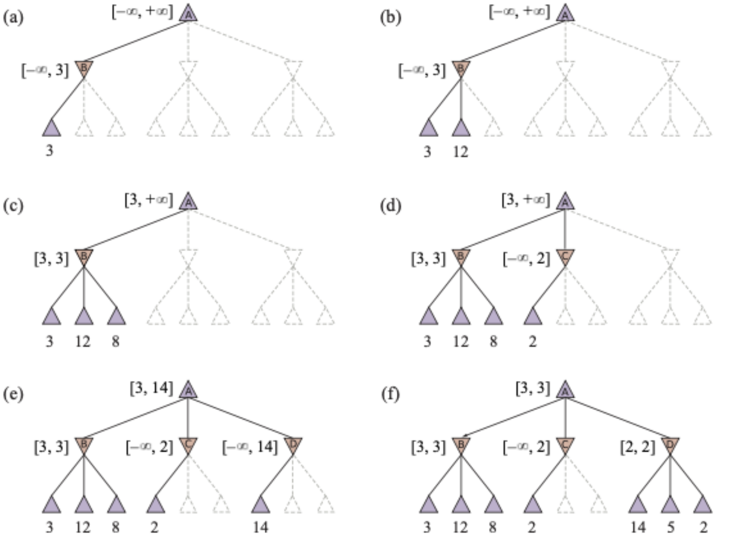


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2.

Alpha-Beta Pruning

i. We saw that the **number** of game states is **exponential** in the **depth** of the tree

ii. No algorithm can **completely eliminate** the **exponent**, but we can sometimes **cut in half**,
   - by computing the correct minimax decision **without examining** every state by **pruning**
   - i.e., by **eliminating large parts** of the tree that makes no difference to the outcome.
   - The particular technique we examine is called **alpha-beta pruning**

iii. On paying careful attention at each point in the figure, we can identify the minimax decision **without ever evaluating** two of the leaf nodes
   - Let the two **unevaluated** successors of Node C in the figure have values x and y. Then the **value** of the root node is given by

$$
\begin{aligned}
\text{MINIMAX}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}
$$

- In other words, the value of the **root** and hence the **minimax decision** are **independent** of the values of the leaves x and y, and therefore can be **pruned**

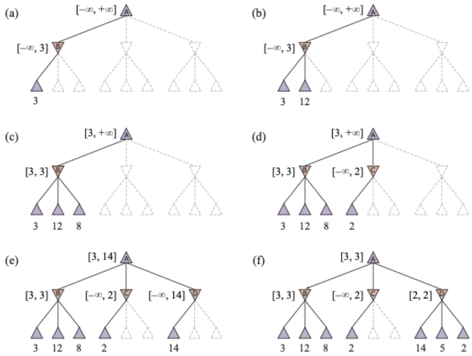


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2.

v. **Alpha-beta pruning** can be applied to trees of any depth, and it is often possible to **prune entire subtrees** rather than just leaves

vi. General principle is that:
- Consider a node n (in fig), such that player has a choice of moving to n,
- If player has a **better choice** either at the same level (m') or at any point higher up in the tree (m), then player **will never move** to n.
- So once we have **found out enough** about n to reach this conclusion, we can **prune** it

vii. The **Alpha-beta pruning** gets its name from the **two extra parameters** in `MAX-VALUE`(s, α, β) that describe bounds on the backed-up values that appear anywhere along the path

$\alpha =$ the value of the best (i.e., **highest-value**) choice we have found along the path for `MAX`. Think: $\alpha =$ **"at least"**

$\beta =$ the value of the best (i.e., **lowest-value**) choice we have found along the path for `MIN`. Think: $\beta =$ **"at most"**

viii. Alpha-beta search **updates** the values of α and β as it goes along and **prunes** the remaining branches at a node, as soon as the value of the current node is known to be **worse than** the current α or β value `MAX` and `MIN` respectively

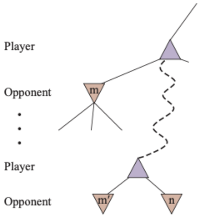


Figure 6.6 The general case for alpha–beta pruning. If *m* or *m′* is better than *n* for Player, we will never get to *n* in play.

ix. The **effectiveness** of alpha-beta pruning is highly **dependent** on the **order** in which the states are examined
- If done perfectly, **alpha-beta** would need to examine only $\mathcal{O}(b^{m/2})$ nodes to pick up the best move, instead of $\mathcal{O}(b^m)$ for **minimax**
- This means that the **effective branching factor** becomes $\sqrt{b}$ instead of b : for chess, about 6 instead of 35

```
function ALPHA-BETA-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state, −∞, +∞)
   return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
      if v2 > v then
         v, move ← v2, a
         α ← MAX(α, v)
      if v ≥ β then return v, move
   return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
      if v2 < v then
         v, move ← v2, a
         β ← MIN(β, v)
      if v ≤ α then return v, move
   return v, move
```

Figure 6.7 The alpha–beta search algorithm. Notice that these functions are the same

Heuristic Alpha-Beta Tree Search

i. To make use of our **limited computation time**, we can **cut-off** the search early and apply a **heuristic evaluation function** to states, effectively treating **non-terminal nodes** as if they were **terminal**
   - In other words, we replace the `UTILITY` function with `EVAL`, which estimates a state's utility
   - We also replace the terminal test by a `CUTOFF` test, which must **return true** for terminal states, but is otherwise **free** to decide when to **cutoff** the search, based on the **search depth** and **any property** of the state that chooses to consider

ii. That gives us the formula $\texttt{H-MINIMAX}(s, d)$ for the **heuristic minimax value** of state s at search depth d:

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s, \text{MAX}) & \text{if IS-CUTOFF}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MIN.} \end{cases}$$

iii. A **heuristic evaluation** function $\texttt{EVAL}(s, p)$ returns an estimate of the **expected utility** of state s to player p
   - For terminal states, it must be that

$$\texttt{EVAL}(s, p) = \texttt{UTILITY}(s, p)$$

   - For non-terminal states, the evaluation must be somewhere between a loss and a win

$$\texttt{UTILITY}(loss, p) \leq \texttt{EVAL}(s, p) \leq \texttt{UTILITY}(win, p)$$

   - Further, the evaluation function should be **strongly correlated** with the actual chances of winning

iv. Most **evaluation functions** work by calculating various **features** of the state
   - For example, in chess, we would have **features** for the number of **white pawns**, **black pawns**, **white Queens**, **black Queens**, and so on

v. Suppose our experience suggests that 82% of the states encountered in a particular category lead to a win (utility $+1$), 2% to a loss (0), and 16% to a draw ($1/2$)
   - Then a **reasonable evaluation** for these states is the **expected value**

$$(0.82 \times +1) + (0.02 \times 0) + (0.16 \times 0.5) = 0.90$$

vi. This kind of analysis requires **too much experience** to estimate all the **probabilities**
   - Instead, most **evaluation functions** compute separate numerical contribution from **each feature** and then combine them to find the **total value**
   - For example, each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the Queen 9
   - This **feature values** are then simply added up to obtain the **evaluation** of the position

vii. Mathematically, this kind of **evaluation function** is called a **weighted linear function** because it can be expressed as

$$\begin{aligned}\texttt{Eval}(s) &= w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)\\ &= \sum_{i=1}^{n} w_i f_i(s)\end{aligned}$$

   - where each f_i is a **feature** of the position and
   - each w_i is a **weight**

Where are we ?

Example: Backgammo

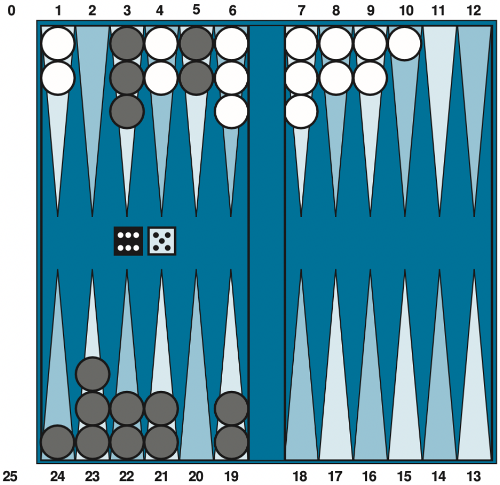


Figure 6.12 A typical backgammon position. The goal of the game is to move all one's pieces off the board. Black moves clockwise toward 25, and White moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, Black has rolled 6–5 and must choose among four legal moves: (5–11,5–10), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

Stochastic Games

i. Backgammon is **stochastic game** that combines luck and skill.
   - It is a little closer to the **unpredictability** of real life by including a random element, such as **throwing of dice**

ii. In the figure shown, **black** has rolled a 6-5 and has **four possible** moves
   - Each of which moves one piece forward clockwise 5 positions, and one piece forward 6 positions

iii. At this point **black** knows what moves can be made, but **does not know** what **White** is going to roll and thus does not know what **White's** legal moves will be
   - As a result a **game tree** in **backgammon** must include **chance nodes** in addition to `MAX` and `MIN` nodes

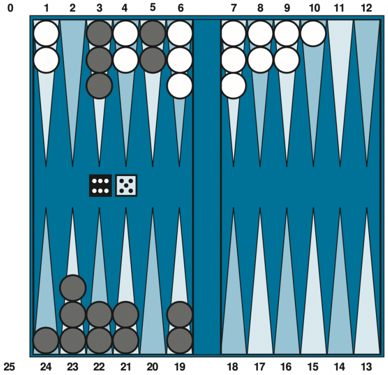


Figure 6.12 A typical backgammon position. The goal of the game is to move all one's pieces off the board. Black moves clockwise toward 25, and White moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, Black has rolled 6–5 and must choose among four legal moves: (5–11,5–10), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

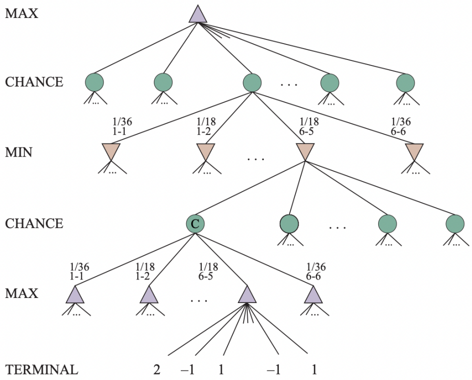


Figure 6.13 Schematic game tree for a backgammon position.

iv. In the figure, the **branches** leading from each **chance node** denote the possible **dice rolls**, each branch is labelled with the **roll** and its **probability**

- The six doubles (1-1 through 6-6) each have a **probability** of 1/36, i.e., $P(1-1) = 1/36$, and
- the other 15 distinct rolls each have a 1/18 **probability**

v. As a result, positions do not have **definite minimax** values, instead we can only calculate the **expected value** of a position

- i.e., the **average** overall possible outcomes of the chance to nodes

vi. This leads us to the **expectiminimax value** for games with chance nodes, i.e., a generalization of the minimax value

- For **chance nodes** we compute the expected value, which is the sum of the value over all outcomes, **weighted** by the **probability** of it chance action

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{min} \\ \sum_r P(r) \textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{To-Move}(s) = \textsc{chance} \end{cases}$$

- where r represents a possible dice roll

vii. Considering all the possible **dice roll sequences** the program take $\mathcal{O}(b^m n^m)$ time to solve the problem where n is the number of distinct rolls
   - Even if the search is limited to some small depth d, the **extra cost** compared with that of **minimax** makes it **unrealistic** to consider looking ahead very far in most **games of chance**
   - In backgammon n is 21 and b is usually around 20, with this we could probably only manage **three ply** of search

Summary

I. A game can be defined by the **initial state**, the **legal actions** in each state, the **result** of each action, it **terminal test**, and an **utility function** that applies to terminal stage to say who won and what the **final score** is
II. In two player, discrete, deterministic, turn-taking zero-sum games with **perfect information**, the **minimax algorithm** can select **optimal** move by a depth first enumeration of the game tree
III. The **alpha-beta search** algorithm computes the **same optimal** move as **minimax**, but achieves much **greater efficiency** by **eliminating** sub-trees that are **irrelevant**
IV. Usually, it is **not feasible** to consider the whole game tree even with **alpha-beta**, so we need to **cut the search** of at some point and apply a heuristic **evaluation function** that **estimates** the utility of a state
V. **Games of chance** can be handled by **expectiminimax**, an **extension** to the minimax algorithm that **evaluates the chance** node by taking the **average** utility of all its children, **weighted** by the **probability** of each child

Where are we ?

Introduction

I. In **AI**, **knowledge based agents** use the process of **reasoning** over an internal **representation of knowledge** to decide what **actions** to take
   - We could give a **human** the goal of driving to US town with population less than 10000, however
   - to say that a **problem-solving agent**, we need to formally describe the goal as an **explicit** set of 16,000 towns that **satisfy** the description

II. In this section, we develop **logic** as a general class of **representations** to support **knowledged-based agents**

III. These agents can combine and re-combine **information** to suit myriad purposes
   - They can accept new task in the form of **explicitly** described goals,
   - they can **achieve competence** quickly by being **told** or **learning** new knowledge about the environment, and
   - they can **adapt** to changes in the environment by **updating** the relevant knowledge

IV. In this section we explain the general **principles of logic**, and the specifics of **proportional logic**
   - **Proportional logic** is a **factored representation** and is less expressive then **first-order logic**, whereas
   - The **first-order logic** is the canonical structured representation

Knowledge-Based Agents

i. A **central component** of a knowledge-based agent is its **knowledge base**, or **KB**
ii. A knowledge base is a set of **sentences**
   - Each **sentence** is expressed in a language called in **knowledge representation language** and **represents** some **assertion** about the **world**
   - When the sentence is taken as being given **without being derived** from other sentences, we call it an **axiom**
iii. The operation to **add** new sentences to the knowledge base and a way to **query** are `TELL` and `ASK` respectively
   - The operation of deriving **new** sentences from **old** may involve **inference**
iv. The figure shows the **outline** of a **knowledge-based agent** and program
   - It takes a **percept** as input and returns and an **action**
   - The agent maintains a knowledge base, **KB**, which me initially contain some background knowledge
v. Each time the agent program is called, it does the following three tasks,
   I. It `TELLS` the knowledge base what it perceives
   II. It `ASKS` the knowledge base what action it should perform
   III. It `TELLS` the knowledge base which **action was chosen**
vi. The details of the **representation language** are hidden inside three functions
   I. `MAKE-PERCEPT-SENTENCE`: Construct a sentence asserting that the agent perceived the given percept at the given time
   II. `MAKE-ACTION-QUERY`: Construct a sentence that asks what action should be done at the current time
   III. `MAKE-ACTION-SENTENCE`: Construct a sentence asserting that the chosen action was executed

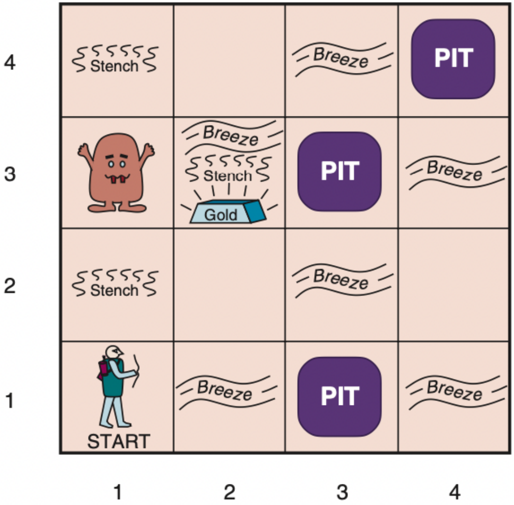
function KB-AGENT(*percept*) **returns** an *action*
  persistent: *KB*, a knowledge base
      t, a counter, initially 0, indicating time

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  action ← ASK(*KB*, MAKE-ACTION-QUERY(*t*))
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  $t \leftarrow t + 1$
  return *action*

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

The Wumpus World

i. The **Wumpus world** is a cave consisting of rooms connected by passageways
ii. Somewhere in the cave is the terrible Wumpus, a **beast** that eats anyone who enters its room
iii. The Wumpus can be **shot by an agent**, but the agent has only one arrow
iv. Some rooms contain **bottomless pits** that will trap anyone who wanders into this rooms
v. The only redeeming feature of this bleak environment is the possibility of finding a **heap of gold**
vi. The precise definition of the task environment is given by the below **PEAS description**:

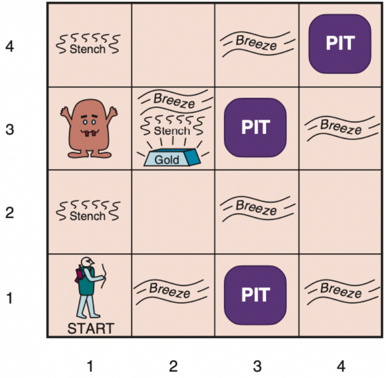


The Wumpus World: PEAS description

I. **Performance measure:**
- +1000 for **climbing out** of the cave with a 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

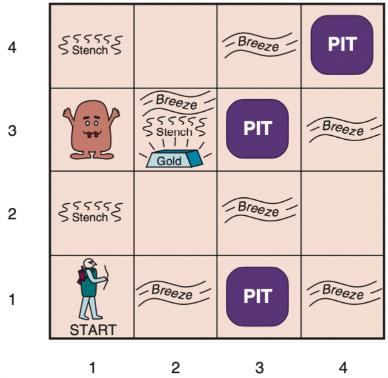
II. **Environment:** A 4×4 grid of rooms, with walls surrounding the grid
- The agent always starts in the square labelled $[1, 1]$, facing to the east
- The location of the gold and the wumpus are **chosen randomly** from the squares other than the start square
- In addition, each square other than the start can be a pit, with **probability 0.2**

III. **Actuators:** The agent can `move forward`, `turn left by` $90°$, or `turn right by` $90°$
- 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 kills the wumpus or hits a wall
  - The agent has only one arrow, so only the first `Shoot` action has any effect
- The action `Climb` can be used to climb out of the cave but only from square [1,1]



IV. **Sensors**: The agent has five sensors, each of which gives a single bit of information

- In the squares adjacent to the wumpus, the agent will perceive a `Stench`
- In the squares directly adjacent to a pit, the agent will pursue a `Breeze`
- In the square where the gold is, the agent will perceive a `Glitter`
- When an agent walks into a wall, it will receive a `Bump`
- When the wumpus is killed, it emits a `Scream` that can be perceived anywhere in the cave



The Wumpus World: Agent Inferences and Actions

i. 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]`

ii. For an agent in the environment, the **main challenge** is its initial **ignorance** of the configuration of the environment
   - Overcoming this ignorance seems to require **logical reasoning**
   - Occasionally, the agent must choose between going home **empty-handed** and **risking death** to find the gold

iii. The adjacent figure uses an **informal knowledge representation language** consisting of writing down symbols in a grid
   - The agents **initial knowledge base** contains the **rules of the environment**, in particular, it knows that it is in [1,1] and that [1,1] is safe Square

iv. The first percept is `[None, None, None, None, None]` from which the agent can **conclude** that its neighbouring squares [1,2], [2,1] are free of danger

v. Let us suppose the agent **decides** to move forward to [2,1]
   - The agent perceives a `breeze` in [2,1], so there must be a `Pit` in a neighbouring Square i.e., [2,2] and/or [3,1]
   - At this point, there is only one square that is **OK** and that has not yet been visited. So the agent will turnaround, go back to [1,1], and then proceed to [1,2]

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 moving to [2,1] and perceiving [*None*, *Breeze*, *None*, *None*, *None*].

vi. The agent perceives a Stench in [1,2], resulting in the **state of knowledge** shown in adjacent fig (a) (indicates there must be a **wumpus** nearby)
  - The wumpus cannot be in [1,1] and it cannot be in [2,2], therefore the agent can **infer** that the wumpus is in [1,3]
  - Moreover, the lack of a breeze in [1,2] **implies** that there is no pit in [2,2]. The agent then **infers** that there must be a pit in [3,1]
    - This is a fairly **difficult inference**, because it **combines knowledge** gained at different times in different places and relies on the lack of percept to **make one crucial step**
  - The agent has now proved to itself that there is neither a pit nor a wumpus in [2,3], so it is OK to move there

vii. In [2,3] the agent detects a glitter, so it should Grab the gold and then return home

viii. Note that in each case for which the agent draws a **conclusion** from the **available information**, that conclusion is guaranteed to be correct if the available information is correct
  - This is a **fundamental property** of logical reasoning

ix. In the next section, we describe how to **build logical agents** that can **represent information** and **draw conclusions** such as those described in the previous section

Figure 7.4 Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [*Stench*, *None*, *None*, *None*, *None*]. (b) After moving to [2,2] and then [2,3], and perceiving [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

Logic: Fundamental concepts

i. The knowledge base is consist of **sentences**
   - These sentences are expressed according to the **syntax** of the representation language
   - For example "$x + y = 4$" is a well-formed sentence using arithmetic syntax, whereas "$x4y+ =$" is not

ii. The logic must also define the **semantics**, or meaning of sentences
   - The semantics define the **truth** of each sentence with respect to each possible model
   - For example, the semantics for arithmetics specifies that the sentence "$x + y = 4$" is true in a model where x is 2 and y is 2, but false in a model where x is 1 and y is 1

iii. 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 $M(\alpha)$ to mean the set of all models of α

iv. The relation of logical **entailment** between sentences means - the idea that a sentence follows logically from another sentence.
   - In mathematical notation, we write

$$\alpha \models \beta$$

   - to mean that the sentence α entails the sentence β
   - The formal definition of entitlement is $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true

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

     Note that α is a stronger assertion then β
   - Example: In arithmetic; sentance $x = 0$ entails the sentence $xy = 0$

v. Consider an initial situation where the agent has detected nothing in [1,1] and a breeze in [2,1]
   - These percepts, combined with the agents knowledge of the rules of the wumpus world, constitute the KB

vi. The agent is then interested in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits
   - Each of the three squares might or might not contain a pit, there are $2^3 = 8$ possible models

vii. The KB can be thought of set of sentences that asserts all the individuals sentences
   - The KB fails in models that contradict what the agent knows
     - For example, the KB fails in any model in which [1,2] contains a pit, because there is no breeze in [1,1]

viii. Now let us consider two possible conclusions:

$$\alpha_1 = \text{“There is no pit in [1,2] ''}$$
$$\alpha_2 = \text{“There is no pit in [2,2] ''}$$

By inspection, we see the following
   - In every model in which KB is true, α_1 is also true. I.e., $KB \models \alpha_1$
   - In some models in which KB is true, α_2 is false. I.e., The agent cannot conclude that there is no pit in [2,2]

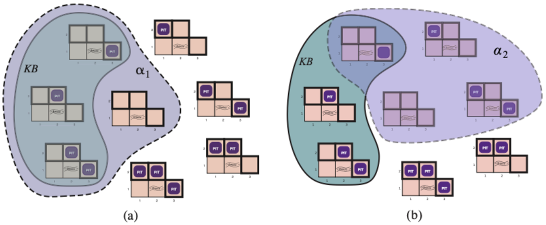


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in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

ix. The above example illustrates entitlement and also shows how the definition of entilement can be applied to derive conclusions that is, to carry out **logical inference**
   - The inference algorithm shown above is called **model checking**, because it enumerates all possible models to check that α is true in all models in which KB is true, $M(KB) \subseteq M(\alpha)$

x. **Entailment** is like the needle (α) being in the haystack (KB); **inference** is like finding it
- If an **inference algorithm** i can derive α from KB, we write

$$KB \models_i \alpha$$

Which is pronounced " α **is derived from KB by** i "

xi. An inference algorithm that derives only entiled sentences is called **sound** or **truth-preserving**
- An unsound inference procedure sequentially makes things up as it goes along
- It is easy to see that model checking, is a sound procedure

xii. **Completeness**: An inference algorithm is **complete** if it can derive any sentence that is entiled

xiii. We have described the **reasoning process** in which, if KB is true in the real world, then any sentence α derived from KB by a **sound inference** procedure is also true in the real world
- So, while an **inference process** operates on syntax i.e., **physical configuration** such as bits in a register, the process corresponds to the **real world relationship**

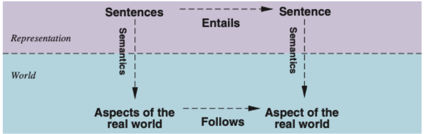


Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations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

xiv. **Grounding**: Grounding is the connection between logical reasoning processes and the real environment in which the agent exists. In particular, how do we know that KB is true in the real world ?
- For example, wumpus-world agent has a small sensor. The agent program creates a suitable sentence whenever there is a smell
- Then, whenever that sentence is in the knowledge base, it is true in the real world

Where are we ?

Propositional Logic

I. In this section we present **propositional logic**
   - We describe its **syntax**, that is the structure of the sentences and
   - it's **semantic** that is the way in which the truth of sentences is determined

II. We derive a simple, **syntactic algorithm** for **logical inference** that implements the **semantic notion** of entilement

Syntax

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

ii. The **atomic sentences** consist of a single **proposition symbol**
   - Each such symbol stands for **proposition** that can be **true or false**
   - For example we use $W_{1,3}$ to stand for the proposition that the wumpus is in [1,3]

iii. **Complex sentences** are **constructed** from simpler sentences, using parenthesis and operators called **logical connectives**. Five connectives in common use are:

   I. **¬ (not)**: A **sentence** such as
   $$\neg W_{1,3}$$
   is called the **negation** of $W_{1,3}$

   II. **∧ (and)**: A **sentence** whose main connective is $\wedge$, such as
   $$W_{1,3} \wedge P_{3,1},$$
   is called a **conjuction**

   III. **∨ (or)**: A **sentence** whose main connective $\vee$, such as
   $$(W_{1,3} \wedge P_{3,1}) \vee W_{2,2},$$
   is a **disjunction**

   IV. **⇒ (implies)**: A **sentence** such as
   $$(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$$
   is called an **implication**.
   - Its **premise** is $(W_{1,3} \wedge P_{3,1})$ and
   - its **conclusion** is $\neg W_{2,2}$

   V. **⇔ (if and only if)**: The sentence
   $$W_{1,3} \Leftrightarrow \neg W_{2,2}$$
   is a **biconditional**

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\, \mathit{Sentence} \,) \\
 & \mid & \neg\, \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Rightarrow\; \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \;\Leftrightarrow\; \mathit{Sentence} \\
 & & \\
\textsc{Operator Precedence} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

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

Semantics

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

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

- With three proposition symbols, there are $2^3 = 8$ **possible models**

ii. The semantics for propositional logic must specify **how to compute the truth** value of any sentence, given a model. This is done recursively.
- All sentences are constructed from **atomic sentences** and the **five connectives**, therefore,
- we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed there on

iii. **Atomic sentences** are easy:
- The truth value of every proposition symbol must be specified directly in the model.
- For example, in the model m_1 one given earlier $P_{1,2}$ is false

iv. For **complex sentences**, we have **five connectives**, which holds for any subsentences P and Q in any model m

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

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

v. $P \Rightarrow Q$ is saying, "`If` P `is true, I am claiming that` Q `is true; otherwise I am making no claim`"

vi. $P \Leftrightarrow Q$, is true whenever both $P \Rightarrow Q$ and $Q \Rightarrow P$ are true

A simple knowledge base

i. We can now constructive **knowledge base** for the wumpus world using the **semantics for propositional logic**

ii. We focus first on the **immutable** aspects of the wumpus world

$P_{x,y}$	is true if there is a pit in	$[x,y]$
$W_{x,y}$	is true if there is a wumpus in	$[x,y]$
$B_{x,y}$	is true if there is a breeze in	$[x,y]$
$S_{x,y}$	is true if there is a stench in	$[x,y]$
$L_{x,y}$	is true if the agent is in	$[x,y]$

iii. Further we write the following **sentences** each labelled

R_i

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

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

- A square is **breezy** if an only if there is a pit in a **neighboring square**

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

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

- The breeze **percepts** for the first two squares visited leads us to

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

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

A simple inference procedure

i. Our **goal** now is to decide whether $KB \models \alpha$
   - For Example: Is $KB \models \neg P_{1,2}$?

ii. Our first **algorithm for inference** is a **model-checking** approach i.e.,
   - **Enumerate the models**, and check that α is true in every model in which KB is true

iii. In our wumpus-world example, the relevant **proposition symbols** are $B_{1,1}, B_{2,1}, P_{1,1}, P_{1,2}, P_{2,1}, P_{2,2}$, and $P_{3,1}$
   - With **seven symbols**, there are $2^7 = 128$ **possible models**, in three of this, KB is true
     - In those **three models**, $\neg P_{1,2}$ is true, hence there is **no pit** in [1,2]
   - On the other hand, $P_{2,2}$ is true in two of the three models and fails in one, so we **cannot yet tell** whether there is a pit in [2,2]

iv. This algorithm is **sound** because it implements directly the definition of entilement, and **complete** because it works for any KB and α and always terminates
   - However, the **time complexity** of the algorithm is $\mathcal{O}(2^n)$

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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 7.9 A truth table constructed for the knowledge base given in the text. *KB* is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

Propositional Theorem Proving

I. In this section, we show how entilement can be done by **theorem proving**, i.e., applying **rules of inference directly** to the sentences in our knowledge base to **deduce a desired** sentence **without** consulting models
   - The theory of proving can be **more efficient** than model checking

II. **Inference and proofs:** This section covers inference rules that can be applied to **derive a proof**

   A. **Modus Ponens:** This is a best known rule and is written as

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

   - The **notation** means that, whenever any sentences of the form
     - $\alpha \Rightarrow \beta$ and α are **given**, then
     - the sentence β can be **inferred**
   - For example,
     - if $(\texttt{WumpusAhead} \wedge \texttt{WumpusAlive}) \Rightarrow \texttt{Shoot}$ and $(\texttt{WumpusAhead} \wedge \texttt{WumpusAlive})$ are given, then
     - `Shoot` can be **inferred**

   B. **And-Elimination:** This rule says that, from a **conjunction**, any of the conjuncts can be **inferred**

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

   - For example,
     - From $(\texttt{WumpusAhead} \wedge \texttt{WumpusAlive})$,
     - `WumpusAlive` can be **inferred**

C. We now apply the inference rules to the wumpus world. We start with knowledge base containing R_1 through R_5 and derive $\neg P_{1,2}$

$$
\begin{aligned}
R_1 : &\quad \neg P_{1,1}. \\
R_2 : &\quad B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}). \\
R_3 : &\quad B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}). \\
R_4 : &\quad \neg B_{1,1}. \\
R_5 : &\quad B_{2,1}.
\end{aligned}
$$

Apply biconditional elimination to R_2, to obtain

$$R_6 : \quad (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

Apply And-Elimination to R_6, to obtain

$$R_7 : \quad (P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}.$$

Logical equivalence for contrapositives gives

$$R_8 : \quad \neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})$$

Apply Modus Ponens with R_8 and the percept R_4, to obtain

$$R_9 : \quad \neg(P_{1,2} \vee P_{2,1})$$

Apply De Morgan's rule, giving the conclusion

$$R_{10} : \quad \neg P_{1,2} \wedge \neg P_{2,1}$$

- That is, neither [1,2] nor [2,1] contains a pit
- Any of the **search algorithms** studied earlier can be used to find a **sequence of steps** that consitutes a proof like this

III. **Proof by resolution:** Here we use a simple version of the **resolution** rule in the wumpus world

- Let us consider the steps where the agent returns from [2,1] to [1,1] and then goes to [1,2], where it **perceives** a stench, but no breeze.

$R_{11}: \quad \neg B_{1,2}.$

$R_{12}: \quad B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3}).$

By the same process that led to R_{10}, we can derive

$R_{13}: \quad \neg P_{2,2}.$

$R_{14}: \quad \neg P_{1,3}.$

Applying biconditional elimination to R_3, followed by Modus Ponens with R_5

$R_{15}: \quad (P_{1,1} \vee P_{2,2} \vee P_{3,1}).$

Using the resolution rule, R_{13} resolves the literal $P_{2,2}$ in R_{15} to give the resolvent

$R_{16}: \quad (P_{1,1} \vee P_{3,1}).$

Further, R_1 resolves $P_{1,1}$ in R_{16} to give

$R_{16}: \quad P_{3,1}.$

- These last two inference steps are examples of the **unit resolution inference rule**

Where are we ?

First-Order Logic

Introduction

I. **Propositional logic** sufficed to illustrate the **basic concepts** of logic, inference, and knowledge-based agents
   - Unfortunately, propositional logic **lacks the expressive power** to concisely describe an environment with many objects
   - For example, we were forced to write a **separate rule** about breezes and pits for each square, such as

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

   - On the other hand, in English it seems easy enough to say, "`Squares adjacent to pits are breezy`"

II. In this section, we examine **first-order logic (FOL)**, which can concisely represent much more
   - The language of first-order logic, is built around **objects and relations**
   - It can also express facts about **some or all** of the objects in the universe - Such as the statement "`Squares neighbouring to the wumpus are smelly` "

III. The primary **difference** between PL and FOL is that,

- **Propositional logic** assumes that there are **facts** that either **hold** or **do not hold** in the world
  - Each fact can be in one of the two states - `true` or `false` - and
  - Each **model** assigns true or false to each **proposition symbol**
- The **First-order logic** assumes that the world consists of **object** with **certain relations** among them that **do** or **do not hold**
  - For example

$$\forall t \ \texttt{Glitter}(t) \Rightarrow \texttt{BestAction}(\texttt{Grab}, t)$$

  - The formal models are correspondingly more **complicated** than those of propositional logic

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Syntax and Semantics of First-order Logic

Models for FOL

i. The **models** of a logical language are the **formal structures** that constitute the **possible worlds** under consideration
   - Each model **links** the vocabulary of the **logical sentences** to **elements** of the possible world

ii. The models for first order logic have **objects** in them
   - The **domain** of a model is the **set of objects** or the domain elements it contains - Every possible world must contain **at least one object**

iii. Figure shows a model with five objects:
   - `Richard the Lionheart, king of England from 1189 to 1199;`
   - `His younger brother, the evil king John, who ruled from 1199 to 1215 ;`
   - `The left leg of Richard and John` ; and
   - `a crown`.

iv. The objects in the model may be **related** in various ways

- A **relation** is just the **set of tuples of objects** that are related (A tuple is a collection of objects arranged in a fixed order)

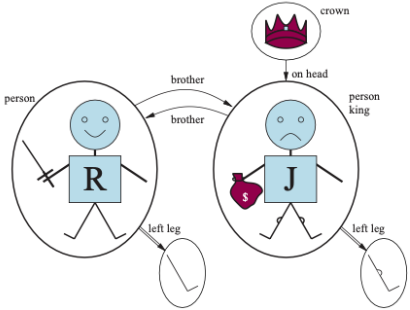


Figure 8.2 A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

v. **Binary relations:**

- The **brotherhood** relation in this model is the set
  { ⟨`Richard the Lionheart, king John` ⟩,
  ⟨`King John, Richard the Lionheart` ⟩ }

- The crown is on King John's head, so the "`on head`" **relation** contains just **one tuple**
  { ⟨`the crown, king John` ⟩ }

- Note that the "`brother`" and the "`on head`" relations are **binary relations** - i.e., they relate pairs of objects

vi. **Unary relations or properties:**

- The "`person`" **property** is true of both Richard and John
- The "`king`" **property** is true only of John and
- the "`crown`" **property** is true only of the crown

vii. **Functions**: Certain kinds of **relationships** are best considered as **functions**, in that a given object must be related to **exactly one object** in this way

- For example, each person has one left leg, so the model has a unary **left leg function**
- a mapping from a one-element tuple to an object
  ⟨`Richard the Lionheart`⟩ $\rightarrow$ `Richard's left leg`
  ⟨`King John`⟩ $\rightarrow$ `John's left leg` .

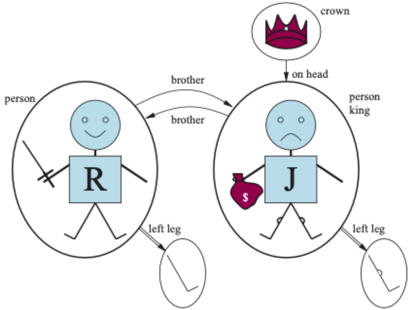


Figure 8.2 A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

Symbols and Interpretations

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate} \mid \textit{Predicate}(\textit{Term},\ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} & \rightarrow & (\ \textit{Sentence}\ ) \\
& \mid & \neg\ \textit{Sentence} \\
& \mid & \textit{Sentence} \land \textit{Sentence} \\
& \mid & \textit{Sentence} \lor \textit{Sentence} \\
& \mid & \textit{Sentence} \ \Rightarrow\ \textit{Sentence} \\
& \mid & \textit{Sentence} \ \Leftrightarrow\ \textit{Sentence} \\
& \mid & \textit{Quantifier Variable},\ldots\ \textit{Sentence} \\
\\
\textit{Term} & \rightarrow & \textit{Function}(\textit{Term},\ldots) \\
& \mid & \textit{Constant} \\
& \mid & \textit{Variable} \\
\\
\textit{Quantifier} & \rightarrow & \forall \mid \exists \\
\textit{Constant} & \rightarrow & A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\textit{Predicate} & \rightarrow & \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} & \rightarrow & \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\textsc{Operator Precedence} & : & \neg, =, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Symbols and Interpretations

i. The basic **syntactic** elements of first order logic are the symbols that stand for **objects**, **relations**, and **functions**
ii. The symbols therefore, come in three kinds
   A. **Constant symbols**, which stand for **objects**. For example: `Richard` and `John`
   B. **Predicate symbols**, which stand for **relations**. For example: `Brother`, `OnHead`, `Person`, `King`, and `Crown`
   C. **Function symbols**, which stand for **functions** . For example: `LeftLeg`
iii. A **model** in first order logic consists of a set of objects and an interpretation that maps
   - **Constant** symbols to objects,
   - **Function** symbols to functions on those objects, and
   - **Predicate** symbols to relations

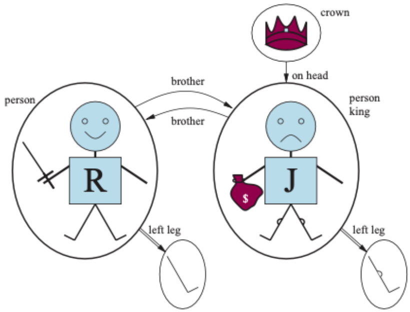


Figure 8.2 A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

iv. **Terms:** It is a logical expression that **refers** to an **object**
- **Constant symbols** are terms, but it is **NOT** always convenient to have a **distinct symbol** to name every object
    - For example: **Instead** of using a constant symbol, we use `LeftLeg(John)`
- A **complex term** is formed by a function symbol followed by a parenthesized **list of terms** as arguments to the function symbol
- Note that a **complex term** is just a complicated kind of **name**

v. For formal semantics of terms, consider a term $f(t_1, \cdots, t_n)$
- The symbol f refers to some **function** in the model,
- The argument terms $t_1, \cdots, t_n$ refer to **objects** in the domain, and
- The **term** $f(t_1, \cdots, t_n)$ as a whole refers to the **object** that is the value of the function f applied to $t_1, \cdots, t_n$

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \\
& \mid & \neg\, \mathit{Sentence} \\
& \mid & \mathit{Sentence} \land \mathit{Sentence} \\
& \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
& \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
& \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
& \mid & \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
\\
\mathit{Term} & \rightarrow & \mathit{Function}(\mathit{Term}, \ldots) \\
& \mid & \mathit{Constant} \\
& \mid & \mathit{Variable} \\
\\
\mathit{Quantifier} & \rightarrow & \forall \mid \exists \\
\mathit{Constant} & \rightarrow & A \mid X_1 \mid \mathit{John} \mid \cdots \\
\mathit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\mathit{Predicate} & \rightarrow & \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \cdots \\
\mathit{Function} & \rightarrow & \mathit{Mother} \mid \mathit{LeftLeg} \mid \cdots \\
\textsc{Operator Precedence} & : & \neg, =, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

v. **Atomic sentences:** An atomic sentence is formed from the **predicate symbol** optionally followed by a parenthesized list of terms,

- For Example:

  `Brother(Richard, John).`

  This states, under the intended interpretation, that `Richard the Lionheart is the brother of King John`

- Atomic sentences can have **complex terms** as arguments, such as

  `Married(Father(Richard), Mother(John))`

- An atomic sentence is **true** in a given moment if the **relation referred** to by the predicate symbol **holds** among the objects referred to by the arguments

vi. **Complex sentences:** We can use **logical connectives** to construct more **complex sentences**, with the same syntax and semantics as in a propositional calculus

- Here are **four sentences** that are **true** in the model under our intended interpretation

$$\neg \texttt{Brother}(\texttt{LeftLeg}(\texttt{Richard}), \texttt{John})$$

$$\texttt{Brother}(\texttt{Richard}, \texttt{John}) \wedge \texttt{Brother}(\texttt{John}, \texttt{Richard})$$

$$\texttt{King}(\texttt{Richard}) \vee \texttt{King}(\texttt{John})$$

$$\neg \texttt{King}(\texttt{Richard}) \Rightarrow \texttt{King}(\texttt{John})$$

$$
\begin{array}{rcl}
Sentence & \rightarrow & AtomicSentence \mid ComplexSentence \\
AtomicSentence & \rightarrow & Predicate \mid Predicate(Term, \ldots) \mid Term = Term \\
ComplexSentence & \rightarrow & (\ Sentence\ ) \\
& \mid & \neg\ Sentence \\
& \mid & Sentence \wedge Sentence \\
& \mid & Sentence \vee Sentence \\
& \mid & Sentence \Rightarrow Sentence \\
& \mid & Sentence \Leftrightarrow Sentence \\
& \mid & Quantifier\ Variable, \ldots\ Sentence \\
\\
Term & \rightarrow & Function(Term, \ldots) \\
& \mid & Constant \\
& \mid & Variable \\
\\
Quantifier & \rightarrow & \forall \mid \exists \\
Constant & \rightarrow & A \mid X_1 \mid John \mid \cdots \\
Variable & \rightarrow & a \mid x \mid s \mid \cdots \\
Predicate & \rightarrow & True \mid False \mid After \mid Loves \mid Raining \mid \cdots \\
Function & \rightarrow & Mother \mid LeftLeg \mid \cdots \\
\text{OPERATOR PRECEDENCE} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

vi. **Quantifiers**: First-order logic contains two standard quantifiers, called **universal** and **existential**

A. **Universal quantification ($\forall$):** This quantification makes statements about **every** object.

- "`All kings are persons`," is written in first-order logic as

$$\forall\, x \quad \texttt{King}(x) \Rightarrow \texttt{Person}(x)$$

i.e., **For all** x, if x is a king, then x is a person

- $\Rightarrow$ is the **natural** connective to use with $\forall$
- The symbol x is called a **variable**, which is a **term** all by itself and can also serve as the argument of a function
- A term with no variable is called a **ground term**
- ◦ The sentence $\forall x P$, Where P is any logical sentence, says that P is true for **every** (i.e., **each**) object x

B. **Existential quantification ($\exists$):** This quantification makes statement about **some object** without naming it

- For example, to say, that `King John has a crown on his head`, we write

$$\exists\, x \quad \texttt{Crown}(x) \land \texttt{OnHead}(x, \texttt{John})$$

i.e., **For some** x, it is a `Crown` and is `OnHead` of John

- $\exists x$ is pronounced "**There exists** an x such that $\cdots$" or "**For some** x $\cdots$"
- $\land$ is the **natural** connective to use with $\exists$
- ◦ The sentence $\exists\, x\, P$, says that P is true for at **least one** (or **some**, **a few**) object x

vii. **Nested Quantifiers:** We will often want to express more complex sentences using **multiple quntifiers**

- For example, "`Brothers are siblings`"

$$\forall\, x\ \forall\, y \quad \texttt{Brother}(x, y) \Rightarrow \texttt{Sibling}(x, y)$$

- For example, "`Everybody loves somebody`"

$$\forall\, x\ \exists\, y \quad \texttt{Loves}(x, y)$$

- For example, "`There is someone who is loved by everyone`"

$$\exists\, y\ \forall\, x \quad \texttt{Loves}(x, y)$$

viii. **Equality:** We can use the equality symbol to signify that two terms refer to the **same object**.

- For example

$$\texttt{Father}(\texttt{John}) = \texttt{Henry}$$

says that the object referred by `Father`(`John`) and object referred to by Henry are the same

Using First-Order Logic

I. **Assertions:** Sentences are **added** to knowledgebase using $\texttt{TELL}$. Such sentences and called **assertions**.

- For example, we can **assert** that $\texttt{John is a king}$, $\texttt{Richard is a person}$, and $\texttt{all Kings are persons}$

$$\texttt{TELL}(\texttt{KB}, \texttt{King}(\texttt{John})).$$

$$\texttt{TELL}(\texttt{KB}, \texttt{Person}(\texttt{Richard})).$$

$$\texttt{TELL}(\texttt{KB}, \forall x\, \texttt{King}(x) \Rightarrow \texttt{Person}(x)).$$

II. **Queries:** We can **ask questions** of the knowledge base using $\texttt{ASK}$. Questions asked with $\texttt{ASK}$ are called **Queries** or **goals**

- For example:

$$\texttt{ASK}(\texttt{KB}, \texttt{King}(\texttt{John}))$$

- Any query that is **logically entiled** by the knowledge base should be answered **affirmatively**

III. If we want to know what value of x makes the sentence $\texttt{true}$, we will need a different function, which are called $\texttt{ASKVARS}$

- For example:

$$\texttt{ASKVARS}(\texttt{KB}, \texttt{Person}(x))$$

- Such an answer is called a **substitute** or **binding list**

The wumpus world

i. Recall that the wumpus agent receives a **percept vector** with five elements

ii. The corresponding first-order sentence **stored** in the knowledge base must include both the **percept** and the **time** at which it occurred

- A typical **percept sentence** would be

$$\texttt{Percept}\,([\texttt{Stench, Breeze, Glitter, None, None}\ ], 5)$$

Here, percept is a **binary predicate**, and $\texttt{Stench}$ and so on are constants placed in a **list**

iii. The **actions** in the wumpus world can be represented by **logical terms**:

$$\texttt{Turn}(\texttt{Right}), \texttt{Turn}(\texttt{Left}), \texttt{Forward}, \texttt{Shoot}, \texttt{Grab}, \texttt{Climb}.$$

iv. To **determine** which is **best**, the agent program **executes** the query

$$\texttt{ASKVARS}(\texttt{KB}, \texttt{BestAction}(a, 5))$$

Which returns of **binding list** such as $\{a/\texttt{Grab}\}$

v. The **raw percept** data implies certain facts about the current state

$$\forall\, t, s, g, w, c \quad \texttt{Percept}([s, \texttt{Breeze}, g, w, c], t) \Rightarrow \texttt{Breeze}(t)$$

$$\forall\, t, s, g, w, c \quad \texttt{Percept}([s, \texttt{None}, g, w, c], t) \Rightarrow \neg\texttt{Breeze}(t)$$

$$\forall\, t, s, g, w, c \quad \texttt{Percept}([s, b, \texttt{Glitter}, w, c], t) \Rightarrow \texttt{Glitter}(t)$$

$$\forall\, t, s, g, w, c \quad \texttt{Percept}([s, b, \texttt{None}, w, c], t) \Rightarrow \neg\texttt{Glitter}(t)$$

These rules exhibit a trivial form of the reasoning process called **perception**

vi. Simple **reflex behaviour** can also be implemented by **quantified implication sentences**, for examples

$$\forall\, t \quad \texttt{Glitter}(t) \Rightarrow \texttt{BestAction}(\texttt{Grab}, t)$$

vii. Further, to **represent the environment**, it is better to use **complex term** in which the row and column appear as integers; for example, use the list term $[1, 2]$

viii. **Adjacency** of any two squares can be defined as

$$\forall\, x, y, a, b \;\; \texttt{Adjacent}\,([x, y], [a, b]) \Leftrightarrow (x = a \;\wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \;\wedge (x = a - 1 \vee x = a + 1))$$

ix. It is simpler to use a **unary perdicate** `Pit` that is `true` of squares containing pit

x. Finally, since there is exactly one wumpus, a constant `Wumpus` can be used

xi. The agent's location changes over time, so we write $\texttt{At}(\texttt{Agent}, s, t)$ to mean that the agent is at square s at time t

xii. We can fix the wumpus to a specific location forever with $\forall\, t\; \texttt{At}(\texttt{Wumpus}, [1, 3], t)$

xiii. We can then say that objects can be at only one location at a time

$$\forall\, x, s_1, s_2, t \;\; \texttt{At}(x, s_1, t) \wedge \texttt{At}(x, s_2, t) \Rightarrow s_1 = s_2.$$

xiv. Given its current location, the agent can **infer** properties of the square from properties of its **current percept**

- For example, if the agent is at a square and **perceives a breeze**, then that square is breezy

$$\forall\, s, t \;\; \texttt{At}(\texttt{Agent}, s, t) \wedge \texttt{Breeze}(t) \Rightarrow \texttt{Breezy}(s).$$

- It is useful to know that a square is `breezy` because we know that the pits cannot move about. Notice that breezy has no time argument

xv. Having discovered which places are `breezy` (smelly), `not breezy` (not smelly), the agent can **deduce** where the pits (the wumpus) are (is) using just one first-order logic axiom

$$\forall\, s \;\; \textbf{Breezy}(s) \Leftrightarrow \exists\, r \;\; \textbf{Adjacent}(r, s) \wedge \textbf{Pit}(r).$$

xvi. The axiom for the **arrow** becomes

$$\forall\, t \;\; \textbf{HaveArrow}(t + 1) \Leftrightarrow (\textbf{HaveArrow}(t) \wedge \neg\textbf{Action}(\texttt{Shoot}, t))$$

Summary

I. Knowledge representation languages should be **declarative**, **compositional**, **expressive**, **context independent**, and **unambiguous**

II. While **propositional logic** commits only to the existence of **facts**, **first-order logic** commits to the existence of **objects** and **relations** and thereby gain expressive power

III. The syntax of first-order logic **builds on** that of propositional logic. It adds terms to represent **objects**, and has universal and existential **quantifiers** to construct assertions about all or some of the possible values of the variables

IV. A possible world, or model, for first-order logic includes
   i. a set of objects and
   ii. an interpretation that maps
      - **constant symbols** to objects,
      - **predicate symbols** to relations among objects, and
      - **function symbols** to functions on objects

V. An **atomic sentence** is true only when the relation named by the **predicate** holds between the objects named by the terms
   - Extended interpretations, which map **quantifier** variables to objects in the model, define the truth of quantified sentences

VI. Developing a **knowledge base** in first-order logic requires a **careful process** of analysing the domain, choosing a vocabulary, and encoding the axioms required to support the desired inferences

Where are we ?

Forward Chaining

First-order definite clause

i. The forward chaining algorithm can be used on knowledge bases of **first-order definite clauses**

ii. **First order definite clauses** are disjunctions of literals of which exactly **one is positive**

- That means a definite clause is either **atomic** or
- is an implication of the form $\texttt{Antecedent} \Rightarrow \texttt{Consequent}$
  - where the `antecedent` is a conjunction of positive literal and
  - the `consequent` is a single positive literal
- Typical first-order definite clauses look like this:

$$\texttt{King}(\texttt{John}),$$
$$\texttt{Greedy}(y),$$
$$\texttt{King}(x) \land \texttt{Greedy}(x) \Rightarrow \texttt{Evil}(x),$$

  - Note that if you see a variable like x in a definite clause, that means there is an implicit universal quantifier $\forall\, x$
  - i.e., $\texttt{Greedy}(y)$ is interpreted as "`everyone is greedy`"

iii. Let us put definite clauses to work in representing the following problem

- `The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is an American.`

iv. First, we will represent these facts as first order definite clauses:

- " $\cdots$ `it is a crime for an American to sell weapons to hostile nations`"
$$\texttt{American}(x) \wedge \texttt{Weapon}(y) \wedge \texttt{Sells}(x, y, z) \wedge \texttt{Hostile}(z) \Rightarrow \texttt{Criminal}(x) \tag{1}$$
- "`Nono` $\cdots$ `has some missiles`". The sentence $\exists\, x \texttt{Owns}(\texttt{Nono}, x) \wedge \texttt{Missile}(x)$ is transformed into two definite clauses by introducing a new constant M_1
$$\texttt{Owns}(\texttt{Nono}, M_1) \tag{2}$$
$$\texttt{Missile}(M_1) \tag{3}$$
- "`All of its missiles were sold to it by Colonel West`"
$$\texttt{Missile}(x) \wedge \texttt{Owns}(\texttt{Nono}, x) \Rightarrow \texttt{Sells}(\texttt{West}, x, \texttt{Nono}). \tag{4}$$
- We will also need to know that missiles are weapons
$$\texttt{Missile}(x) \Rightarrow \texttt{Weapon}(x) \tag{5}$$
- And we must know that an enemy of America counts as hostile
$$\texttt{Enemy}(x, \texttt{America}) \Rightarrow \texttt{Hostile}(x) \tag{6}$$
- "`West, who is American` $\cdots$"
$$\texttt{American}(\texttt{West}) \tag{7}$$
- "`The country Nono, an enemy of America` $\cdots$ "
$$\texttt{Enemy}(\texttt{Nono}, \texttt{America}) \tag{8}$$

A simple forward-chaining algorithm

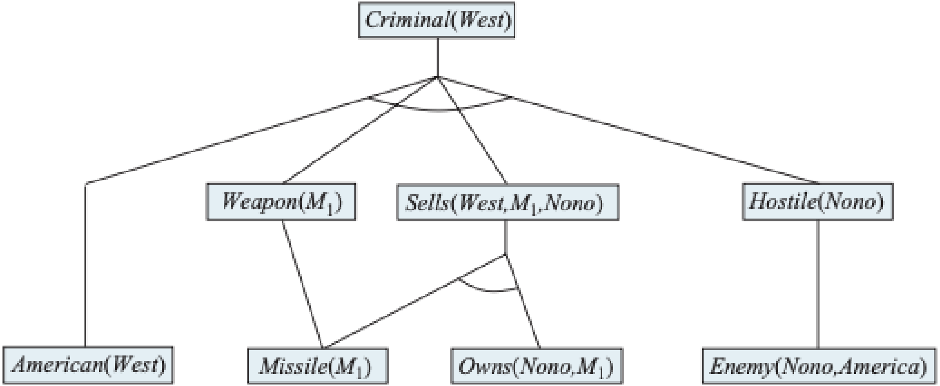
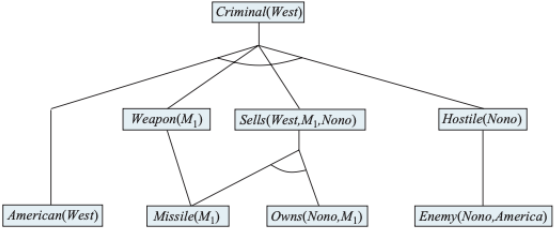


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

i. Figure shows a simple **forward chaining inference algorithm** `FOL-FC-ASK`(KB, α)
   - Starting from the **unknown facts**, it triggers **all the rules** whose premises **satisfied**, adding their **conclusions** to the known facts
   - The process repeats until the **query is answered**

ii. The implication sentences available for chaining are (1), (4), (5) and (6). Two iterations are required:
   - On the first iteration, rule (1) has unsatisfied premises.
     - Rule (4) is satisfied with $\{x/M_1\}$, and `Sells`(`West`,M_1,`Nono`) is added.
     - Rule (5) is satisfied with $\{x/M_1\}$, and `Weapon`(M_1) is added.
     - Rule (6) is satisfied with $\{x/\texttt{Nono}\}$, and `Hostile`(`Nono`) is added.
   - On the second iteration, rule (1) is satisfied with $\{x/\texttt{West}, y/M_1, z/\texttt{Nono}\}$, and the inference `Criminal`(`West`) is added



iii. `FOL-FC-ASK` is easy to analyse.
   - First, it is **sound**, because every inference is just an application of generalised Modus Ponens, which is **sound**
   - Second, it is **complete** for definite clause knowledge base, that is, it answers every query whose answers are entiled by any knowledge base of definite clause

Backward Chaining

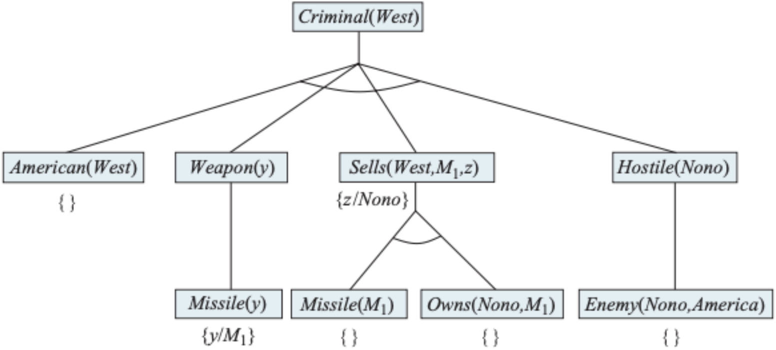


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal*(*West*), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally *Hostile*(z), z is already bound to *Nono*.

i. The backward chaining algorithms work **backwards** from the goal, chaining through rules to find known facts that support the proof

ii. `FOL-BC-ASK`(`KB`,`goal`) will be proved if the knowledge base contains a rule of the form `lhs` $\Rightarrow$ `goal`
   - where `lhs` (left-hand-side) is a list of conjuncts

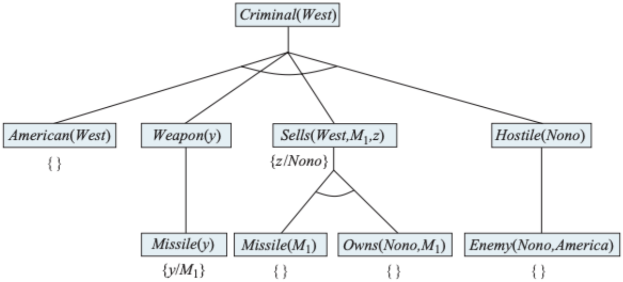
iii. An atomic fact like `American(West)` is considered as a clause whose `lhs` is the empty list

iv. A query `Person`(x) could be proved with the substitution $\{x/$`John`$\}$ as well as with $\{x/$`Richard`$\}$

v. Backward chaining is a kind of AND/OR search
   - The OR part because the goal query can be proved by any rule in the knowledge base, and
   - the AND part because all the conjuncts in the `lhs` of a clause must be proved

vi. Figure below is the proof tree for deriving `Criminal(West)` from the sentences (1) through (7)

vii. Backward chaining, is clearly a **depth-first** search algorithm
   - its space requirements are **linear** in the size of the proof .
   - However, unlike forward chaining, it suffers from problems with **repeated** states and **incompleteness**



Where are we ?

Summarizing uncertainity

i. Agents in the real world need to handle **uncertainity**
   - An agent may never know for sure what state it is in, or where it will end up after a sequence of action

ii. For example, diagnosing a dental patients to take almost always involves uncertainity
   - Consider the following simple rule:

$$\texttt{Toothache} \Rightarrow \texttt{Cavity}.$$

   - However, **not all** patients with toothaches have cavities, i.e, might have several other problems

$$\texttt{Toothache} \Rightarrow \texttt{Cavity} \vee \texttt{GumProblem} \vee \texttt{Abscess} \cdots .$$

   - Further trying to turn the rule into a causal rule, that is

$$\texttt{Cavity} \Rightarrow \texttt{Toothache}.$$

     is also **not right**; as not all cavities cause pain

iii. Trying to use logic with a domain with uncertainity like medical diagnosis fails for 3 main reasons
   I. **Laziness**: It is too much work to list the complete set of consequences to make an exceptionless rule
   II. **Theoretical ignorance:** Medical science has no complete theory for the domain
   III. **Practical ignorance:** We might be uncertain about a particular patient because not all the tests have been done

iv. The agents knowledge can at best provide only a **degree of belief** in the relevant sentences
   - The main tool for dealing with degrees of belief is **probability theory**
   - *The theory of probability provides a way of summarising the uncertainty that comes from our laziness and ignorance*
   - For example, we say
     - `"The probability that the patient has a cavity, given that she has a toothache, is 0.8"`
     - `" The probability that the patient has a cavity, given that she has toothache and history of gum disease, is 0.4 "`
     - Note that the statements do not **contradict** each other; it is a separate assertion about a different knowledge state

Basic Probability Notation

i. **Sample space**: In probability theory, the set of all possible worlds is called a **sample space**
- The possible worlds are **mutually exclusive** and **exhaustive**
- For example, if we are about to roll two dice, there are 36 possible worlds to consider: i.e., sample space Ω is
$$\Omega = \{(1,1),(1,2),\cdots,(6,6)\}$$
and ω refers to elements of the space

ii. **Probability model**: A fully specified probability model associates a numerical probability $P(\omega)$ with each possible world. Where
$$0 \leq P(\omega) \leq 1 \qquad \text{for every } \omega, \text{and}$$
$$\sum_{\omega \in \Omega} P(\omega) = 1.$$

iii. **Event**: Probabilistic assertions and queries or not usually about particular possible worlds, but about sets of them i.e., called **events** in probability theory, and **proposition** in logic language
- For example the probability that the two dice add up to 11, or the probability that doubles are rolled, and so on
- The probability associated with a proposition is defined to be the sum of probabilities of the worlds in which it holds: i.e., for any proposition ϕ
$$P(\phi) = \sum_{\omega \in \phi} P(\omega)$$
   - When rolling fair dice, we have
$$\begin{aligned} P(\texttt{Total} = 11) &= P((5,6)) + P((6,5)) \\ &= 1/36 + 1/36 = 1/18 \\ P(\texttt{doubles}) &= 1/4 \end{aligned}$$

iv. **Unconditional probability or Prior probability**: Probabilities such as $P(\texttt{Total} = 11)$, $P(\texttt{doubles})$ are called **unconditional** or **prior** probabilities; they refer to degrees of belief in propositions in the **absence of any other information**

v. **Evidence**: Most of the time, however we have some information, usually called **evidence**
   - For example, the first die may already be showing a 5 and we are waiting for the other one to stop spinning

vi. **Conditional probability or Posterior probability**: When some evidence is available, we are interested in the **conditional** or **posterior** probability of rolling `doubles` given that the first die is a 5, i.e.,

$$P(\texttt{doubles} \mid \texttt{Die}_1 = 5)$$

   - Further the assertion that $P(\texttt{cavity}|\texttt{toothache}) = 0.6$,
     - Does not mean "Whenever `toothache` is true, conclude that `cavity` is true with probability 0.6 ", rather
     - it mean "Whenever `toothache` is true and we have *no further information*, conclude that `cavity` is true with probability 0.6"
   - Mathematically, conditional probability is or defined in terms of unconditional probabilities as,

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)}$$

   that is

$$P(\texttt{doubles} \mid \texttt{Die}_1 = 5) = \frac{P(\texttt{doubles} \wedge \texttt{Die}_1 = 5)}{P(\texttt{Die}_1 = 5)}$$

vii. **Product rule**: The definition of conditional probability, can be written in a different form called the **product rule**:

$$P(a \wedge b) = P(a \mid b)\, P(b)$$

It comes from the fact that for a and b to be true, we need b to be true, and we also need a to be true given b

Propositions in probability assertions

i. **Random variable**: Variables in probability theory are called as **random variables**. Example: `Total`, $\texttt{Die}_1$
   - Every random variable is a function that maps from the domain of possible worlds Ω to some range

ii. **Range**: The **range** is the set of possible values that a random variable can take on
   - The range of `Total` for two dice is the set $\{2, \cdots, 12\}$
   - and the range for $\texttt{Die}_1$ is $\{1, \cdots, 6\}$.
   - A boolean random variable has the range $\{true, false\}$
     - For example, the proposition that doubles are rolled can be written as `Doubles` = true
     - Propositions are abbreviated simply as, `Doubles` or $\neg$`Doubles`

- Further, ranges can be set of arbitrary tokens, for example
  - the range of `Age` can be the set $\{\texttt{infant}, \texttt{teen}, \texttt{adult}\}$, and
  - the range of `Weather` might be $\{\texttt{sun, rain, cloud, snow}\}$
- We can **combine** elementary prepositions by using the connectives of proportional logic
  - For example can express " `The probability that the patient has a cavity, given that she is a teenager with no tooth one is 0.1` " as follows

$$P(\texttt{cavity} \mid \neg\texttt{toothache} \wedge \texttt{teen}) = 0.1$$

$$P(\texttt{cavity} \mid \neg\texttt{toothache}, \texttt{teen}) = 0.1$$

iii. **Probability distribution**: The **probability distribution** $\mathbf{P}$, of a random variable is an assignment of a probability for each possible value of that random variable

- For Example:

$$\mathbf{P}(\texttt{Weather}) = \langle 0.6, 0.1, 0.29, 0.01 \rangle$$

iv. **Joint probability distribution**: A **Joint probability distribution** is a distribution on multiple variables

- For example, $\mathbf{P}(\texttt{Weather, Cavity})$ denotes the probabilities of all combinations of the values of `Weather` and `Cavity`
- This is a 4×2 table of probabilities

v. **Full joint probability distribution**: A probability model is completely determined by the joint distribution for all of the random variables - the so-called **full joint probability distribution**

- For example, joint distribution $\mathbf{P}(\texttt{Cavity, Toothache, Weather})$ can be represented as a $2 \times 2 \times 4$ table with 16 entries
- The fully joint distribution **suffices** to calculate the probability of every proposition by using the product rule

vi. **Probability of negation**: We can derive the relationship between the probability of a proposition and the probability of its **negation** as

$$P(\neg a) \ = \ 1 - P(a)$$

vii. **Inclusion-exclusion principle**: The probability of a **disjunction** is also sometimes called the inclusion-exclusion principle and is given by

$$P(a \vee b) \ = \ P(a) + P(b) - P(a \wedge b)$$

Inference using Full joint distribution

i. **Probabilistic inference Query**: The method of computation of **posterior probabilities** for **query** propositions given **evidence** is called as **probabilistic inference**

- The full joint distribution is used as the **knowledge base** to derive answers to all the questions
- The $2 \times 2 \times 2$ table below lists a full joint distribution for a **simple example** consisting of just three boolean variables `Toothache`, `Cavity`, and `Catch` (the dentist's steel probe that catches in the tooth)

	toothache		*¬toothache*	
	catch	*¬catch*	*catch*	*¬catch*
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

Figure 12.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

- Note that to calculate the probability of **any proposition**, simply **identify** those possible worlds in which the **proposition** is true and **add up** their probabilities
  - For example there are six possible worlds in which `cavity` $\vee$ `toothache` holds:

$$P(\texttt{cavity} \vee \texttt{toothache}) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28$$

ii. **Marginal probability**: Adding the entries in the first row gives the **unconditional** or **marginal probability** of `cavity`

$$P(\texttt{cavity}) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2$$

- This process is called **marginalization**, or summing out - Because we sum up the probabilities for each possible value of the other variables, thereby taking them out of the equation

iii. **Marginalization**: The general marginalization rule for any set of variables $\mathbf{Y}$ and $\mathbf{Z}$ is given by

$$\mathbf{Y} = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y}, \mathbf{Z} = \mathbf{z})$$

- For example, we can obtain the distribution as $\mathbf{P}(\texttt{Cavity})$, as follows

$$\begin{aligned}\mathbf{P}(\texttt{Cavity}) = &\ \mathbf{P}(\texttt{Cavity, toothache, catch}) + \\ &\ \mathbf{P}(\texttt{Cavity, toothache, } \neg \texttt{ catch}) + \\ &\ \mathbf{P}(\texttt{Cavity, } \neg \texttt{ toothache, catch}) + \\ &\ \mathbf{P}(\texttt{Cavity, } \neg \texttt{ toothache, } \neg \texttt{ catch}) \\ = &\ \langle 0.108, 0.016 \rangle + \langle 0.012, 0.064 \rangle + \\ &\ \langle 0.072, 0.144 \rangle + \langle 0.008, 0.576 \rangle \\ = &\ \langle 0.2, 0.8 \rangle\end{aligned}$$

iv. **Conditioning**: Using the product rule, we can replace $\mathbf{P}(\mathbf{Y}, \mathbf{z})$ in the above equation with $\mathbf{P}(\mathbf{Y} \mid \mathbf{z})\mathbf{P}(\mathbf{z})$, and obtain a rule called **conditioning** :

$$\mathbf{Y} = \sum_{\mathbf{z}} \mathbf{P}(\mathbf{Y} \mid \mathbf{z})\mathbf{P}(\mathbf{z})$$

- In most cases, we are interested in computing **conditional probabilities** of some variables, given **evidence** about others
- For example, we can compute the conditional probability $P(\texttt{cavity}|\texttt{toothache})$

$$\begin{aligned}P(\texttt{cavity}|\texttt{toothache}) &= \frac{P(\texttt{cavity} \wedge \texttt{toothache})}{P(\texttt{toothache})} \\ &= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} \\ &= 0.6\end{aligned}$$

v. Given the **full joint distribution**, the answers to all the probabilistic queries for discrete variables can be **derived**
- However for a domain described by n boolean variables, it requires an input **table of size** $\mathcal{O}(2^n)$ and takes $\mathcal{O}(2^n)$ **time to process** the table
- In a realistic problem we could easily have $n = 100$, requiring a table with $2^{100} \approx 10^{30}$ entries, which is **impractical**

vi. **Independence:** Two propositions a and b are called **independent** (or marginal independent) if the following properties holds:
$P(a|b) = P(a)$ or $P(b|a) = P(b)$ or $P(a \wedge b) = P(a)P(b)$

Where are we ?

Bayes Theorem or Bayes' Rule

i. The **product rule** can actually be written in two forms :

$$P(a \wedge b) = P(a|b)\,P(b) \quad \text{and}$$
$$= P(b|a)\,P(a)$$

- Equating the two right hand sides, we get an equation called **Bayes' Rule** (also Bayes' law or Bayes' theorem)

$$P(b|a) = \frac{P(a|b)\,P(b)}{P(a)}$$

- The more general case of Bayes' rules for multi-valued variables can be written in the $\mathbf{P}$ notation as

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\,\mathbf{P}(Y)}{\mathbf{P}(X)}$$

ii. It allows us to **compute** the single time $P(b|a)$ in terms of three terms: $P(a|b)$, $P(b)$ and $P(a)$

- There are many cases where **we do have good** probability estimates for these three numbers and **need to compute** the fourth

iii. Often, we perceive as **evidence** the `effect` of some unknown `cause` and we would like to determine the `cause`. In that case, Bayes rule becomes

$$P(\texttt{cause}|\texttt{effect}) = \frac{P(\texttt{effect}|\texttt{cause})\,P(\texttt{cause})}{P(\texttt{effect})}$$

iv. **Causal** and **Diagnostic**:

- The conditional probability $P(\texttt{effect}|\texttt{cause})$ quantifies the relationship in the causal direction
- Whereas $P(\texttt{cause}|\texttt{effect})$ describes the **diagnostic** direction.
- For example, the doctor knows $P(\texttt{symptoms}|\texttt{disease})$ and wants to derive a diagnosis, $P(\texttt{disease}|\texttt{symptoms})$

Example

v. Consider a doctor knows that the disease meningitis causes a patient to have a stiff neck say 70% of the time. The doctor also knows some **unconditional** facts, that
   - the **prior probability** that any patient has meningitis $P(m)$ is $1/50,000$ and
   - The **prior probability** that any patient has a stiff neck $P(s)$ is 1%

vi. Then we have

$$
\begin{aligned}
\text{Given:} \quad & P(s|m) = 0.7 \\
\text{Given:} \quad & P(m) = 1/50000 \\
\text{Given:} \quad & P(s) = 0.01 \\
\text{Compute:} \quad & P(m|s) = \frac{P(s|m)\,P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} \\
& = 0.0014
\end{aligned}
$$

   - That is, we expect only 0.14% of patients with stiff neck to have meningitis

Text Books

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*.
Third Edition, Prentice-Hall, 2009.

[2] S. N. Elaine Rich, Kevin Knight, *Artificial Intelligence*.
Third Edition, The McGraw Hill Publications, 2009.

[3] G. F. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*.
Sixth Edition, Pearson Education, 2009.

Thank you