Problem solving by searching

Problem Solving agents: Intelligent agents are supposed to maximize their performance measure. As we mentioned , achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it.

- **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider, given a goal.
- **Known knowledge**

Ex: Imagine an agent in the city of **Mumbai**, enjoying a touring holiday. The agent's performance measure contains many factors: it wants to improve its insights, enjoy the nightlife, see sunrise and so on.

- Now, suppose the agent has a nonrefundable ticket to fly out to **Delhi** the following day. Three roads lead out of Delhi. None of these achieves the goal, so unless the agent is familiar with the geography of Mumbai, it will not know which road to follow.

Well-defined problems and solutions:

A **problem** can be defined formally by five components:

- The **initial state** that the agent starts in. For example, the initial state for our agent is Mumbai.
- A description of the possible **actions** available to the agent. Given a particular state s, returns the set of actions that can be executed in s. We say that each of these actions is applicable in s.
- Description of what each action does; the formal name for this is the **transition model**, specified by a function RESULT(s, a) that returns the state that results from action a in state s.

- Together, the initial state, actions, and transition model implicitly define the **state space** of the problem
- The state space forms a directed network or **graph** in which the **nodes** are **states** and the **links between nodes** are **actions**.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test**, which determines whether a given state is a goal state, agent chooses a cost function that reflects its own performance measure
- A **path cost** function assigns a numeric cost to each path.

• **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

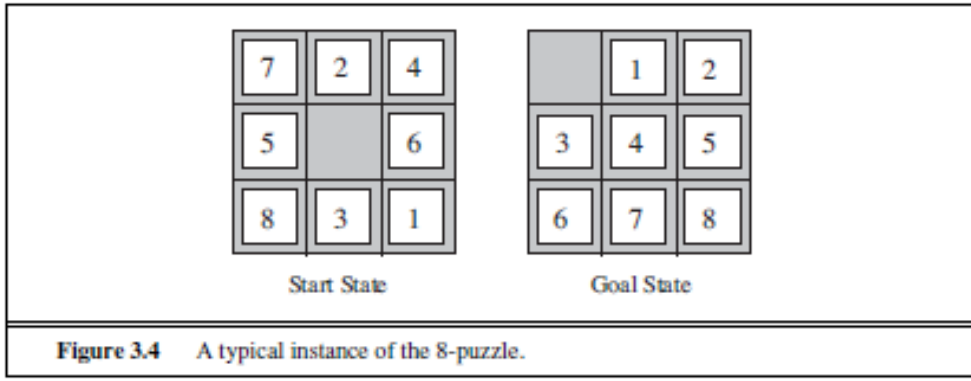


Figure 3.4 A typical instance of the 8-puzzle.

• **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states

• **Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.

Transition model: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state the resulting state has the 5 and the blank switched.

• **Goal test**: This checks whether the state matches the goal configuration shown

There are two types of searching techniques :

1. **Uninformed Search**

2. **Informed Search**

- **Uninformed:** These algorithms do not know anything about what they are searching for and where they should search for it.
- Uninformed searching takes a lot of time to search as it doesn’t know where to go and where the best chances of finding the element are.
- Ex: Linear Search, Binary Search, Depth-First Search, & Breadth-First Search.

- **Informed:** The algorithm is aware of where the best chances of finding the element are and the algorithm heads that way!
- **Heuristic** search is an informed search technique.
- A heuristic value tells the algorithm which path will provide the solution as early as possible.

8 Puzzle problem:

1	2	3
	4	6
7	5	8

1	2	3
4	5	6
7	8	

N-Puzzle or sliding puzzle is a popular puzzle that consists of N tiles where N can be 8, 15, 24, and so on. In this example N = 8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns.

The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in a single empty space and thus achieving the Goal configuration.

Rules for solving the puzzle.

- Instead of moving the tiles in the empty space, we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions viz.,

1. Up

2.Down

3. Right or

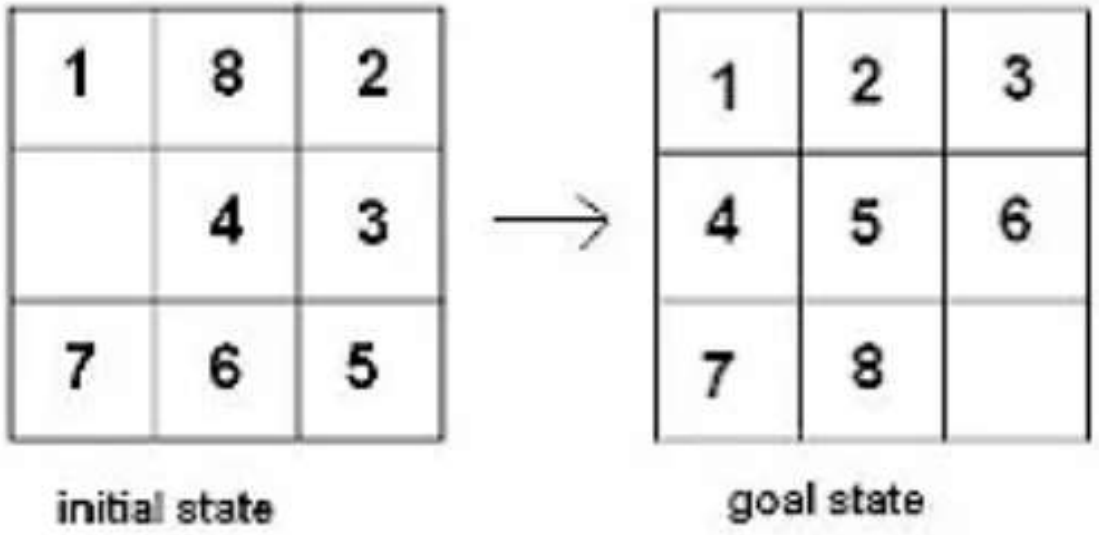
4. Left

Given an initial state of a 8-puzzle problem and final state to be reached

Find the most cost-effective path to reach the final state from initial state using A* Algorithm.

Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

Given an initial state of a 8-puzzle problem and final state to be reached



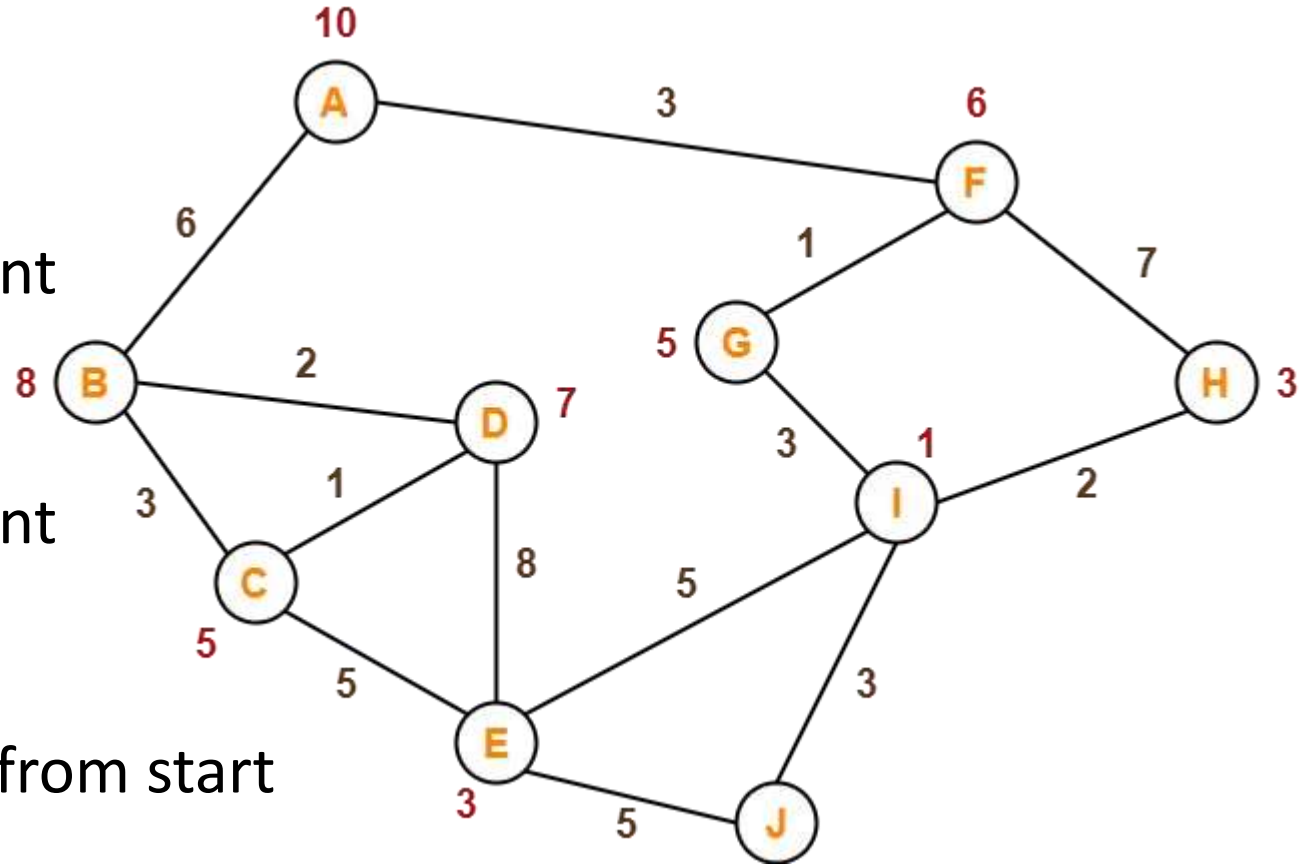
Find the most cost-effective path to reach the final state from initial state using A* Algorithm.

Consider g(n) = Depth of node and h(n) = Number of misplaced tiles.

Consider the following graph-

The numbers written on edges represent the distance between the nodes.

The numbers written on nodes represent the heuristic value.

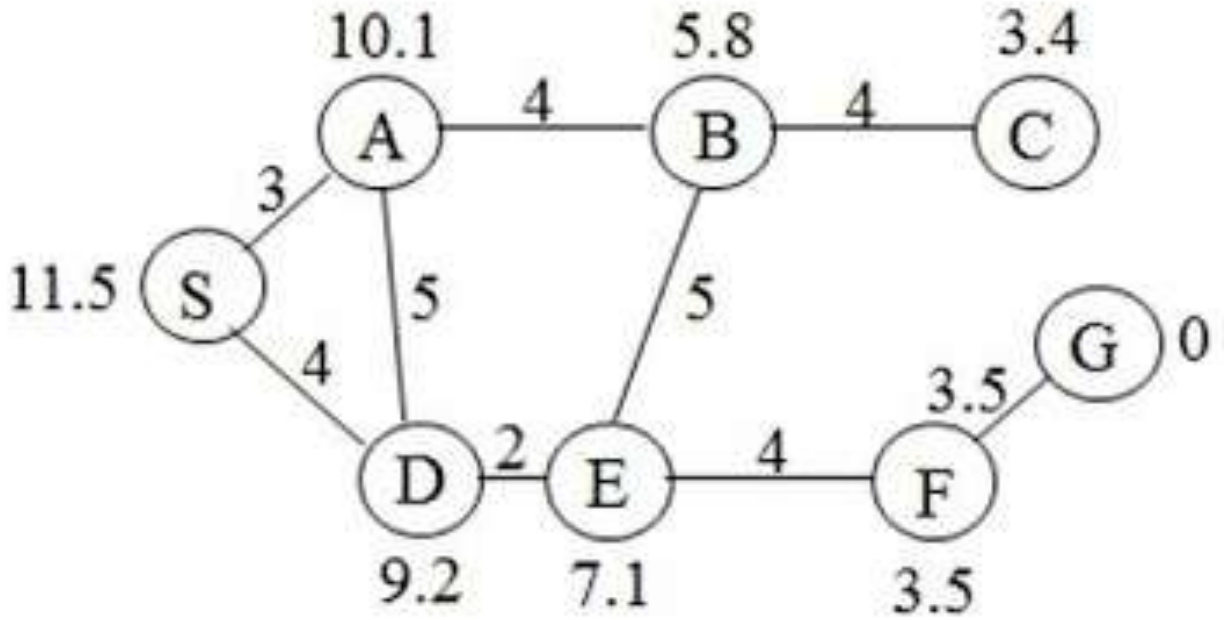


Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

Consider the following graph-

The numbers written on edges represent the distance between the nodes.

The numbers written on nodes represent the heuristic value.



Find the most cost-effective path to reach from start state A to final state G using A* Algorithm.

- **Bidirectional search:** It is an uninformed search. The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

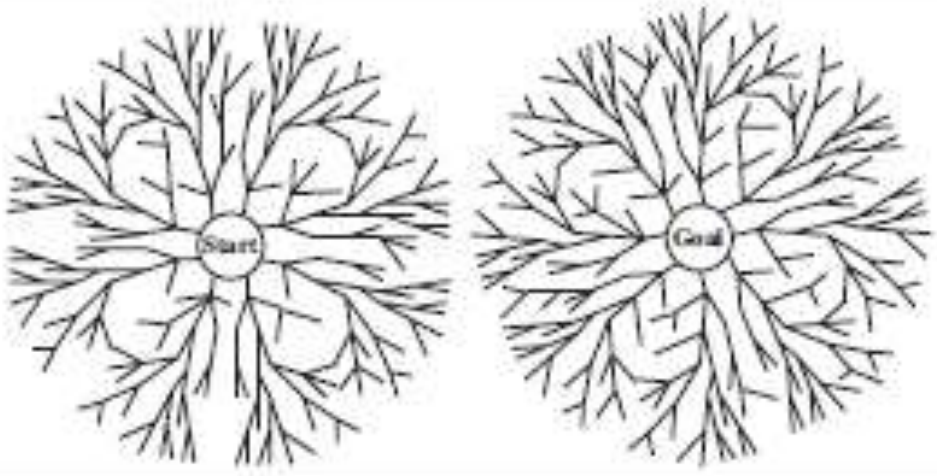
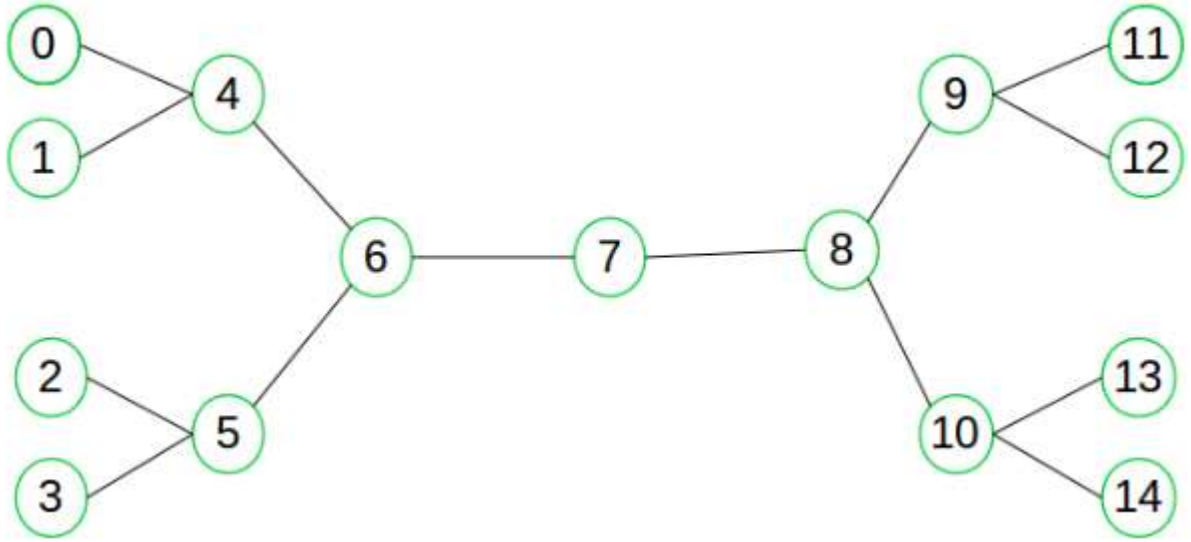


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

It runs two simultaneous search –

- Forward search from source/initial vertex toward goal vertex
- Backward search from goal/target vertex toward source vertex

Ex:



When to use bidirectional approach?

- Both initial and goal states are unique and completely defined.
- The branching factor is exactly the same in both directions.

Informed Search: A* search: Minimizing the total estimated solution cost

- The most widely known form of best-first search is called **A∗** 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:

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

g(n) = path cost from the start node to node n, and

h(n) = is the estimated cost of the cheapest path from n to the goal,

we have f(n) = estimated cost of the cheapest solution through n.

- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). A∗ search is both complete and optimal.

Ex:

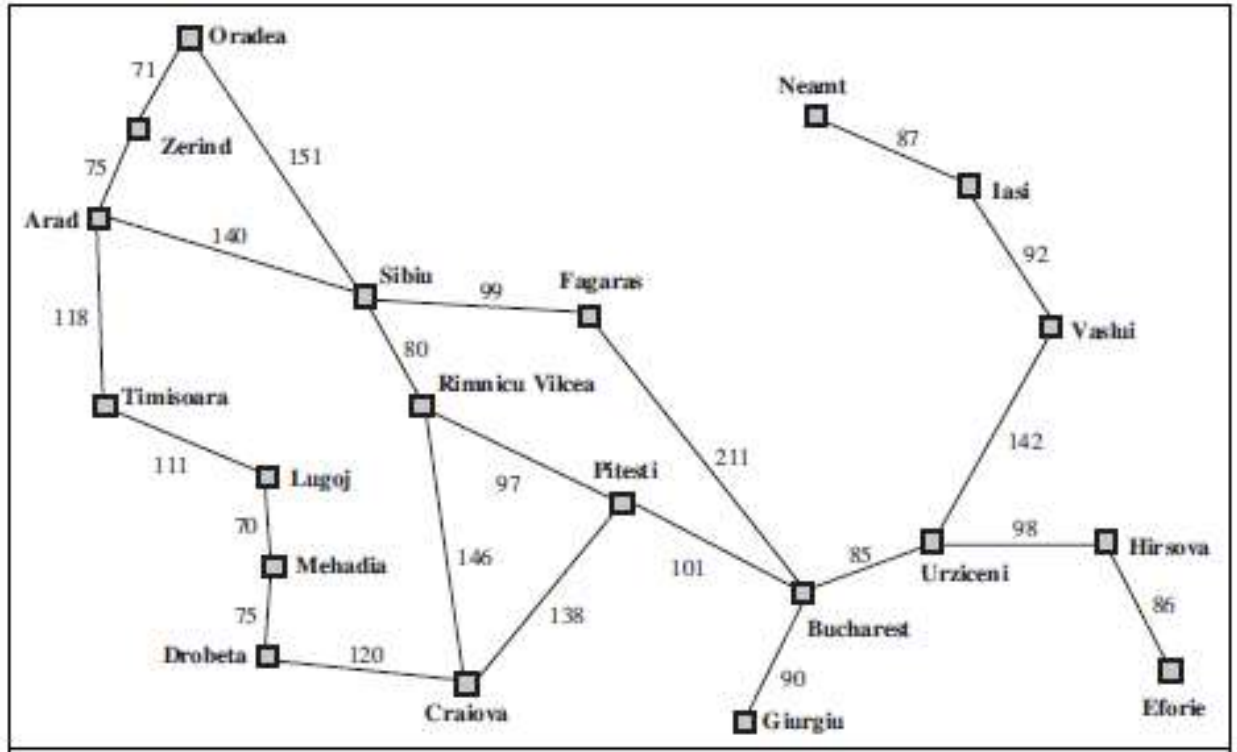
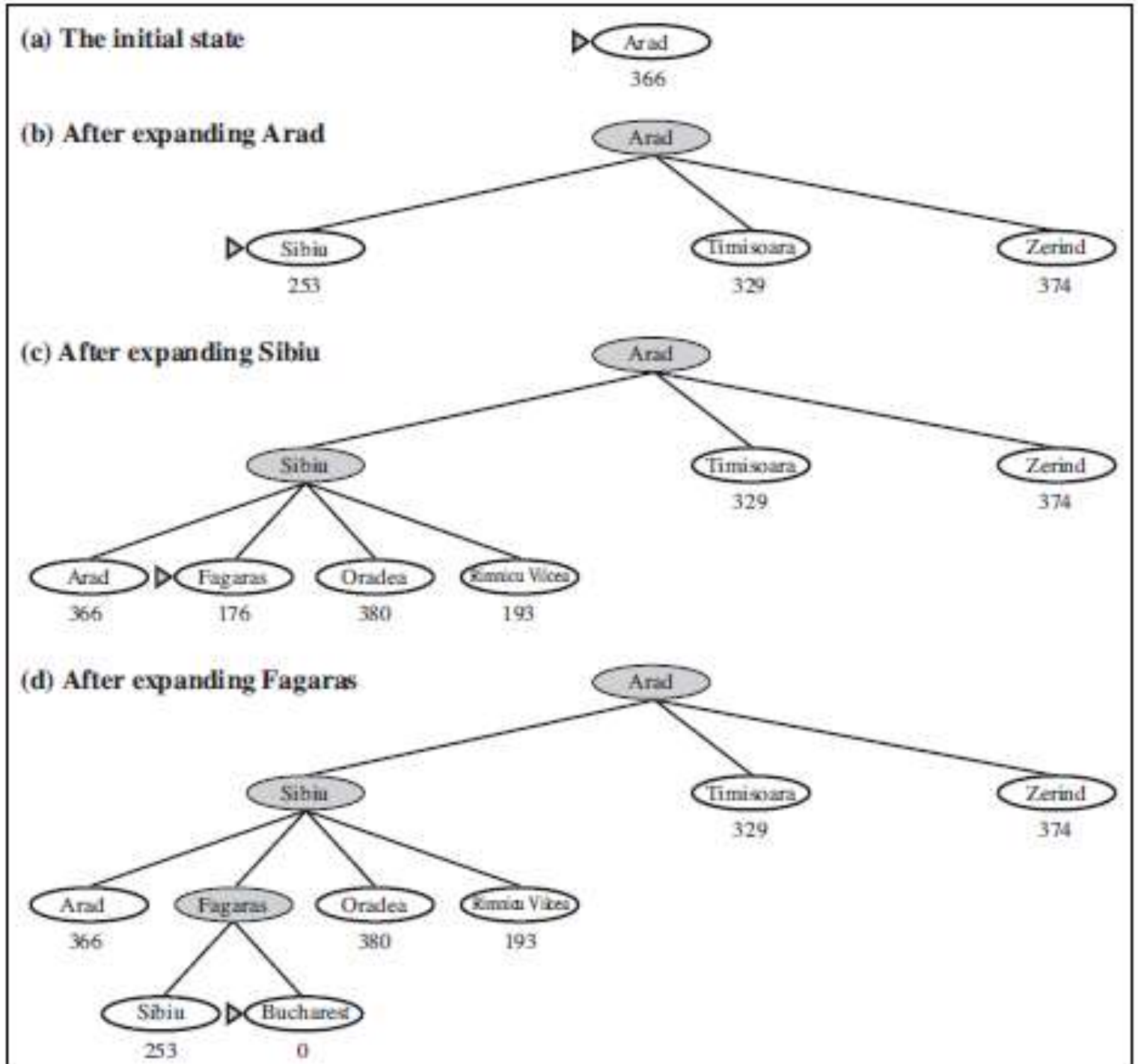


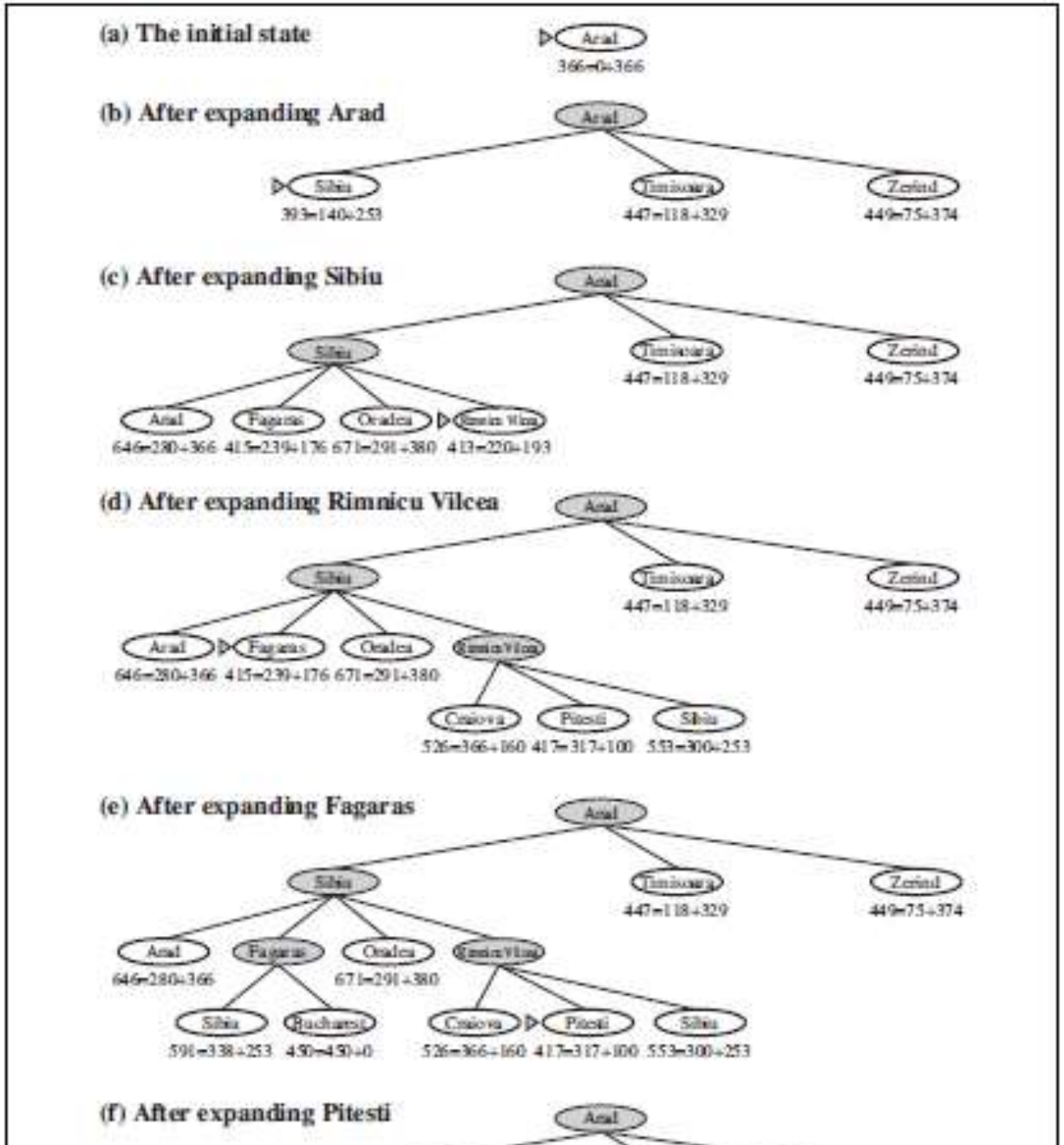
Figure 3.2 A simplified road map of part of Romania.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

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


(a) The initial state
Arad
366
(b) After expanding Arad
Arad
Sibiu
Timisoara
Zerind
253
329
374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
Zerind
329
374
Arad
Fagaras
Oradea
Rimnicu Vilcea
366
176
380
193
(d) After expanding Fagaras
Arad
Sibiu
Timisoara
Zerind
329
374
Arad
Fagaras
Oradea
Rimnicu Vilcea
366
380
193
Sibiu
Bucharest
253
0

- The values of g are computed from the step costs in Figure 3.2, and the values of h_{SLD} are given in Figure 3.22



Conditions for optimality for A*: Admissibility and consistency:

- 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.
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.

A second, slightly stronger condition called **consistency** (or sometimes **monotonicity**) is required only for applications of A∗ to graph search.

A heuristic h(n) is consistent if, for every node n and every successor n 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.

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

- 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.*
- That A∗ search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A∗ is the answer to all our searching needs. The catch is that, for most problems, the number of states within the goal contour search space is still exponential in the length of the solution.

UNIT II: ADVERSARIAL SEARCH

5.1 GAMES:

- **Multiagent** environments**:** in which each agent needs to consider the actions of other agents and how they affect its own welfare.
- **Competitive** environments: in which the agents goals are in conflict, giving rise to **adversarial search** problems—often known as **games.**
- In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, **zero-sum games** of **perfect information** (such as chess).
- For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} or 10^{154} nodes.

A game can be formally defined as a kind of search problem with the following elements:

- **S0:** The **initial state**, which specifies how the game is set up at the start.
- **PLAYER(s):** Defines which player has the move in a state.
- **ACTIONS(s):** Returns the set of legal moves in a state.
- **RESULT(s, a):** The **transition model**, which defines the result of a move.
- **TERMINAL-TEST(s):** A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- **UTILITY(s, p):** A **utility function** (also called an objective function or payoff function). It defines the final numeric value for a game that ends in terminal state s for a player p.

  In chess, the outcome is a win, loss, or draw, with values +1, 0, or 1/2 .

- The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the **nodes** are **game states** and the **edges are moves**.

Ex:

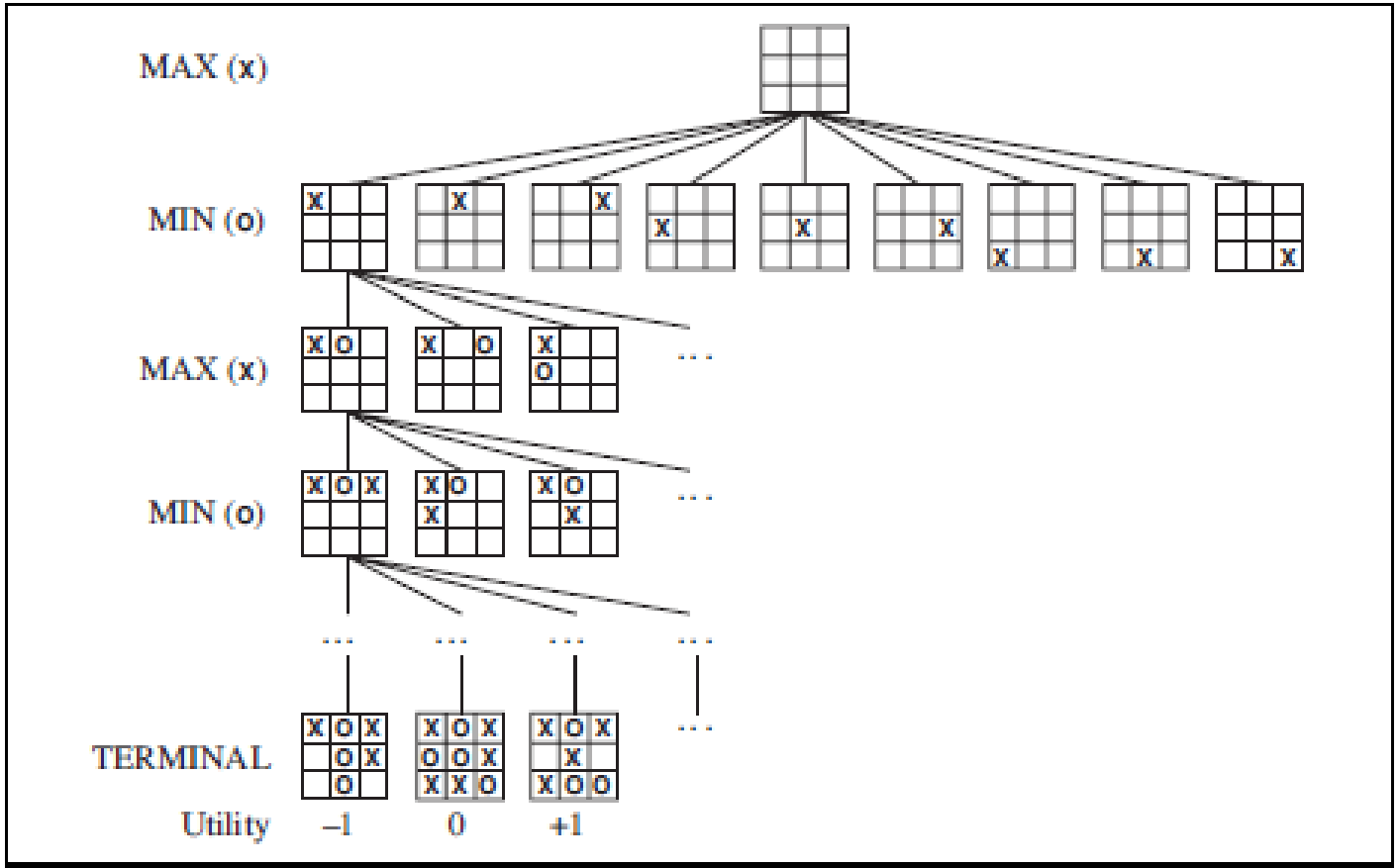
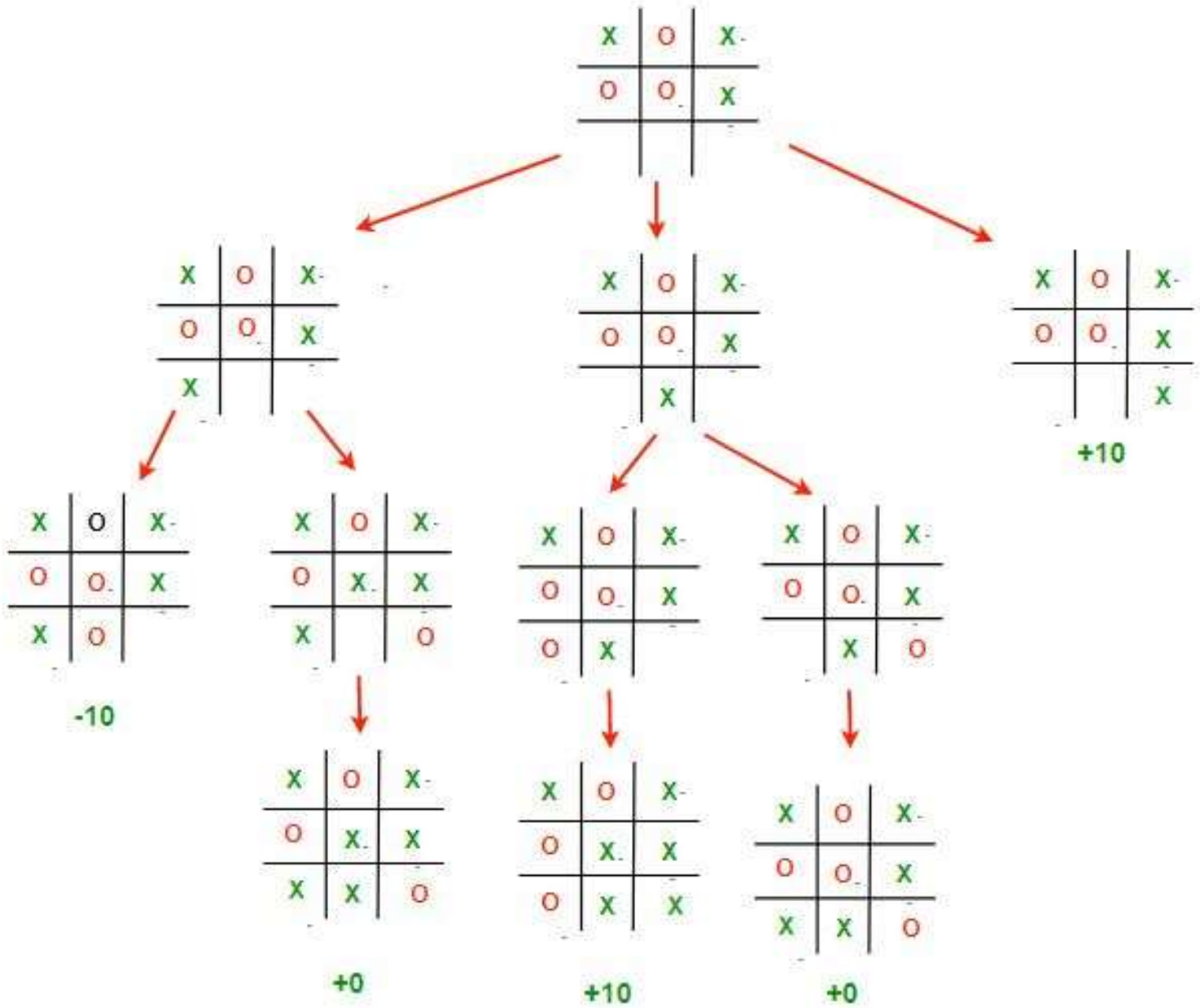


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

For tic-tac-toe the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes.



5.2 OPTIMAL DECISIONS IN GAMES

- The optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win.

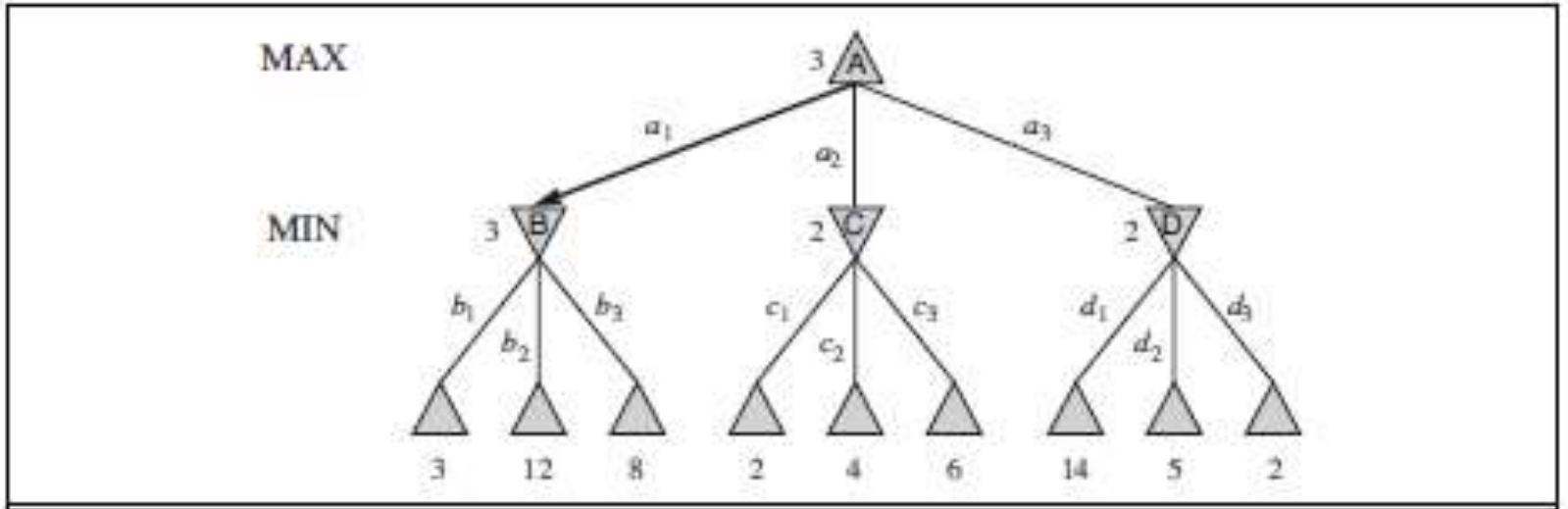


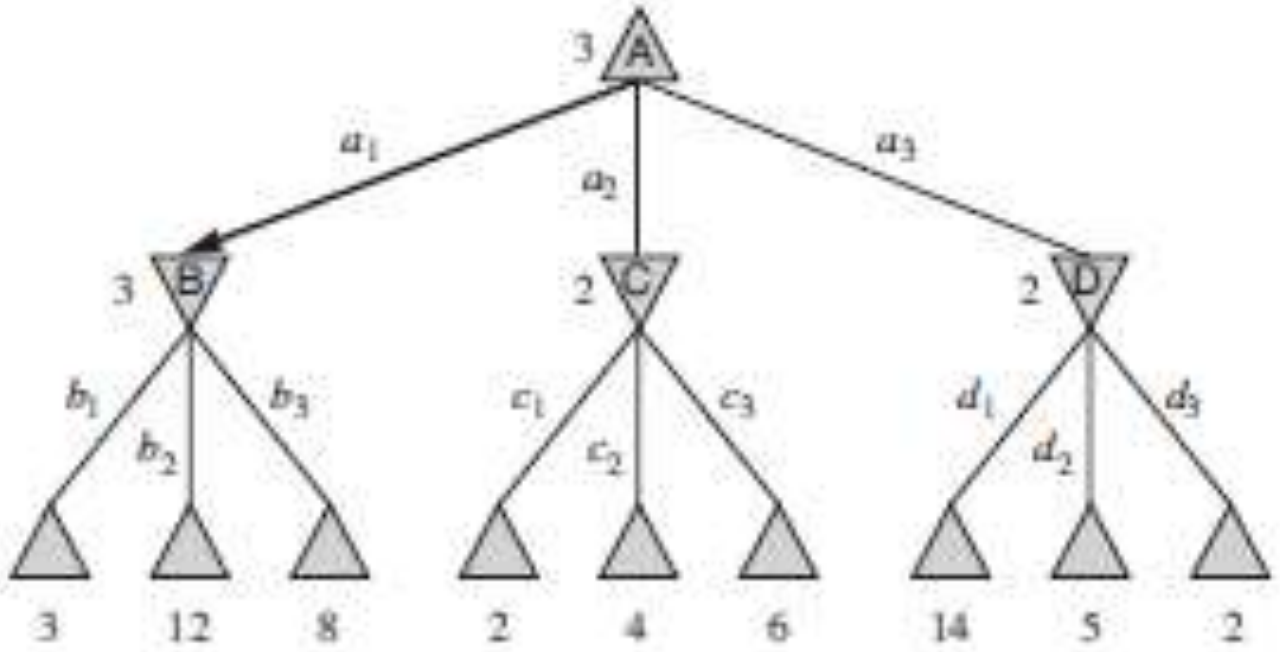
Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

- The possible moves for MAX at the root node are labeled a1, a2, and a3. The possible replies to a1 for MIN are b1, b2,b3, and so on. This particular game ends after one move each by MAX and MIN.

- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as MINIMAX(n).
- The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility.
- Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

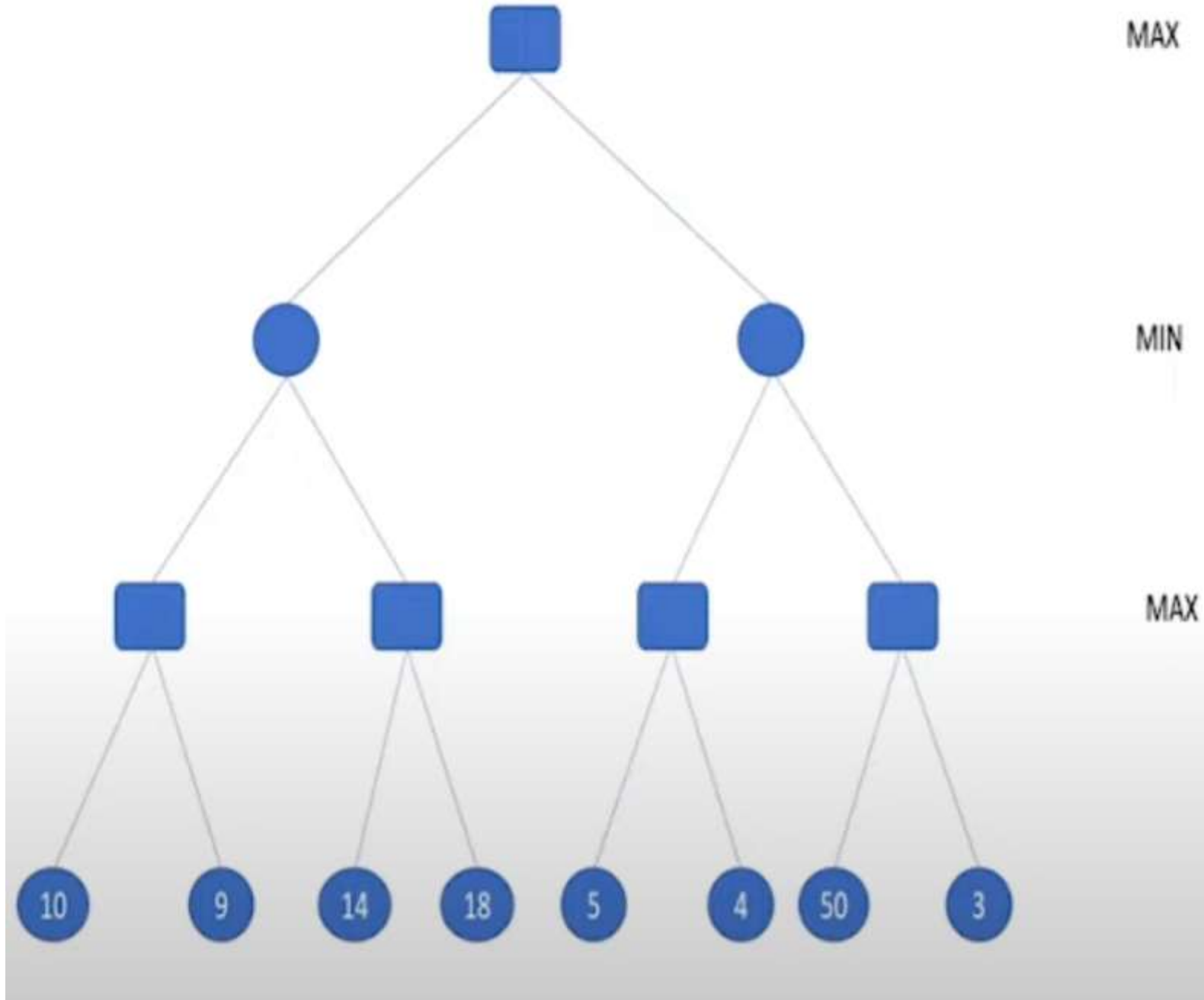
$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

- Let us apply these definitions to the game tree in Figure 5.2.
- The terminal nodes on the bottom level get their utility values from the game's UTILITY function.
- The first MIN node, labeled B, has three successor states with values 3, 12, and 8, so its minimax value is 3 and so on.

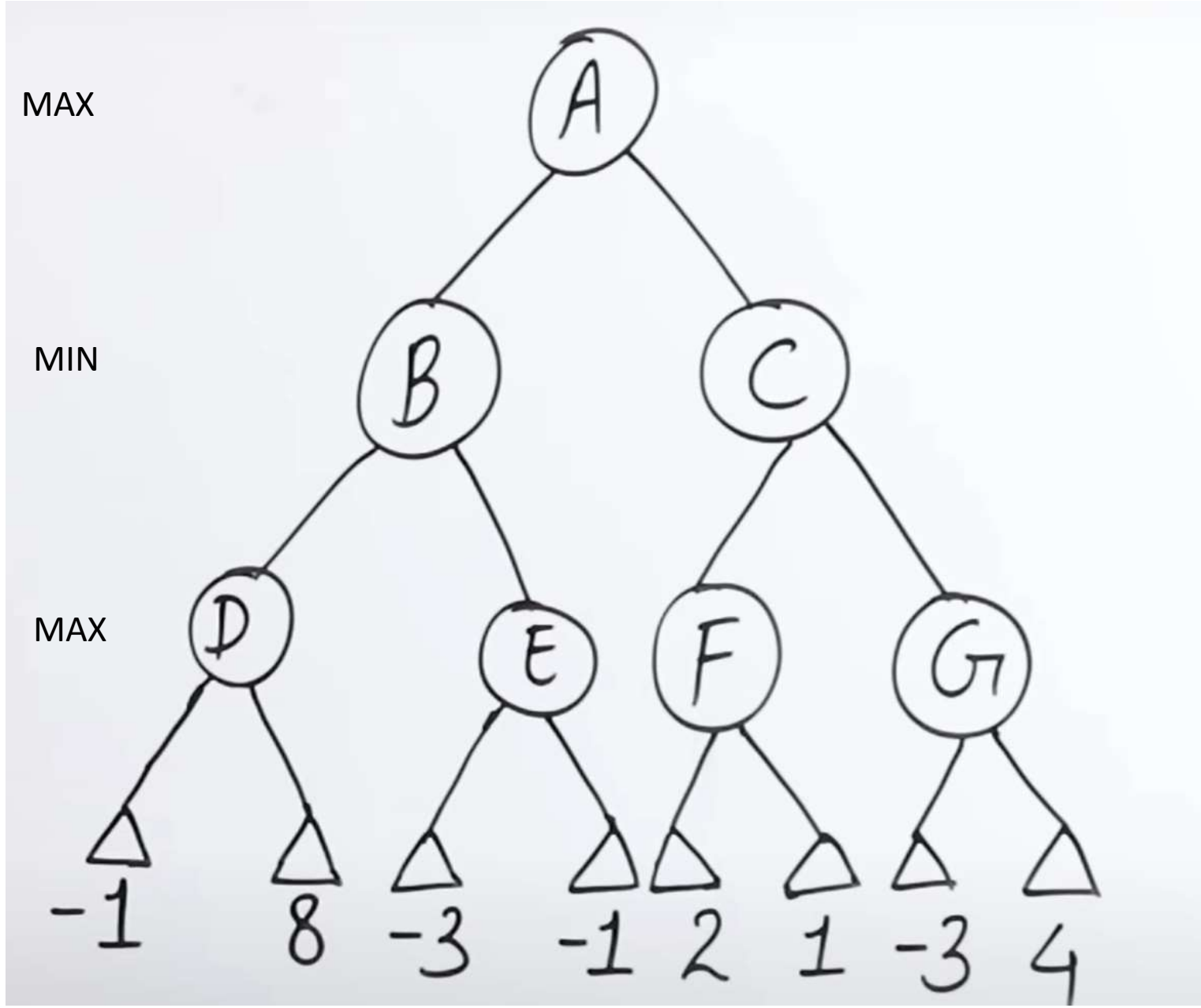


5.2.1 The minimax algorithm

- Consider the following two player game tree in which the static scores are given from the first players point of view.
- Apply **the Mini Max search algorithm** and compute the value of the root of the tree.
- Also find the most convenient path for MAX Node.

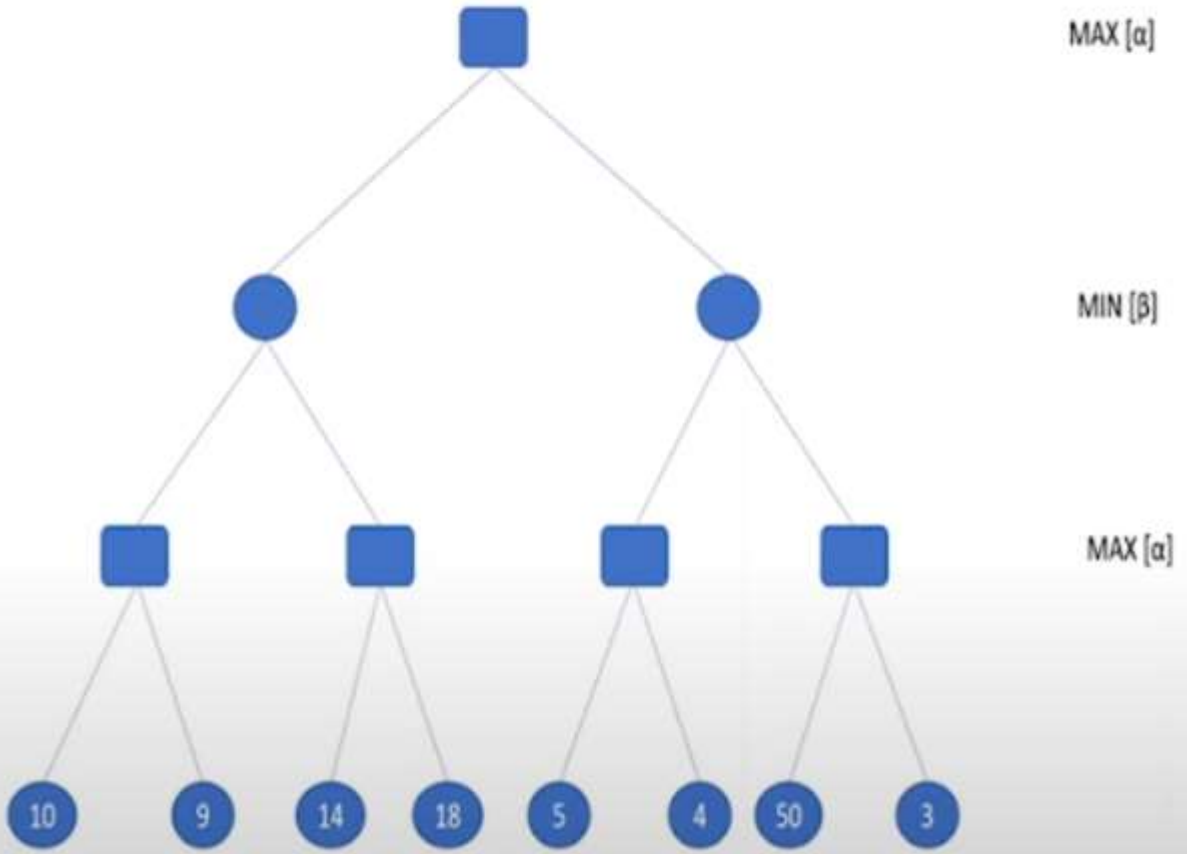

MAX
MIN
MAX
10
9
14
18
5
4
50
3

1. Generate the whole game tree to leaves
2. Apply utility (payoff) function to leaves
3. Use DFS for expanding the tree
4. Back-up values from tree toward the nodes:
    1. Max node computes the maximum value from its child node
    2. Min node computes the minimum value from its child node
5. When value reaches the root: optimal move is determined


MAX
MIN
MAX
A
B
C
D
E
F
G
-1
8
-3
-1
2
1
-3
4

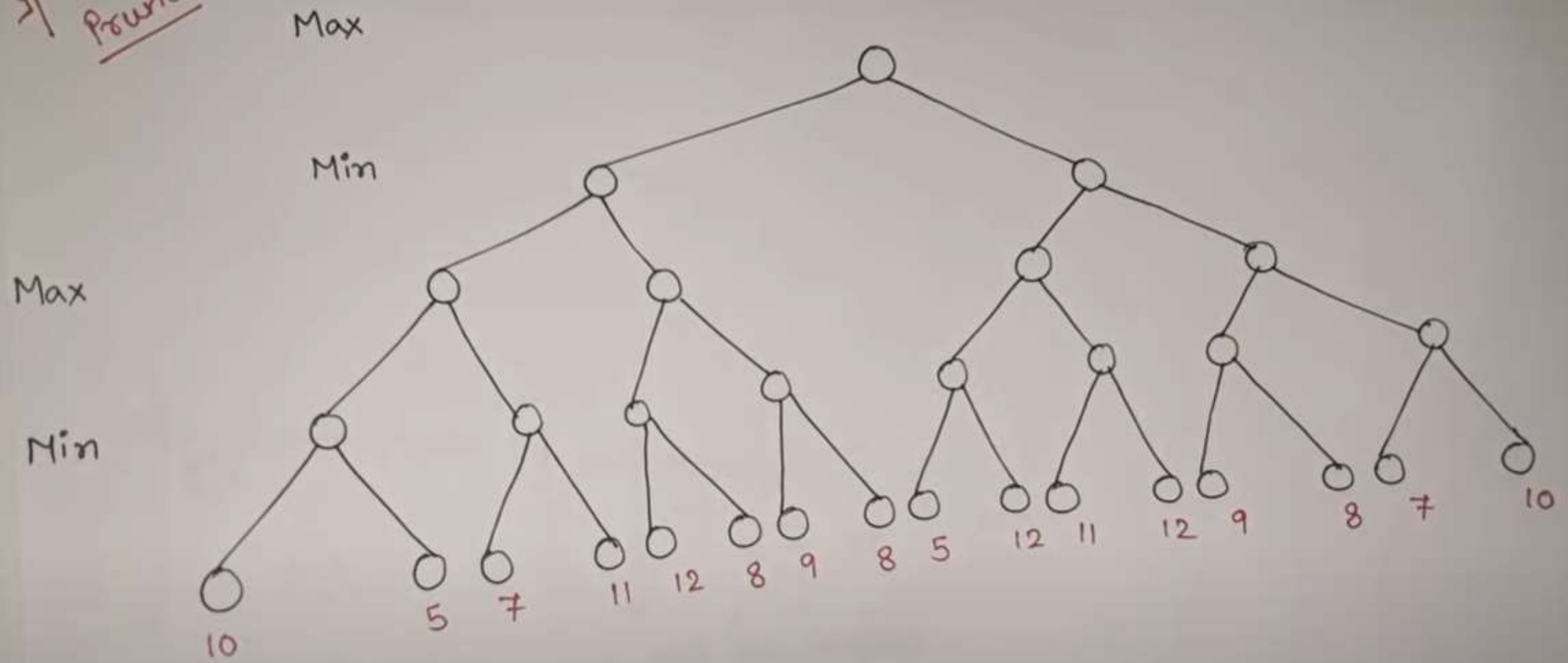
ALPHA–BETA PRUNING:

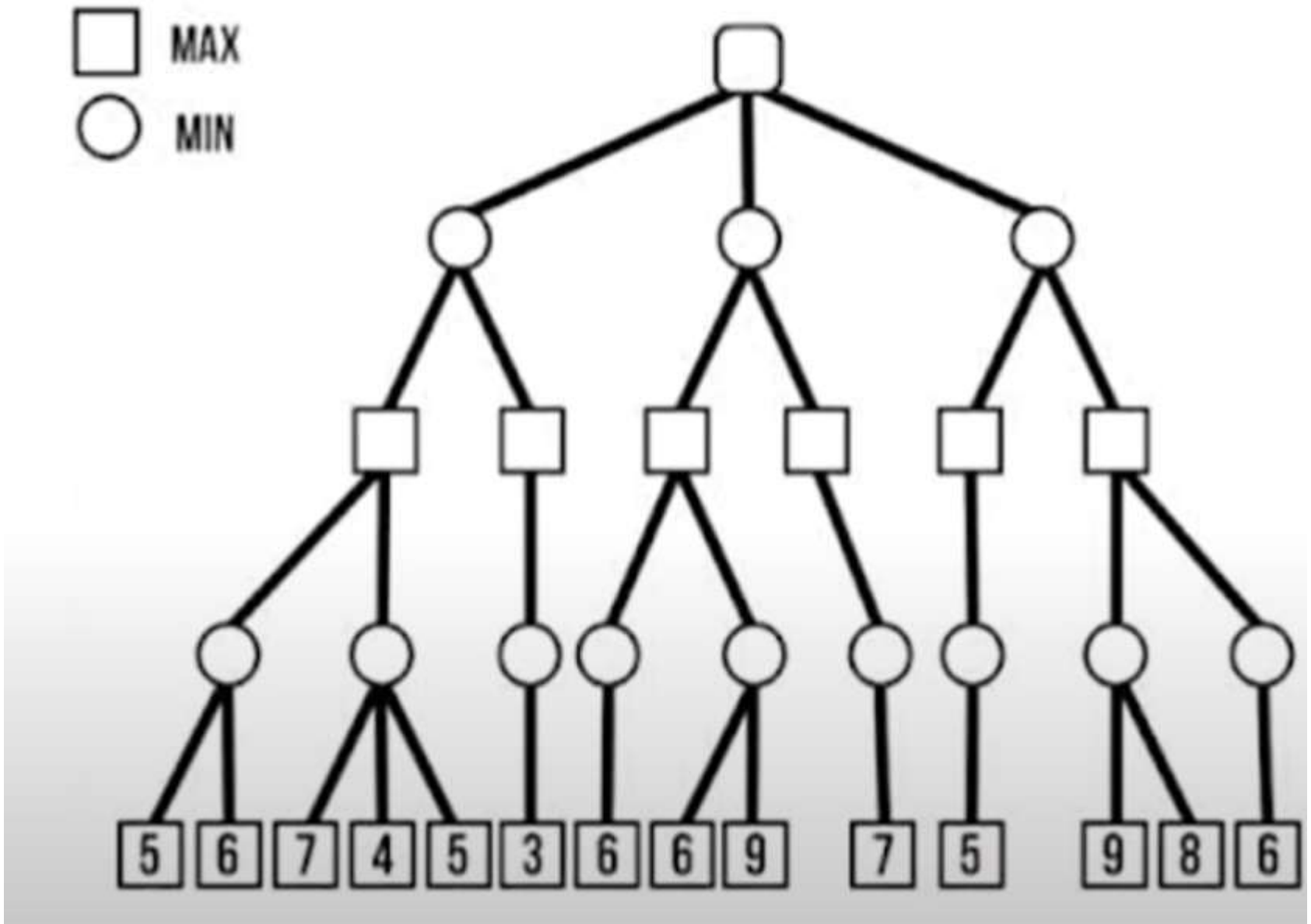
- $\boldsymbol{Alpha(\alpha) - Beta(\beta)}$ proposes to find the optimal path without looking at every node in the game tree.
- **Max contains** $\boldsymbol{Alpha(\alpha)}$ and **Min contains** $\boldsymbol{Beta(\beta)}$ bound during the calculation.
- In both **MIN** and **MAX** node, we return when $\boldsymbol{\alpha \geq \beta}$ which compares with its parent node only.
- Both **minimax and** $\boldsymbol{Alpha(\alpha) - Beta(\beta)}$ cut-off give same path
- $\boldsymbol{Alpha(\alpha) - Beta(\beta)}$ gives **optimal solution** as it takes less time to get the value for the root node.



Alpha Beta Pruning

If $\alpha \geq \beta$, Prune.




MAX
MIN
5
6
7
4
5
3
6
6
9
7
5
9
8
6

5.4 IMPERFECT REAL-TIME DECISIONS

- Disadvantages of minimax and alpha beta pruning: minimax or alpha–beta can be altered in two ways:
- Replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility.
- And replace the terminal test by a **cutoff test** that decides when to apply EVAL. That gives us the following for heuristic minimax for state s and maximum depth d:

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

5.4.1 Evaluation functions:

- An evaluation function returns an *estimate* of the expected utility of the game from a given position, just as the heuristic functions.
- So it should be clear that the performance of a game-playing program depends strongly on the quality of its evaluation function.

How to design good evaluation functions?

- **First**, the evaluation function should order the *terminal* states in the same way as the true utility function: states that are wins must evaluate better than draws, which in turn must be better than losses.
- **Second**, the computation must not take too long! (The whole point is to search faster.)
- **Third**, for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

Logical Agents:

- Humans make strong claims about how the intelligence is achieved—not by purely reflex mechanisms but by processes of **reasoning** that operate on internal **representations** of knowledge.
- In AI, this approach to intelligence is embodied in **knowledge-based agents**.

KNOWLEDGE-BASED AGENTS:

- The central component of a knowledge-based agent is its **knowledge base**, or KB. A knowledge base is a set of **sentences**.
- Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. (like **axiom).**
- The standard names for these operations are **TELL** and **ASK**, respectively. Both operations may involve **inference.**
- **Inference** must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously

- Like all our agents, it takes a **percept as input** and **returns an action**. The agent maintains a knowledge base, KB, which may initially contain some **background knowledge** as given in below fig.

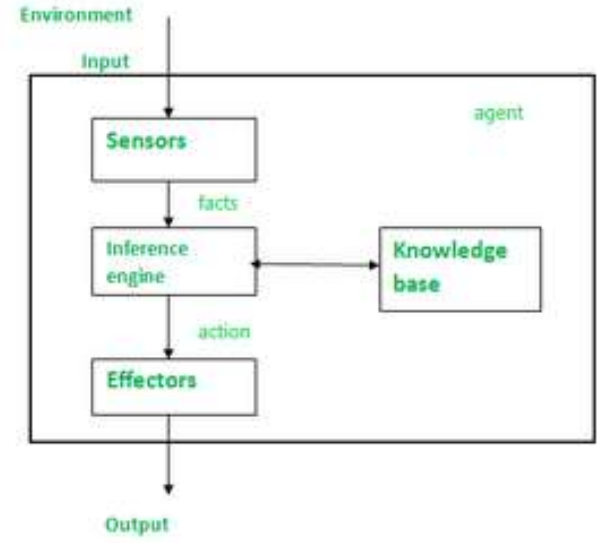
```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

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

- If a percept is given, agent adds it to KB, then it will ask KB for the best action and then tells KB that it has in fact taken that action.

Each time the agent program is called, it does 3 following things:

- 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 on which action was chosen, and the agent executes the action.



Knowledge Based Agents

A Knowledge based system behavior can be designed in following approaches:-

- ***Declarative Approach:*** In this, beginning from an empty knowledge base, the agent can TELL sentences one after another till the agent has knowledge of how to work with its environment. This is known as the declarative approach. It stores required information in empty knowledge-based system.

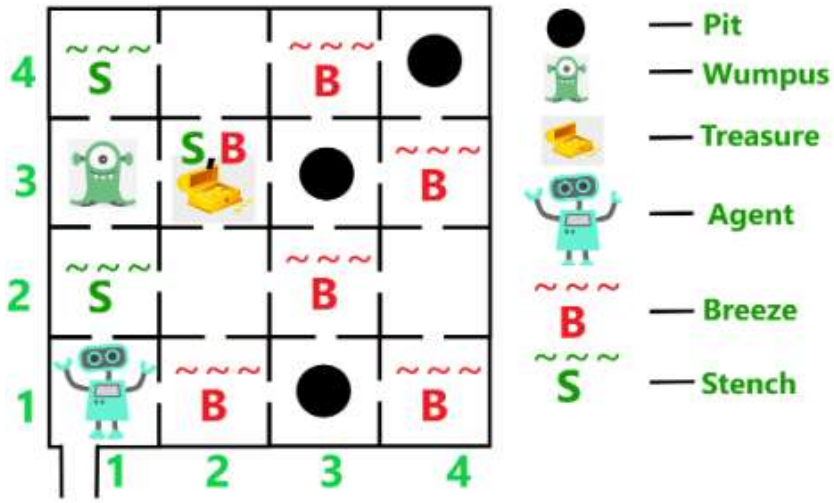
- ***Procedural Approach:*** This converts required behaviors directly into program code in empty knowledge-based system. It is a contrast approach when compared to Declarative approach. In this coding behavior of system is designed .

Characteristics	Declarative Knowledge	Procedural Knowledge
Definition	Knowledge of facts, concepts, and principles	Knowledge of how to do something or perform a task
Type of Knowledge	"Knowing that"	"Knowing how"
Examples	Knowledge of history, geography, and science	Knowledge of cooking, playing a musical instrument, and driving a car
Acquisition Method	Study and memorization	Practice and repetition
Transference	Easily transferred from one situation to another	More difficult to transfer from one situation to another

THE WUMPUS WORLD:

It is an example of a knowledge-based agent that represents Knowledge representation, reasoning and planning. **Knowledge-Based agent** links general knowledge with current percepts to infer hidden characters of current state before selecting actions.

- The **wumpus world** is a cave consisting of rooms connected by passageways. Each room is connected to others through walkways(not diagonally connected).
- The knowledge-based agent starts from *Room[1, 1]*. The cave has – some **pits**, a **treasure** and a beast named **Wumpus**.
- Cave looks as shown:



- The Wumpus cannot move but eats the one who enters its room.
- If the agent enters the pit, it gets stuck there.
- The goal of the agent is to take the treasure and come out of the cave.
- The agent is rewarded, when the goal conditions are met.
- The agent is penalized, when it falls into a pit or being eaten by the Wumpus.
- Some elements support the agent to explore the cave, like -The wumpus's adjacent rooms are stenchy.
- The agent is given one arrow which it can use to kill the wumpus when facing it (Wumpus screams when it is killed).
- The adjacent rooms of the room with pits are filled with breeze.
- The treasure room is always glittery.

PEAS description for wumpus world problem:

1. Performance measures:

- Agent gets the gold and return back safe = *+1000 points*
- Agent dies = *-1000 points*
- Each move of the agent = *-1 point*
- Agent uses the arrow = *-10 points*

2. Environment:

- A cave with *16(4×4)* rooms
- Rooms adjacent (not diagonally) to the Wumpus are stinking
- Rooms adjacent (not diagonally) to the pit are breezy
- The room with the gold glitters
- Agent's initial position – *Room[1, 1]* and facing right side
- Location of Wumpus, gold and *3 pits* can be anywhere, except in *Room[1, 1]*.

3. Actuators:
Devices that allow the agent to perform the following actions in the environment.

- Move forward
- Turn right by 90^0
- Turn left 90^0
- Shoot: The agent dies a miserable death if it enters a square containing a pit or a live wumpus. The agent has only one arrow, so only the first *Shoot* action has any effect.
- Grab The action *Grab* can be used to pick up the gold if it is in the same square as the agent.
- Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].

4. Sensors:

The agent has **five** sensors, each of which gives a single bit of information

- In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a ***Stench***.
- In the squares directly adjacent to a pit, the agent will perceive a ***Breeze*.**
- In the square where the gold is, the agent will perceive a ***Glitter***.
- When an agent walks into a wall, it will perceive a ***Bump***.
- When the wumpus is killed, it emits a woeful ***Scream*** that can be perceived anywhere in the cave.

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

Characterization of Wumpus World:

- **Partially Observable:** knows only the local perceptions
- **Deterministic:** outcome is precisely specified
- **Sequential:** subsequent level of actions performed
- **Static:** Wumpus, pits are immobile
- **Discrete:** discrete environment
- **Single-agent:** The knowledge-based agent is the only agent whereas the wumpus is considered as the environment's feature.

Solution:

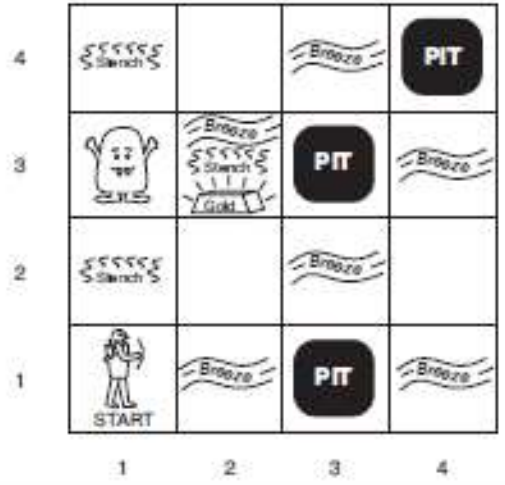


Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing right.

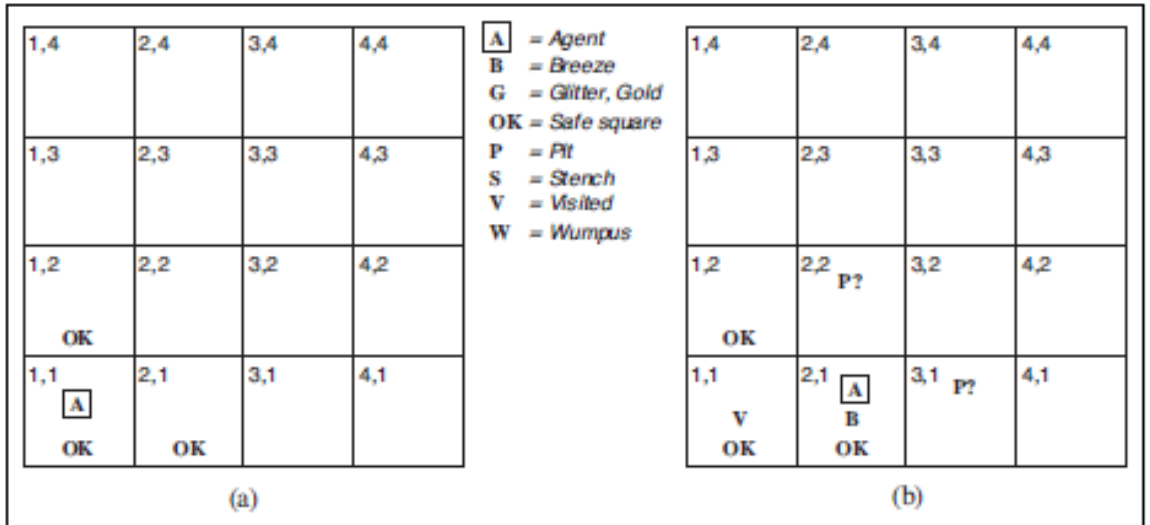


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After one move, with percept [*None*, *Breeze*, *None*, *None*, *None*].

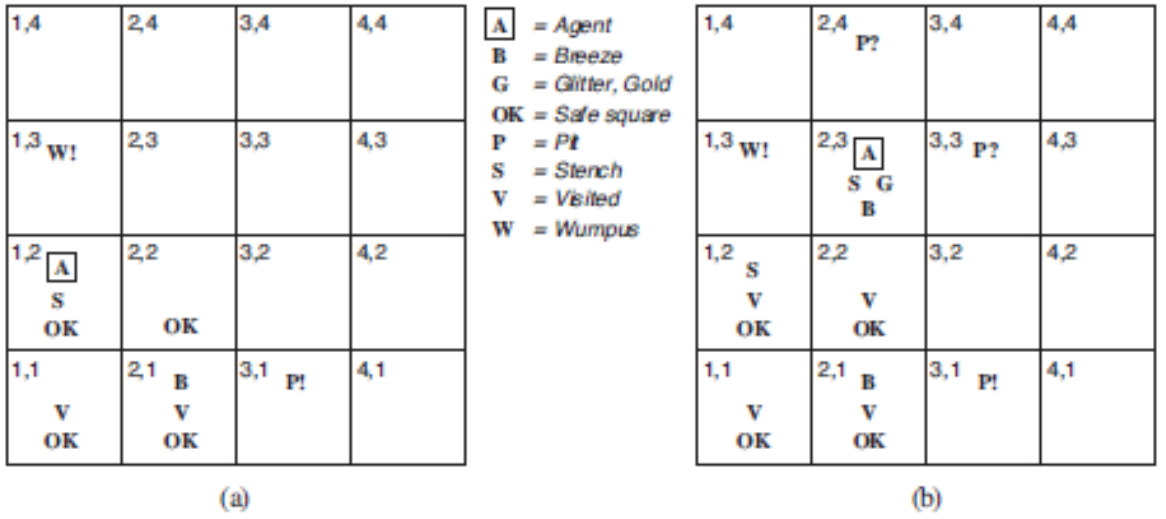


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

- The first percept is [None, None, None, None, None], from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK.
- Let us assume that the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. At this point, there is only one known square that is OK and that has not yet been visited, the agent will turn around, go back to [1,1], and then proceed to [1,2].
- The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3].
- This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

7.3 LOGIC:

- The Knowledge based sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed.
- A logic must also define the **semantics** or meaning of sentences.
- The semantics defines the **truth** of each sentence with respect to each **possible world**. For example, the semantics for arithmetic specifies that the sentence "x + y =4" is true in a world where x is 2 and y is 2.
- When we need to be precise, we use the term **model** in place of "possible world".
- If a sentence α is true in model m, we say that m **satisfies** α or sometimes m **is a model of** α. We use the notation M(α) to mean the set of all models of α.
- In mathematical notation, we write α |= β, meaning α entails β.

7.4 PROPOSITIONAL LOGIC: A VERY SIMPLE AND POWERFUL LOGIC

Proposition is a sentence/ statement
Logic: Reasoning (True/ False/not both)

1. Syntax:

- The **syntax** of propositional logic defines the **allowable sentences**.
- The **atomic sentences** consist of a **single proposition symbol**. Each such symbol stands for a proposition that can be true or false.
- We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, $W_{1,3}$ and North.
- **Complex sentences** are constructed from simpler sentences, using parentheses and **logical connectives**.

There are five connectives in common use in case of complex sentences:

- **¬(not):** A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).
- **∧ (and):** A sentence whose main connective is ∧, such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction**; its parts are the **conjuncts**. (The ∧ looks like an "A" for "And.")
- **∨ (or):** A sentence using ∨, such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction** of the **disjuncts** $(W_{1,3} \wedge P_{3,1})$ and $W_{2,2}$.

- **⇒ (implies):** A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is (W1,3 ∧P3,1), and its **conclusion** or **consequent** is $\neg W_{2,2}$.
  - Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as ⊃ or →.
- **⇔ (if and only if):** The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**. Some other books write this as ≡.

- To compute the truth of sentences :
  - True is true in every model.
  - False is false in every model.
- For complex sentences, we have five rules, which hold for any subsentences P and Q in any model m :
  - ¬P is true iff P is false in m.
  - P ∧ Q is true iff both P and Q are true in m.
  - P ∨ Q is true iff either P or Q is true in m.
  - P ⇒ Q is true unless P is true and Q is false in m.
  - P ⇔ Q is true iff P and Q are both true or both false in m.

Truth Table for complex sentence:

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*.

1. Make a truth table for the statement ¬PVQ.
2. Are the statements, “it will not rain or snow” and “it will not rain and it will not snow” logically equivalent?
3. Prove that the statements ¬(P→Q)¬(P→Q) and P∧¬QP∧¬Q are logically equivalent without using truth tables.
4. Are the statements (PVQ)→R(PVQ)→R and (P→R)V(Q→R)(P→R)V(Q→R) logicall y equivalent?

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

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

7.6 EFFECTIVE PROPOSITIONAL MODEL CHECKING

2 efficient algorithms for general propositional inference based on model checking:

1. Backtracking search.

2. Local hill-climbing search.

Backtracking algorithm: Davis–Putnam algorithm or DPLL algorithm.

It is essentially a recursive, depth-first enumeration of possible models. It embodies three improvements over the simple scheme of TT-ENTAILS?

- ***Early termination***: The algorithm detects whether the sentence must be true or false, even with a partially completed model.

For ex(AvB) ∧(AvC) – is true if A is true regardless of B and C

***Pure symbol heuristic*: pure symbol** is a symbol that always appears with the same "sign" in all clauses.

For example, in the three clauses (A ∨ ¬B), (¬B ∨ ¬C), and (C ∨ A), the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure.

***Unit clause heuristic*:** A **unit clause** was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model.

- **Ex:** if the model contains B =true, then (¬B ∨ ¬C) simplifies to ¬C, which is a unit clause. Obviously, for this clause to be true, C must be set to false. The unit clause heuristic assigns all such symbols. Notice also that assigning one unit clause can create another unit clause.

Local search algorithms: WALKSAT

- On every iteration, the algorithm picks an **unsatisfied clause** and picks a symbol in the clause to flip.
- It chooses randomly between two ways to pick which symbol to flip:

(1) a “min-conflicts” step that minimizes the number of unsatisfied clauses in the new state and

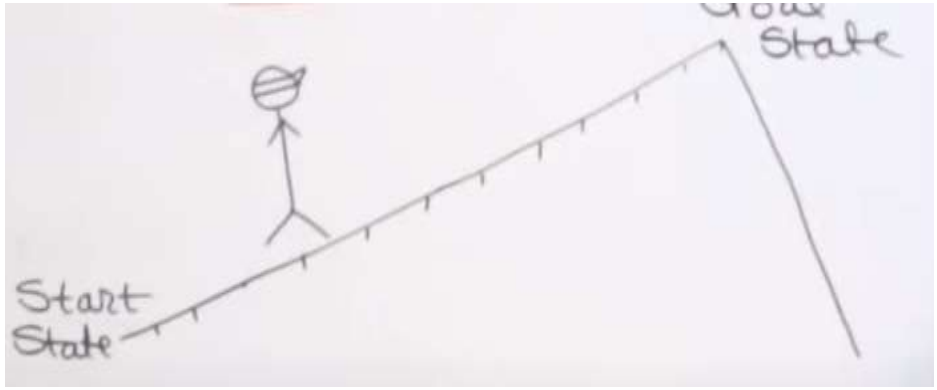
(2) a “random walk” step that picks the symbol randomly.

- When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns failure, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time.

Limitations:

- WALKSAT is most useful when we expect a solution to exist
- WALKSAT cannot always detect *unsatisfiability*, which is required for deciding entailment

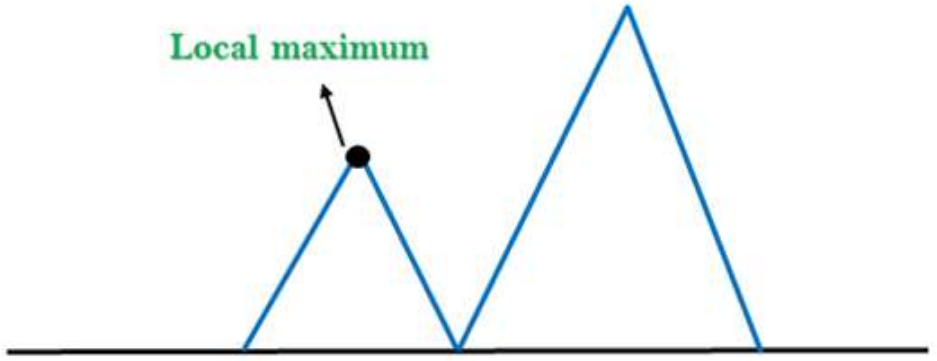
Ex for local search: Hill Climbing algorithm-



- Solve problems with multiple solutions
- Suppose a person is blind folded and left on a hill

and asked to go forward or backward to ultimately

reach the goal state.



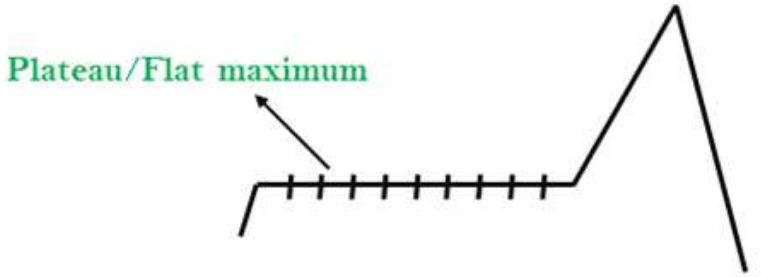
- Next step is better than current step.
- Limitation:
  - Local maxima: In this, after a point we have to get down the hill to reach the top i.e next step is not better than the current step. Hence will be stuck at

  one point.

  Solution: Backtrack few nodes and try a different path

- Plateau: After a point there is a flat surface i.e next step will be same as next step, hence will be stuck at one point

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem.



- Ridge: Cannot reach the goal state in one move
- Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.

First Order Logic: FOL is much more expressive language than the propositional logic. It is an extension of PL. Also called as predicate logic.

- PL is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation.
- Drawback of PL is it 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 $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.

FOL is known as the powerful language which is used to develop information related to objects in a very easy way.

About objects, relations and properties:

- **Objects**: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries . . .
- **Relations:** these can be unary relations or **properties** such as red, round, bogus, prime, multistoried . . ., or more general n-ary relations such as brother of, bigger than, inside,part of, has color, occurred after, owns, comes between, . . .
- **Functions**: father of, best friend, third inning of, one more than, beginning of . . .

Ex: Evil King John ruled England in 1200.”
Objects:
Relations:
Properties:

SYNTAX:

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

Quantifiers in FOL:

- It express the properties of entire collections of objects, instead of enumerating the objects by name.
- First-order logic contains two standard quantifiers, called *universal* and e*xistential*.

Universal quantification (∀):

"Squares neighboring the wumpus are smelly" and "All kings are persons".

The second rule, "All kings are persons," is written in first-order logic as

$$\forall x \; King(x) \Rightarrow Person(x) .$$

"For all x, if x is a king, then x is a person." The symbol x is called a **variable**.

A term with no variables is called a **ground term**.

Consider the model shown in Figure 8.2 and the intended interpretation that goes with it. We can extend the interpretation in 5 ways:

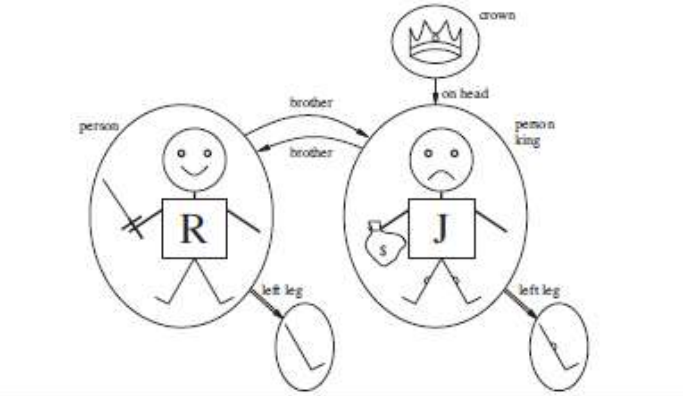


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

x → Richard the Lionheart,

x → King John,

x → Richard's left leg,

x → John's left leg,

x → the crown.

- The universally quantified sentence ∀ x King(x) ⇒ Person(x) is true in the original model if the sentence King(x) ⇒ Person(x) is true under each of the five extended interpretations.
- Richard the Lionheart is a king ⇒ Richard the Lionheart is a person.
- King John is a king ⇒ King John is a person.
- Richard's left leg is a king ⇒ Richard's left leg is a person.
- John's left leg is a king ⇒ John's left leg is a person.
- The crown is a king ⇒ the crown is a person.

Existential quantification (∃): we can make a statement about *some* object in the universe without naming it.

Ex: King John has a crown on his head, can be written as

$$\exists\ x\ \text{Crown}(x) \wedge \text{OnHead}(x, \text{John})$$

∃x is pronounced “There exists an x such that . . .” or “For some x . . .”.

Intuitively, the sentence ∃x P says that P is true for at least one object x.

Wumpus world representation using FOL:

- The wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred;

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

Here, Percept is a binary predicate, and Stench and so on are constants placed in a list.

- The actions in the wumpus world can be represented by logical terms:

  Turn(Right), Turn(Left), Forward , Shoot , Grab, Climb .

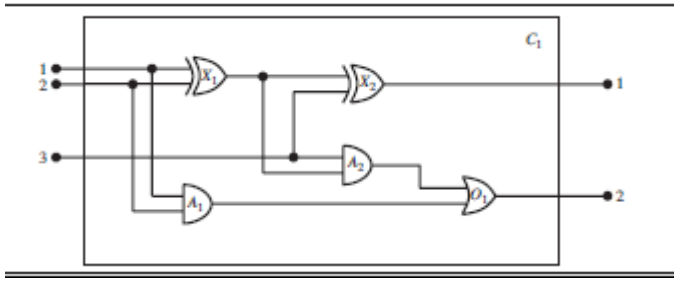
- To determine which is best, the agent program executes the query

  ASK V ARS(∃ a BestAction(a, 5)) ,

which returns a binding list such as {a/Grab}. The agent program can then return Grab as the action to take.

The knowledge-engineering process: Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

- *Identify the task.*
- *Assemble the relevant knowledge*
- *Decide on a vocabulary of predicates, functions, and constants.*
- *Encode general knowledge about the domain.*
- *Encode a description of the specific problem instance.*
- *Pose queries to the inference procedure and get answers.*
- *Debug the knowledge base*



Ex: **The electronic circuits domain**

- **Identify the task:** does the circuit in Figure 8.6 actually add properly? If all the inputs are high, what is the output of gate A2?

- **Assemble the relevant knowledge:** What do we know about digital circuits?
- **Decide on a vocabulary:** The next step is to choose functions, predicates, and constants to represent them.

First, we need to be able to distinguish gates from each other and from other objects. Each gate is represented as an object named by a constant, about which we assert that it is a gate with, say, Gate(X1). Identify predicate Terminal (x).

- **Encode general knowledge of the domain:** One sign that we have a good ontology is that we require only a few general rules, which can be stated clearly and concisely. These are all the axioms we will need:

1. If two terminals are connected, then they have the same signal:

$\forall$ t1, t2 Terminal (t1) $\wedge$ Terminal (t2) $\wedge$ Connected(t1, t2) $\Rightarrow$Signal (t1)=Signal (t2)

Encode the specific problem instance:

Ex: Gate(X1) $\wedge$ Type(X1)=XOR

- **Pose queries to the inference procedure:** What combinations of inputs would cause the first output of C1 to be 0 and the second output of C1 to be 1?
- **Debug the knowledge base:** We can perturb the knowledge base in various ways to see what kinds of erroneous behaviors emerge.

Reduction to propositional inference: For example, suppose our knowledge base contains just the sentences

$\forall$ x King(x) $\wedge$ Greedy(x) $\Rightarrow$ Evil(x)

King(John)

Greedy(John)

Brother (Richard, John) .

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base—in this case, {x/John} and {x/Richard }. We obtain

- King(John) $\wedge$ Greedy(John) $\Rightarrow$ Evil(John)
- King(Richard) $\wedge$ Greedy(Richard) $\Rightarrow$ Evil(Richard)

Now, the knowledge base is essentially propositional if we view the ground atomic sentences—King(John), Greedy(John), and so on—as proposition symbols.