Artificial Intelligence Unit-1&Unit-2

Theory syllabus		
Unit	Content	Hrs
1	**Introduction:** Concept of AI, history, current status, Pattern Recognition, Deep Learning, Robotics, Vision Learning, scope, agents, environments, Problem Formulations, Review of tree and graph structures, State space representation, Search graph and Search tree.	04
2	**Search Algorithms:** Random search, Search with closed and open list, Depth first and Breadth first search, Heuristic search, Best first search, A* algorithm, Game Playing- MiniMax, Alpha-Beta Cut-off	07
3	**Knowledge Representation:** Building a Knowledge Base, Logic based representations, Propositional Logic (PL), First Order Logic (FOL), Resolution and Reasoning in FOL, Inference Engine, Rule Based Expert System	03
4	**Probabilistic Reasoning:** Fuzzy Logic, Probability, Bayes Rule, Bayesian Networks- representation, construction and inference, temporal model, hidden Markov model.	05
5	**Markov Decision process:** MDP formulation, utility theory, utility functions, value iteration, policy iteration and partially observable MDPs	09
6	**Deep Learning: Basics of Neural Network:** Biological Neural Network, Neural Network Representation, Neural Networks as a Paradigm for Parallel Processing, The Perceptron, Training a Perceptron, Learning Boolean Functions, Multilayer Perceptron	05
7	**Genetic Algorithm:** Traditional Methods of Optimization, Genetic Operators, Binary Coded Genetic Algorithm, Real Coded Genetic Algorithm	04
8	**Reinforcement Learning:** Passive reinforcement learning, direct utility estimation, adaptive dynamic programming, temporal difference learning, active reinforcement learning- Q learning.	08

Text Books	
1	Artificial Intelligence, By Rich E. & Kevin Knight, Tata McGraw Hill.
2	Stuart Russell and Peter Norvig, "Artificial Intelligence: A Modern Approach", Prentice Hall
Reference Books	
1	Genetic Algorithms in Search, Optimization, and Machine Learning, D. E. Goldberg, Addison-Wesley.
2	Neural Networks: A Comprehensive Foundation, S. Haykin, PHI
3	Machine Learning, By Tom M. Mitchell, Tata McGraw-Hill.
4	Trivedi, M.C., "A Classical Approach to Artifical Intelligence", Khanna Publishing House, Delhi
5	Saroj Kaushik, "Artificial Intelligence", Cengage Learning India.
6	David Poole and Alan Mackworth, "Artificial Intelligence: Foundations for Computational Agents", Cambridge University Press.
7	Dan W. Patterson, "Artificial Intelligence and Expert Systems", Prentice Hall of India
8	Marsland, Stephen. Machine learning: an algorithmic perspective. Chapman and Hall/CRC.
9	Raschka, Sebastian. Python machine learning. Packt Publishing Ltd.
ICT/MOOCs Reference	
1	https://nptel.ac.in/courses/106105079/
2	https://nptel.ac.in/courses/106105077
3	https://nptel.ac.in/courses/106106126
4	https://nptel.ac.in/courses/106106140/
5	https://nptel.ac.in/courses/106/106/106106202/
6	https://ai.berkeley%2Cedu/project_overview.html

UNIT 1: Introduction

- Concept of AI, history, current status, Pattern Recognition, Deep Learning, Robotics, Vision Learning, scope,
- Agents, environments,
- Problem Formulations, Review of tree and graph structures, State space representation, Search graph and Search tree.

INTELLIGENCE VS AI

	Intelligence	Artificial Intelligence
1.	Natural	Programmed by human beings
2.	Increases with experience and also hereditary.	Nothing called hereditary but systems do learn from experience.
3.	Highly refined and no electricity from outside is required to generate output. Rather knowledge is good for intelligence.	It is in computer system and we need electrical energy to get output. Knowledge base is required to generate output.
4.	No one is an expert. We can always get better solution from another human being.	Expert systems are made which have the capability of many individual person's experiences and ideas.
5.	Intelligence increases by supervi- sed or unsupervised teaching.	We can increase AI's capabilities by other means apart from supervised and unsupervised teaching.

WHAT IS AI? DEFINITION

- **Universally accepted definition:**
- Artificial Intelligence (AI) is the study of how to make computers do things which, at the moment, people do better.

WHAT IS AI? DEFINITION

- **According to Patterson**, "AI is a branch of computer science that deals with the study and the creation of computer systems that exhibit some form of intelligence.
- Intelligence means that

a. Systems that learn new concepts and tasks

b. System that can reason and draw useful conclusions about the world around us.

c. Systems that can understand a natural language or perceive and comprehend a visual scene,

d. And systems that perform other types of feats that require human types of intelligence.

WHAT IS AI? DEFINITION

- **General definitions:** "An understanding of AI requires an understanding of related terms such as intelligence, knowledge, reasoning, thought, cognition, learning and solving problems."
- **By Advert:** "AI is the part of computer science concerned with designing intelligent computer systems i.e. system that exhibit characteristics that we associate with intelligent in human behavior.
- **By heuristic:** "AI is the branch of computer science that deals with the ways of representing knowledge using symbols rather than numbers and with the rules of thumb for processing information."
- **Modern definition:** "AI is defined as the branch of computer science dealing with symbolic and non-algorithmic method of problem solving."

WHAT IS AI? DEFINITION

Systems that think like humans.	Systems that think/raationally.
Systems that act like humans.	Systems that act rationally.

Artificial Intelligence

Foundations of AI

- Many older disciplines contribute to a foundation for artificial intelligence.
  - Philosophy; logic, philosophy of mind, philosophy of science, philosophy of mathematics
  - Mathematics; logic, probability theory, theory ofcomputability
  - Psychology; behaviorism, cognitive psychology
  - Computer Science & Engineering; hardware, algorithms, computational complexity theory
  - Linguistics; theory of grammar, syntax, semantics

Types of Artificial Intelligence

- AI algorithms that have been trained with large amounts of data may make intelligent judgments.
- In a more generalized way, the vast area of AI has been divided into two categories:
  - Weak AI
  - Strong AI

Types of Artificial Intelligence

- **Weak AI**
- Narrow AI is another name for weak AI. It's an artificial intelligence system that's been created and taught to do a certain task.
- Siri and Alexa, for example, are AI that isn't very good.
- Unsupervised programming is used to categorize the information.
- Because they already have pre-programmed answers, they categorize things accordingly.
- So, when you ask Alexa to play a song, pay attention.
- The algorithm will reply by playing a song, but it is only doing so because it has been programmed to do so.

Types of Artificial Intelligence

- **Strong AI**
- Strong AI, often known as artificial general intelligence, is more like the human brain.
- It possesses cognitive abilities that aid in the execution of new tasks and directives.
- It can solve a problem without relying on a pre-programmed algorithm.
- Strong AI may be shown in visual perception, speech recognition, decision-making, and language translation.

Artificial Intelligence - History

- **Maturation of AI (1943 – 1952)**
- **Year 1943:** The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of artificial neurons.
- **Year 1949:** Donald Hebb demonstrated an updating rule for modifying the connection strength between neurons. His rule is now called Hebbian learning.

Artificial Intelligence - History

- **Maturation of AI (1943 – 1952)**
- **Year 1950:** The Alan Turing who was an English mathematician and pioneered Machine learning in 1950.
- Alan Turing publishes "Computing Machinery and Intelligence" in which he proposed a test.
- The test can check the machine's ability to exhibit intelligent behavior equivalent to human intelligence, called a **Turing test**.
- Strong AI, often known as artificial general intelligence, is more like the human brain.
- It possesses cognitive abilities that aid in the execution of new tasks and directives.
- It can solve a problem without relying on a pre-programmed algorithm.
- Strong AI may be shown in visual perception, speech recognition, decision-making, and language translation.

Artificial Intelligence - History

- **Birth of AI (1952 – 1956)**
- **Year 1955:** An Allen Newell and Herbert A. Simon created the "first artificial intelligence program“ Which was named as "Logic Theorist". This program had proved 38 of 52 Mathematics theorems, and find new and more elegant proofs for some theorems.

- **Year 1956:** The word "Artificial Intelligence" first adopted by American Computer scientist John McCarthy at the Dartmouth Conference. For the first time, AI coined as an academic field.

Artificial Intelligence - History

- **Golden years of AI (1956 – 1974)**
- **Year 1966:** The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.

- **Year 1972:** The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

Artificial Intelligence - History

- **Boom of AI (1980 – 1987)**
- **Year 1980:** After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert.

- In the Year 1980, the first national conference of the American Association of Artificial Intelligence was held at Stanford University.

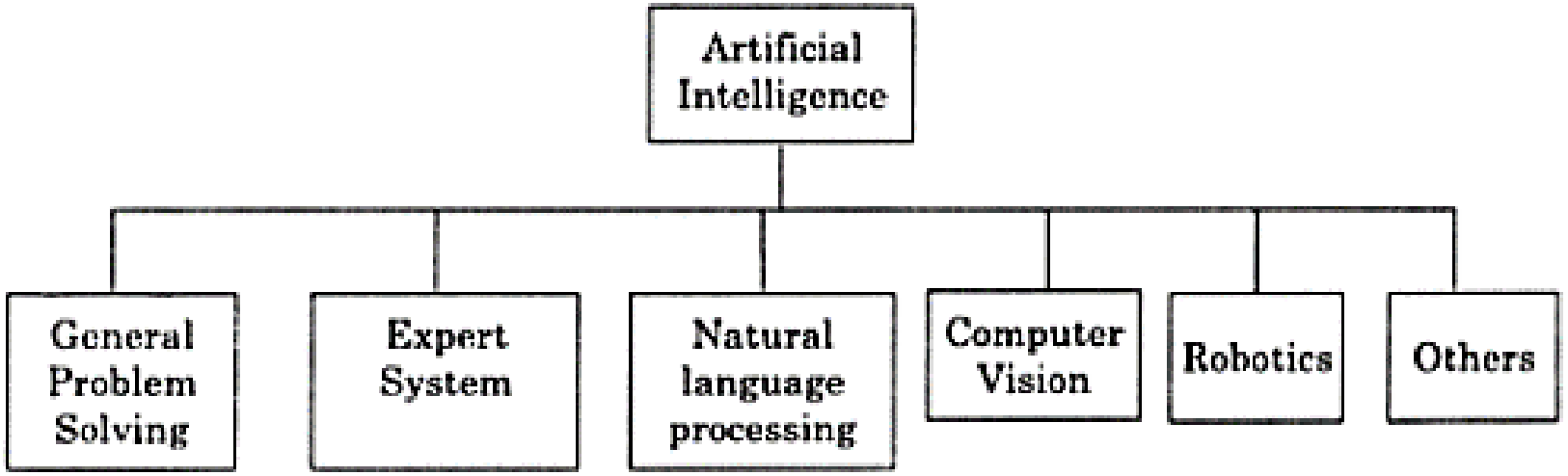
Artificial Intelligence - History

- **Emergence of Intelligent Agents (1993 – 2011)**
- **Year 1997:** In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.
- **Year 2002:** for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.
- **Year 2006:** AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

Artificial Intelligence - History

- **Deep Learning, big data, and AI (2011 to present)**
- **Year 2011:** In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.
- **Year 2012:** Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.
- **Year 2014:** In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."
- **Year 2018:** The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.

Artificial Intelligence



AI – Research Areas

- **Natural Language Processing:** To Understand the natural language and communicate successfully in any language like English
- **Knowledge Processing:** To store what it knows or hears and apply knowledge to solve problem
- **Automated reasoning:** To use the stored information to answer questions. It must be able to reason out and draw new conclusions.
- **Machine Learning:** To adapt to the new changes and detect the patterns or Programs that learn from experience
- **Computer Vision:** To detect and perceive objects
- **Robotics:** To manipulate objects and move about

AI – Research Areas

- **Pattern recognition:** When a program makes observations of some kind, it is often programmed to compare what it sees with a pattern. For example, a vision program may try to match a pattern of eyes and a nose in a scene in order to find a face. More complex patterns, e.g., in a natural language text, in a chess position. or in the history of some event are also studied. These more complex patterns require quite different methods than do the simple patterns.
- **Speech Recognition:** Conversion of speech into text.

AI – Research Areas

- **Expert Systems**
  - It is an integration of software, machine, and special information to provide reasoning and advice
  - Examples – Flight-tracking systems, Clinical systems.
- **Computer Vision**
  - To understand the visual automatically by watching it
- **Natural Language Processing**
  - Examples: Google Now feature, speech recognition, Automatic voice output.

AI – Research Areas

- **Robotics**
  - Examples – Industrial robots for moving, spraying, painting, precision checking, drilling, cleaning, coating, carving, etc.
- **Fuzzy Logic Systems**
  - Examples – Consumer electronics, automobiles, etc.

AI Applications

- **AI in E-Commerce**
  - **Personalized Shopping**

    Artificial Intelligence technology is used to create recommendation engines through which you can engage better with your customers. These recommendations are made in accordance with their browsing history, preference, and interests.
  - **AI-powered Assistants**

    Virtual shopping assistants and chatbots help improve the user experience while shopping online.
  - **Fraud Prevention**

    Credit card frauds and fake reviews are two of the most significant issues that E-Commerce companies deal with. By considering the usage patterns, AI can help reduce the possibility of credit card frauds taking place.

AI Applications

- **<u>AI in Robotics</u>**
  - Robotics is another field where artificial intelligence applications are commonly used. Robots powered by AI use real-time updates to sense obstacles in its path and pre-plan its journey instantly.
  - **It can be used for –**
    - Carrying goods in hospitals, factories, and warehouses
    - Cleaning offices and large equipment
    - Inventory management

AI Applications

- **AI in HealthCare**
  - Artificial Intelligence finds diverse applications in the healthcare sector. AI applications are used in healthcare to build sophisticated machines that can detect diseases and identify cancer cells.
  - Artificial Intelligence can help analyze chronic conditions with lab and other medical data to ensure early diagnosis.
  - AI uses the combination of historical data and medical intelligence for the discovery of new drugs.

AI Applications

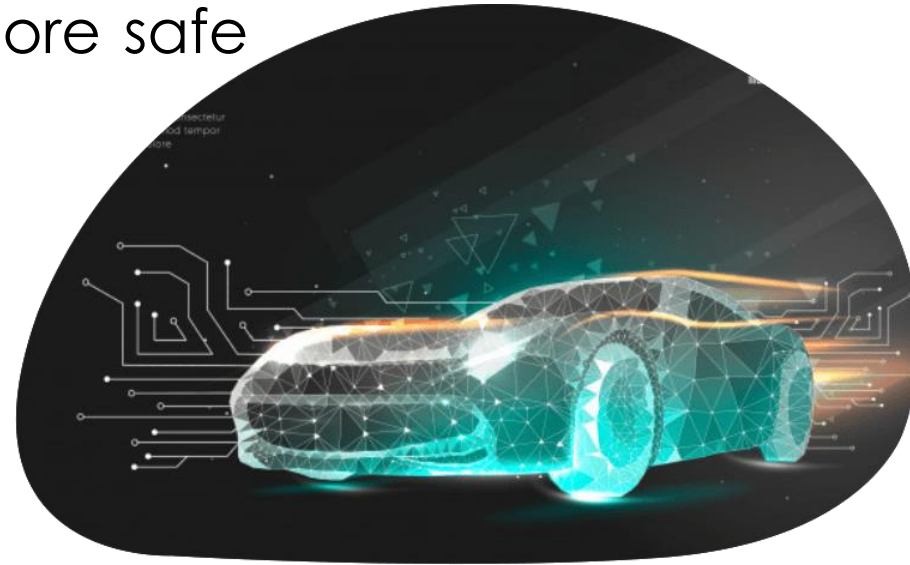
- **AI in Agriculture**
  - Artificial Intelligence is used to identify defects and nutrient deficiencies in the soil.
  - This is done using computer vision, robotics, and machine learning applications, AI can analyze where weeds are growing.
  - AI bots can help to harvest crops at a higher volume and faster pace than human laborers.

AI Applications

- **AI in Gaming**
  - Another sector where Artificial Intelligence applications have found prominence is the gaming sector. AI can be used to create smart, human-like NPCs to interact with the players.
  - It can also be used to predict human behavior using which game design and testing can be improved.

AI Applications

- **AI in Automotive Industry**
  - Some Automotive industries are using AI to provide virtual assistant to their user for better performance. Such as Tesla has introduced TeslaBot, an intelligent virtual assistant.
  - Various Industries are currently working for developing self-driven cars which can make your journey more safe and secure.



Other AI Applications

- AI in Education
- AI in Entertainment
- AI in Social Media
- AI in Travel & Transport
- AI in Data Security
- AI in Finance
- AI in Astronomy

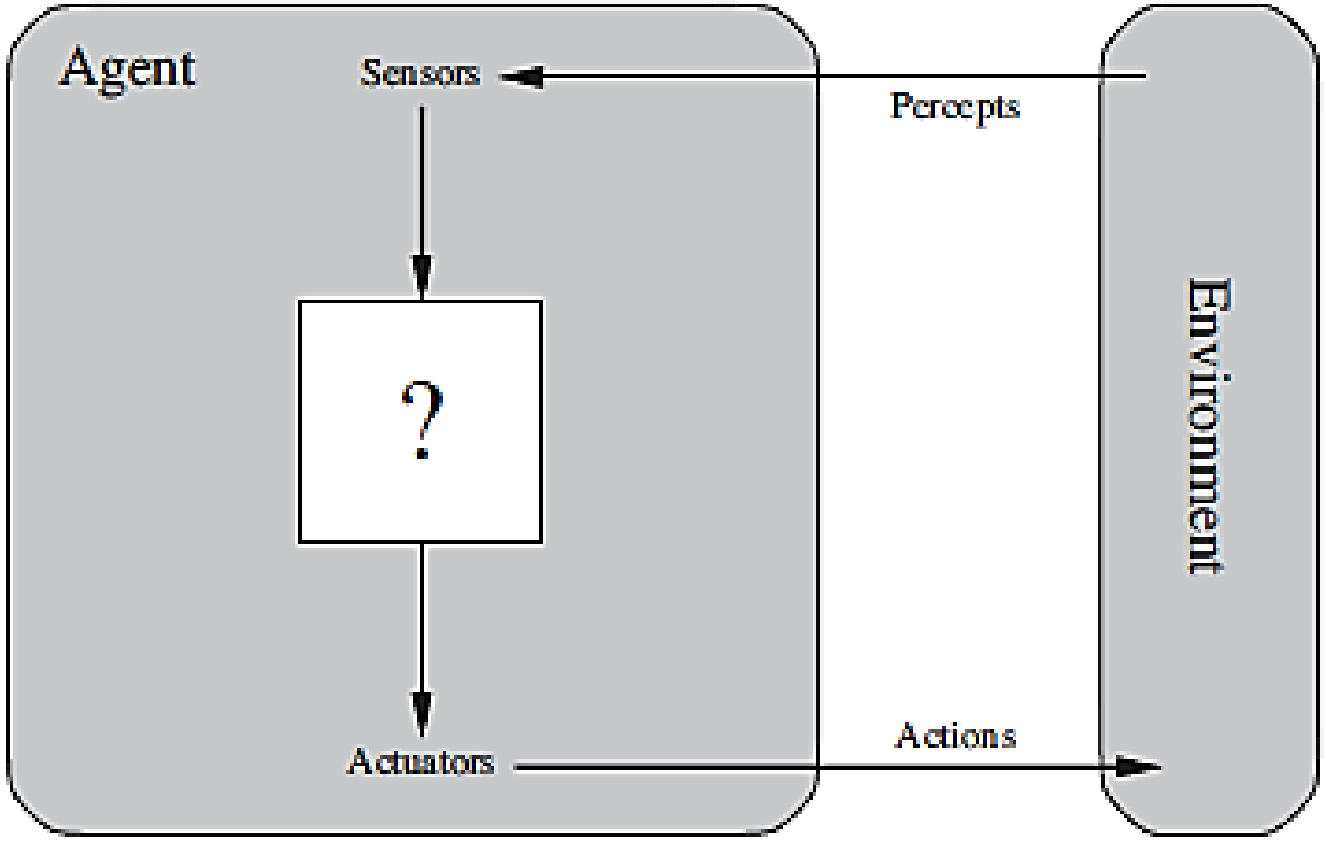
AI Agents

Systems that think like humans.	Systems that think/raationally.
Systems that act like humans.	Systems that act rationally.

- An agent is just something that acts.
- But computer agents are expected to have other attributes that distinguish them from more programs.
- A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

AI Agents

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.



AI Agents

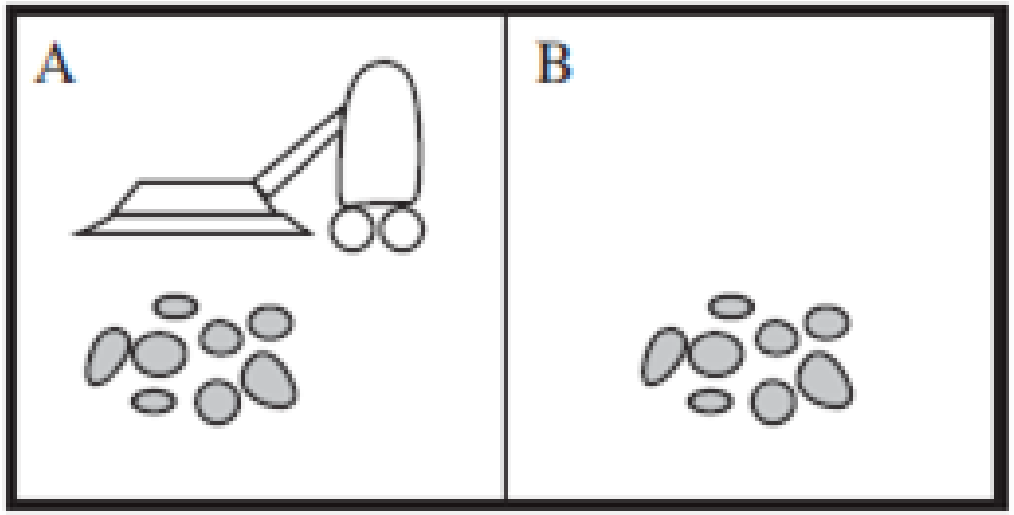
- A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

AI Agents

- We use the term percept to refer to the agent's perceptual inputs at any given instant. An agent's percept sequence is the complete history of everything the agent has ever perceived.
- In general, an agent's choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived.
- By specifying the agent's choice of action for every possible percept sequence, we have said more or less everything there is to say about the agent.

AI Agents

- This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. This particular world has just two locations: squares A and B.
- The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square



AI Agents

- The job of AI is to design an agent program that implements the agent function— the mapping from percepts to actions.
- We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the architecture:
- agent = architecture + program
- Agent programs take the current percept as input from the sensors and return an action to the actuators

Types of AI Agents

- Simple Reflects agents
- Model Based Reflects agents
- Goal based agents
- Utility based agents
- Learning agents

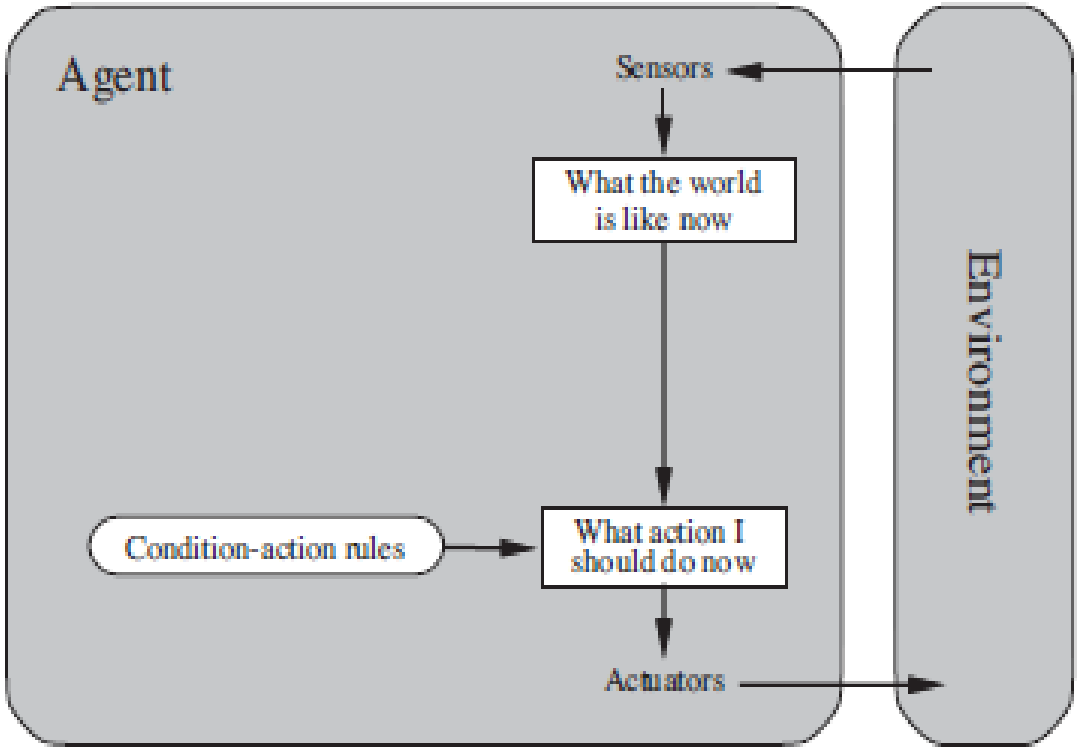
Simple Reflex Agents

- The simplest types of agent is the simple reflex agent
- These agents select actions on the basis of the current percept, ignoring the rest of the percept history. For example, the vacuum agent is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt.
- An agent program for this agent is shown below

```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

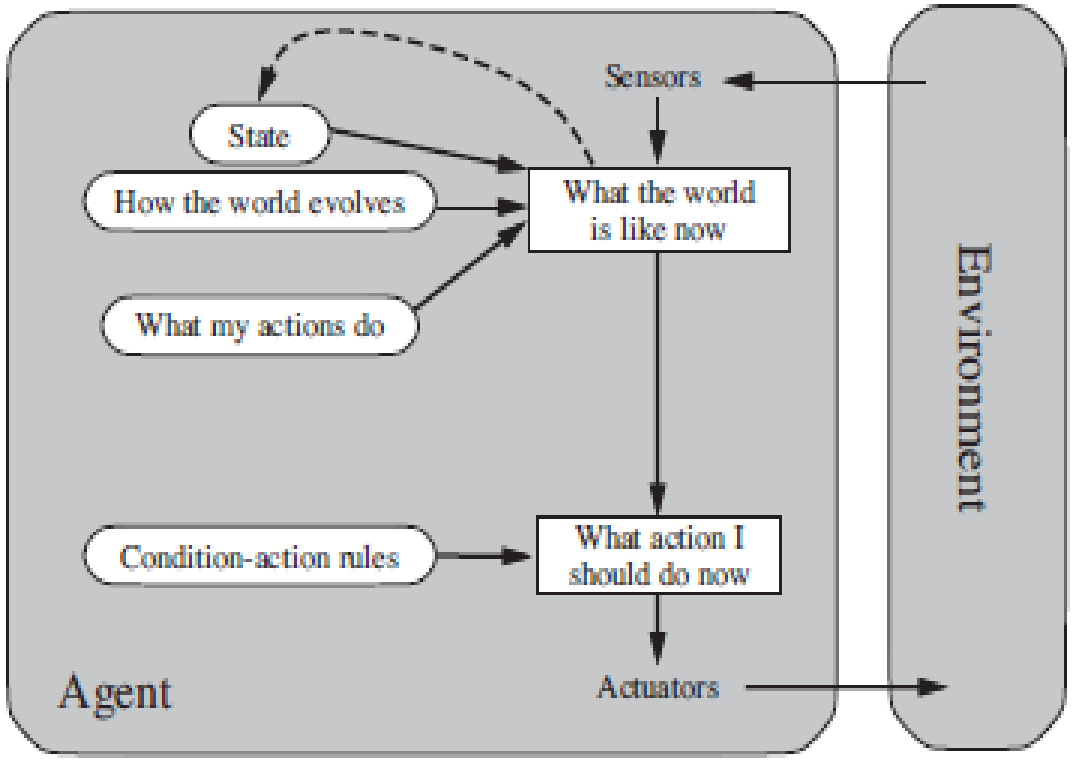
Simple Reflex Agents

- Above figure gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action.
- It is based on If-then else scenario
- Example: Sensor sense in a room that if temp>40, switch on AC



Model Based Reflects Agents

- Works by finding the rule whose conditions matches current situation
- Agent keeps track of internal state which is adjusted by each percept and that depends on percept history.



Model Based Reflects Agents

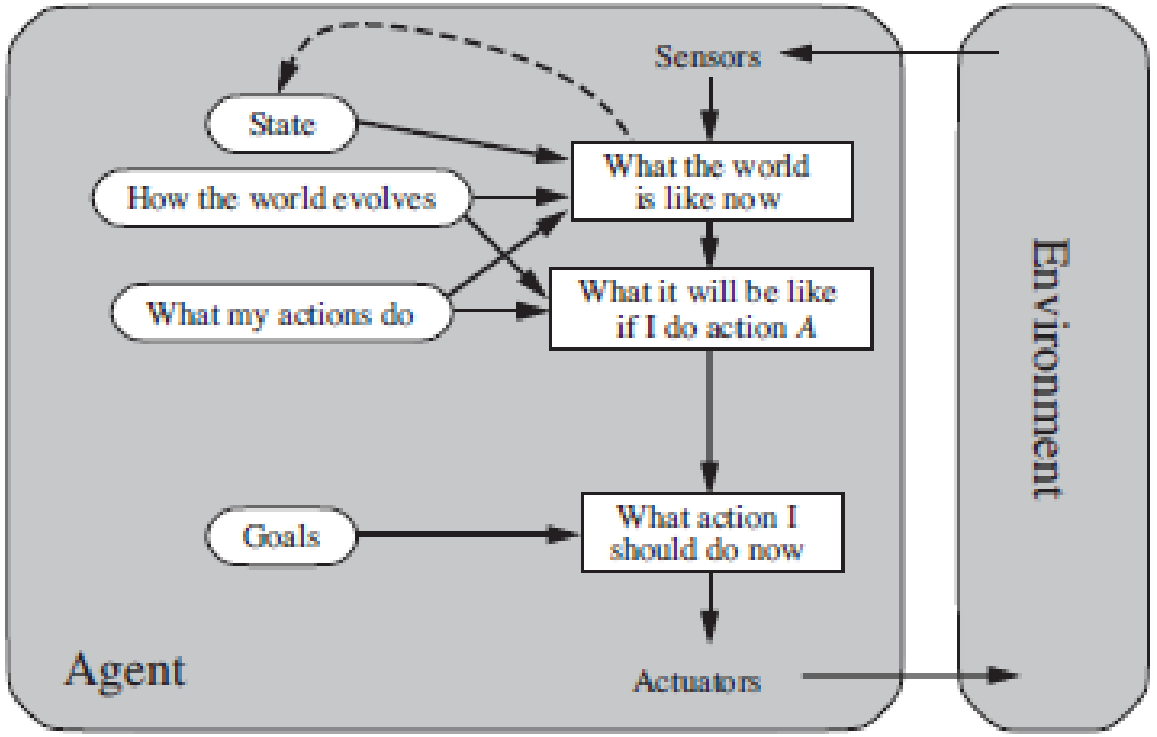
- Figure gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works.
- Worked on partially observable environment : Self driving car

Goal Based Agents

- Knowing something about the current state of the environment is not always enough to decide what to do.
- For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state description, the GOAL agent needs some sort of goal information that describes situations that are desirable—for example, being at the passenger's destination.
- The agent program can combine this with the model to choose actions that achieve the goal.

Goal Based Agents

- Expansion of model based agents
- Desirable situation (Goal)
- Based on searching and planning

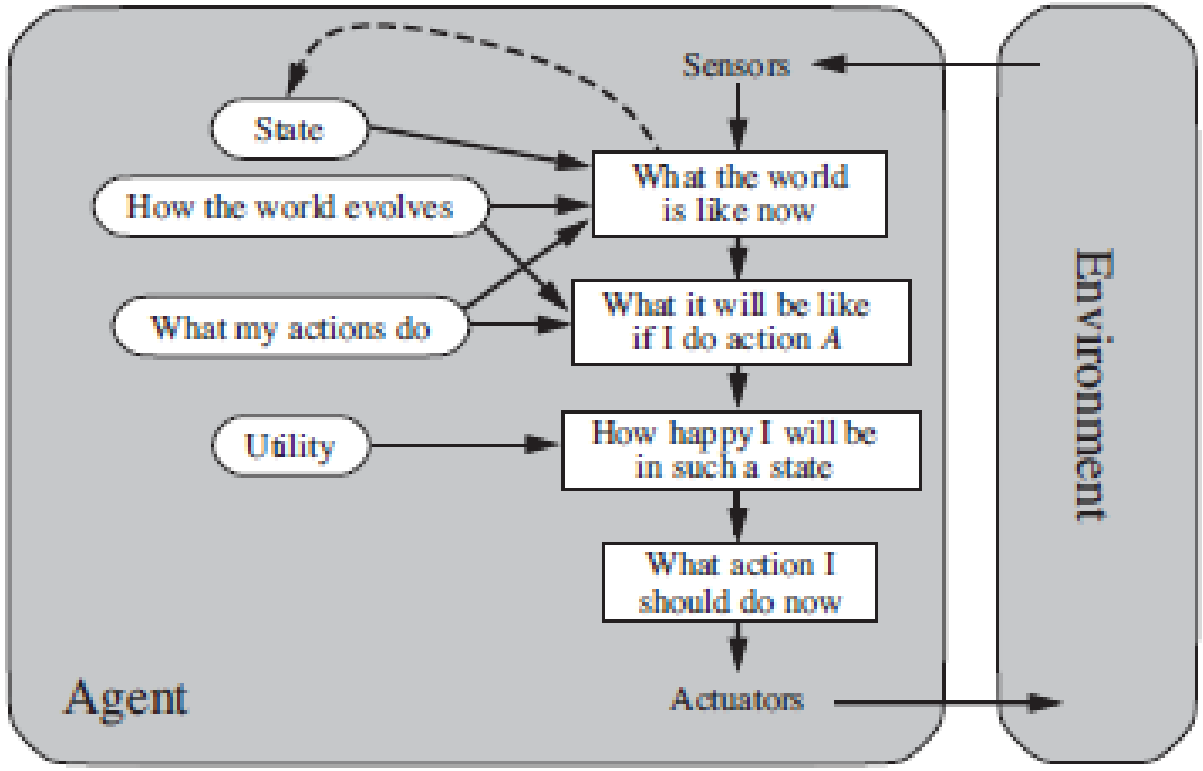


Utility Based Agents

- Goals alone are not enough to generate high-quality behavior in most environments.
- For example, many action sequences will get the taxi to its destination but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between “happy” and “unhappy” states.
- A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent.
- Because “happy” does not sound very scientific, economists and computer scientists use the term utility instead.

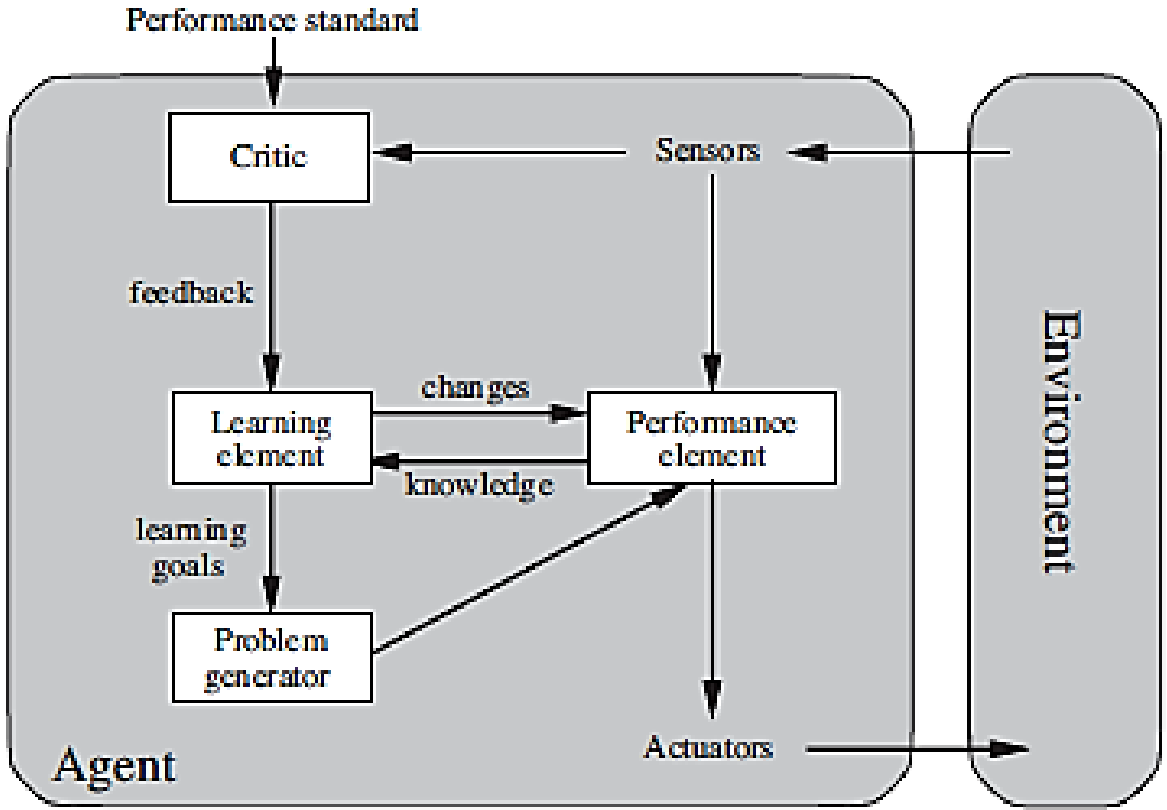
Utility Based Agents

- An agent's utility function is essentially an internalization of the performance measure.
- Focus on utility…not the goal only. There are conflicting goals, out of which only few can be achieved.
- Example: GPS system



Learning Agents

- Can learn from its past experiences.
- It states to act with basic knowledge and then able to act by adopting learning



Learning Agents

- It has four important components in it:
- **Learning Element:** It makes improvement in system by learning from environment
- **Critic:** It gives feedback about agent's performance based on certain standard
- **Performance Element:** It selects the actions to perform
- **Problem Generator:** It suggests the new action to take and from it new info. gets generated

Environment

- An **environment** is everything in the world which surrounds the agent.
- But it is not a part of agents.
- An environment can be described as a situation in which an agent is present.

Types of Environments

- **1. Fully observable (FO) vs Partially observable (PO)**
  - If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable.
  - A task environment is effectively fully observable if the sensors detect all aspects that are relevant to the choice of action; relevance, in turn, depends on the performance measure.

Types of Environments

- **1. Fully observable (FO) vs Partially observable (PO)**
  - Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world. An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.
  - **Example:**
  - **Chess –** the board is fully observable, so are the opponent's moves
  - **Driving –** the environment is partially observable because what's around the corner is not known

Types of Environments

- **2. Deterministic vs Stochastic**
  - If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic.
  - **Example:**
  - **Chess, 8-puzzle –** there would be only few possible moves for a moves at the current state and these moves can be determined
  - **Self Driving Cars –** the actions of a self driving car are not unique, it varies time to time

Types of Environments

- **3. Single agent vs Multi agent**
  - The distinction between single-agent and multiagent environments may seem simple enough.
  - **For example,** an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two agent environment.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku
single-agent environment

Ludo
Multi-agent environment

Types of Environments

- **4. Static vs Dynamic**
  - If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static.
  - **Static environments** are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time.
  - **Dynamic environments**, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing.
  - **Example:** physical world – Dynamic, empty office with no moving objects - Static

Types of Environments

- **5. Discrete vs Continuous**
  - **Discrete:** finite number of actions that can be performed within it.
  - **Continuous:** the actions performed cannot be numbered
  - **Example:**
  - **Chess –** discrete as it has only a finite number of moves.
  - **Self Driving Cars –** actions are driving, parking, etc.

Problem Formulation & Solution by Search

Problem Formulations

- Problem-solving agent is a result-driven agent and always focuses on satisfying the goals.
- The **problem of AI** is directly associated with the nature of humans and their activities. So we need a number of finite steps to solve a problem which makes human easy works.
- Problem Formulation & Method Solving in Artificial Intelligence (AI) organizes several steps to formulate a target/goals which require a specific action to achieve the goal.

Problem Formulations

- These are the following steps which require to solve a problem :
  - **Goal Formulation:** This one is the first and simple step in problem-solving. It organizes finite steps to formulate a target/goals which require some action to achieve the goal. Today the formulation of the goal is based on AI agents.
  - **Problem formulation:** It is one of core steps of problem-solving which decides what action should be taken to achieve the formulated goal. In AI this core part is dependent upon software agent which consisted of the following components to formulate the associated problem.

Problem Formulations

- **Transition:** This stage of problem formulation integrates the actual action done by the previous action stage and collects the final stage to forward it to their next stage.
- **Goal test:** This stage determines that the specified goal achieved by the integrated transition model or not, whenever the goal achieves stop the action and forward into the next stage to determines the cost to achieve the goal.
- **Path costing:** This component of problem-solving numerical assigned what will be the cost to achieve the goal. It requires all hardware software and human working cost.

Problem Formulations

- **Some methods used in Problem Formulation are :-**
  - Tree structure
  - Graphical model
  - Implementation of graph

- **Click Here**

General Problem Solver

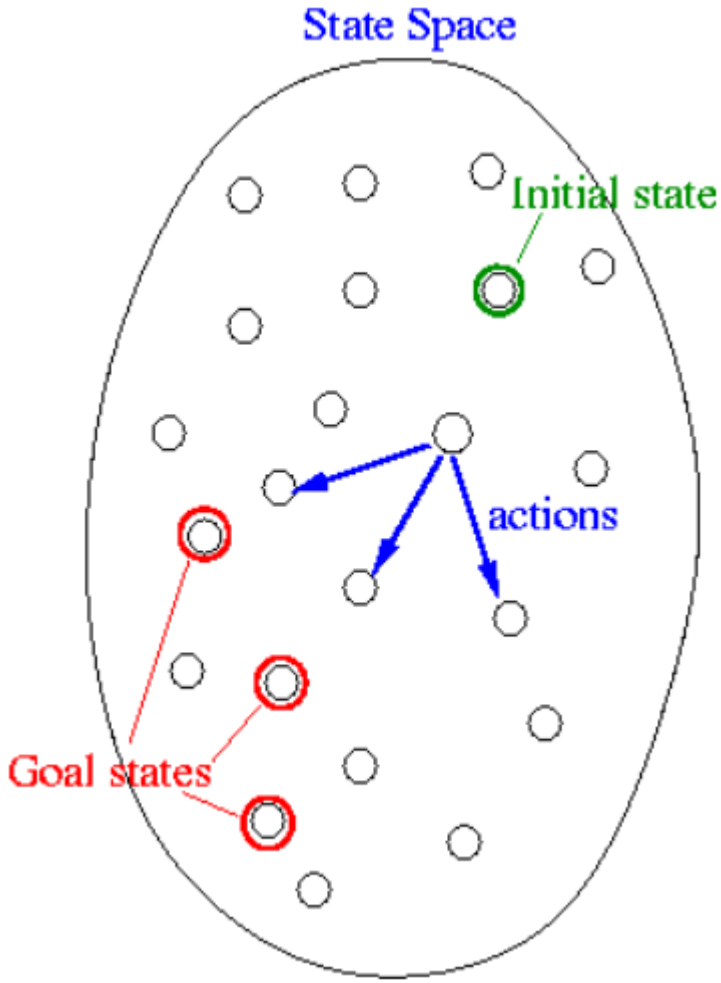
- GPS (General Problem Solver) focus on systems with general capability for solving different types of problems
- Problems represented in terms of
  - **Initial state**
  - **Final state** (goal state)
  - A **set of legal transitions** to transfer states into new states
- Using **states & operators**, GPS generates sequence of transitions that transform initial state into final state

General Problem Solver

- Efficiency in choosing path to reach the goal
- GPS did not use specific info about problem at hand in selection of state transition
- GPS examined all states leading to exponential time complexity
- Breakthrough in AI towards more specialized problem solving system, i. e., Knowledge Based Systems

State Space Search of Problem

- A search problem consists of the following:
  - **S:** the full set of states
  - **s0 :** the initial state
  - **A:S→S** is a set of actions or operators
  - **G** is the set of final states. Note that **G ⊆S**



State Space Search of Problem

- The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state **g∈G**. A search problem is represented by a **4-tuple {S, s0, A, G}**.
  - **S:** set of states (State Space),
  - **s0 ∈ S :** initial state,
  - **A: Si→Sj** is a set of operators to move from state 'i' to state 'j' that transform one state to another state,
  - **G :** goal, a set of states. **G ⊆ S**, the goal state **g ∈ G**.
- **P = {a0, a1, … , aN}** which leads to traversing a number of states **{s0, s1, … , sN+1∈G}.**
- A search problem, represented by sequence of actions of **4-tuple {S, s0, A, G}** is called a Solution Plan. It is a path from the initial state to a goal state. A plan P is a sequence of actions, **P = {a0, a1, … , aN}** which leads to traversing a number of states **{s0, s1, … , sN+1∈G}.**
- A sequence of states is called a **path**. **The cost of a path** is a positive number. In many cases the path cost is computed by taking the sum of the costs of each action.

Representation of search problems

- A search problem is represented using a directed graph.
  - The states are represented as nodes.
  - The allowed actions are represented as arcs.

Searching process to solve Problem

- Do until a solution is found or the state space is exhausted.
  1. Check the current state
  2. Execute allowable actions to find the successor states.
  3. Pick one of the new states.
  4. Check if the new state is a solution state
- If it is not, the new state becomes the current state and the process is repeated

Basic Search Algorithm to solve Problem

```
Let L be a list containing the initial state (L= the fringe)
Loop
        if L is empty return failure
        Node ← select (L)
          if Node is a goal
                    then return Node
                        (the path from initial state to Node)
        else generate all successors of Node, and
              merge the newly generated states into L
End Loop
```

Search Tree – Terminology

- **Root Node:** The node from which the search starts.
- **Leaf Node:** A node in the search tree having no children.
- **Ancestor/Descendant:** X is an ancestor of Y is either X is Y's parent or X is an ancestor of the parent of Y. If S is an ancestor of Y, Y is said to be a descendant of X.
- **Branching factor:** the maximum number of children of a non-leaf node in the search tree
- **Path:** A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.

Search Tree – Node data structure

- A node used in the search algorithm is a data structure which contains the following:
  1. A state description
  2. A pointer to the parent of the node
  3. Depth of the node
  4. The operator that generated this node
  5. Cost of this path (sum of operator costs) from the start state

Evaluating Search strategies

- Three factors to measure this:

1. **Completeness:** Is the strategy guaranteed to find a solution if one exists?
2. **Optimality:** Does the solution have low cost or the minimal cost?
3. What is the search cost associated with the time and memory required to find a solution?
    a. **Time complexity**: Time taken (number of nodes expanded) (worst or average case) to find a solution.
    b. **Space complexity**: Space used by the algorithm measured in terms of the maximum size of fringe

Problem-1
Water Jug Problem

You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

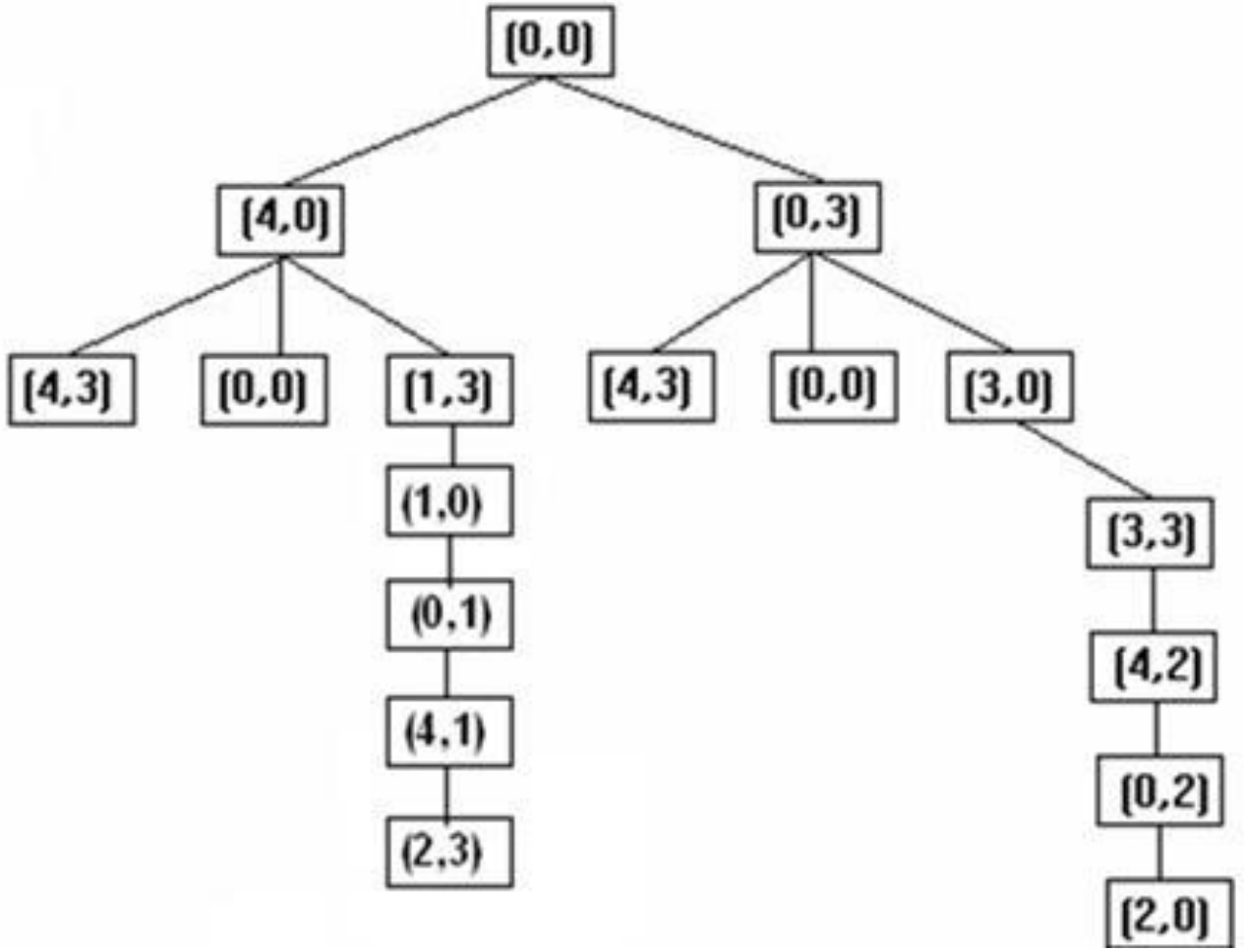
Problem Formulation of Water Jug Problem

- Let X represents the content of the water in 4-gallon jug.
- Let Y represent the content of the water in 3-gallon jug.
- **Initial State:** All Two Jugs are empty. (X=0, Y=0)
- **Goal State:** FIRST Jug contains exactly two gallons of water. (X=2, Y=Any No.)
- **Cost Function:** Charge one point for each gallons of water transferred and each gallon of water filled or emptied.
- **Successor Function:**
  - This function generates next state of problem by applying action on current (input) state.
- **List of Actions: mentioned in next slide**

Actions/Rules of water jug problem

1. Fill 4-gal jug	(x,y) $x < 4$	→	(4,y)
2. Fill 3-gal jug	(x,y) $y < 3$	→	(x,3)
3. Empty 4-gal jug on ground	(x,y) $x > 0$	→	(0,y)
4. Empty 3-gal jug on ground	(x,y) $y > 0$	→	(x,0)
5. Pour water from 3-gal jug to fill 4-gal jug	(x,y) $0 < x+y \geq 4$ and $y > 0$	→	(4, y - (4 - x))
6. Pour water from 4-gal jug to fill 3-gal-jug	(x,y) $0 < x+y \geq 3$ and $x > 0$	→	(x - (3-y), 3)
7. Pour all of water from 3-gal jug into 4-gal jug	(x,y) $0 < x+y \leq 4$ and $y \geq 0$	→	(x+y, 0)
8. Pour all of water from 4-gal jug into 3-gal jug	(x,y) $0 < x+y \leq 3$ and $x \geq 0$	→	(0, x+y)

Search tree of Water Jug Problem (Not Fully generated tree)



Solution

Current State	Next State	Action Rule number
(0, 0)	(0, 3)	2
(0, 3)	(3, 0)	7
(3, 0)	(3, 3)	2
(3, 3)	(4, 2)	5
(4, 2)	(0, 2)	3
(0, 2)	(2, 0)	7
(2, 0)	Final	-

State Space Search Algorithm

State Space Search Categories

- A search procedure is a strategy for selecting the order in which nodes are generated and a given path selected.
- Classified in these categories:
  1. Blind or uninformed search
  2. Informed or directed search or heuristic search
  3. Constraint Satisfaction Search
  4. Adversarial Search
  5. Minimax Search
  6. Alpha Beta Pruning Search

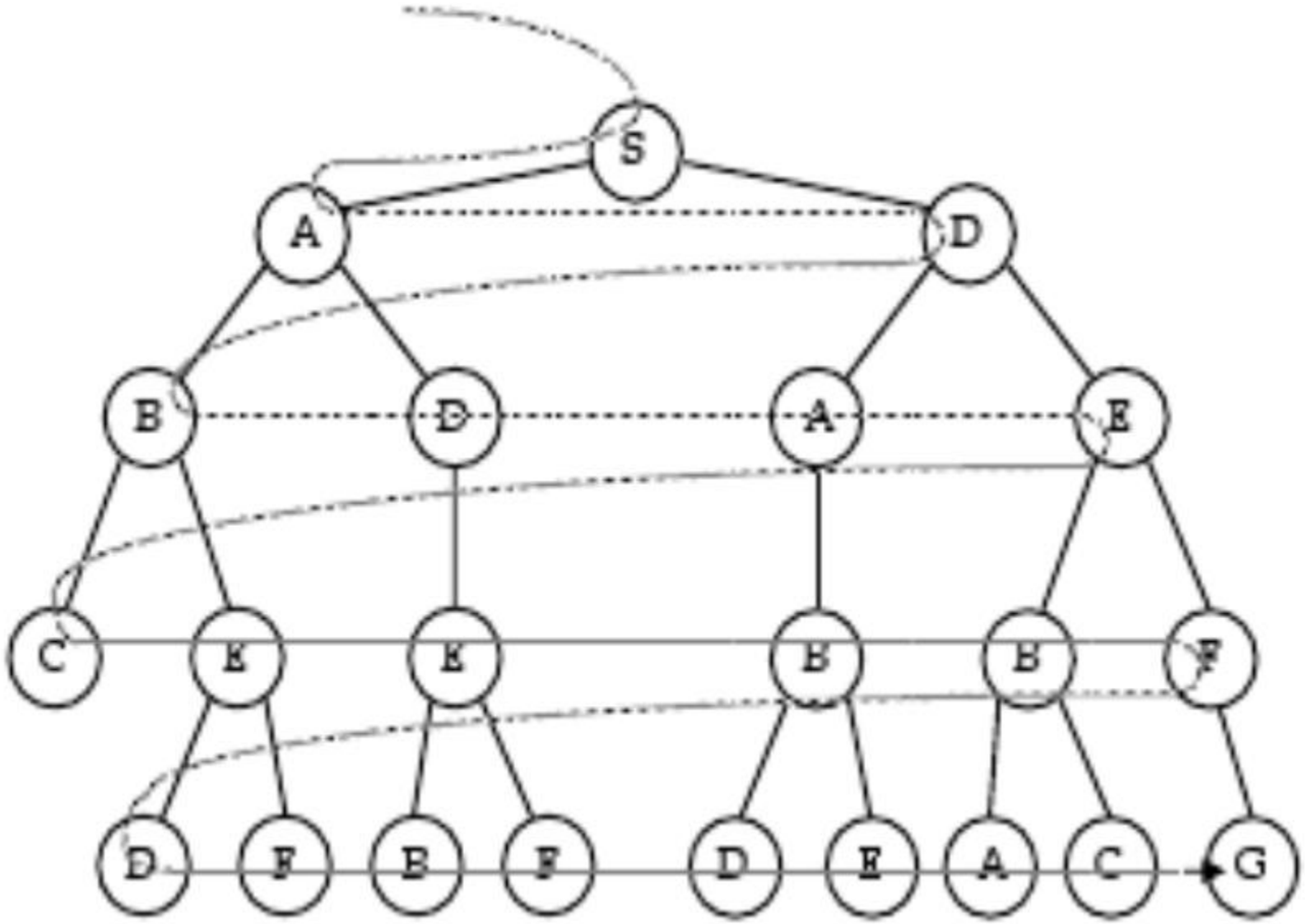
Blind Search or Uninformed Search

- Blind search algorithms
  1. Breadth First Search (BFS)
  2. Depth First Search (DFS)
  3. Depth Limited Search
  4. Iterative Deepening Search
  5. Bidirectional Search
  6. Uniform Cost Search

Breadth First Search (BFS) Algorithm

1. Initialize queue by adding root node (state)
2. If queue is empty then
    1. return “Failure”
3. else
    1. current_node = pop node from queue
    2. If current_node is goal_node then
        1. return current_node
    3. else
        1. child_list = generateAllSuccessors(Current_node,All_actions)
        2. Push all child_list in queue
    4. Continue Loop from step-2

Breadth First Search (BFS)

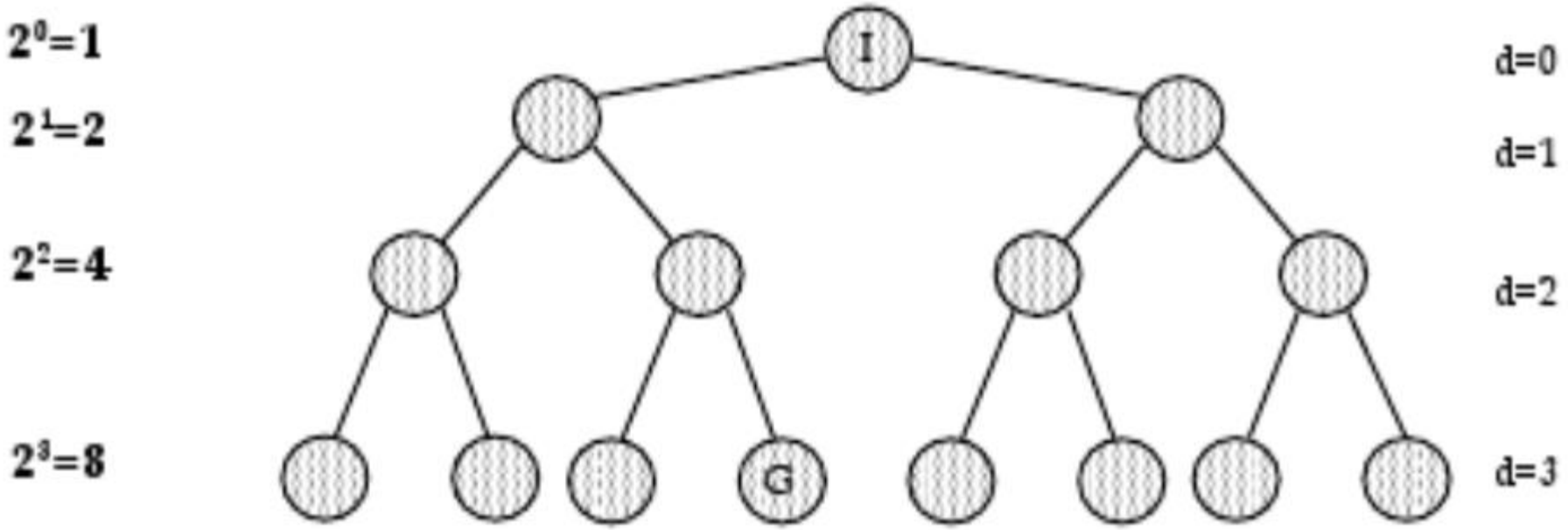


Evaluation of Breadth First Search (BFS)

- Search is performed by extracting from the front and inserting on the back of the list (FIFO)
- The algorithm proceed level by level, hence it is
  - **Complete**
  - **Optimal** (will find the solution at smallest depth)

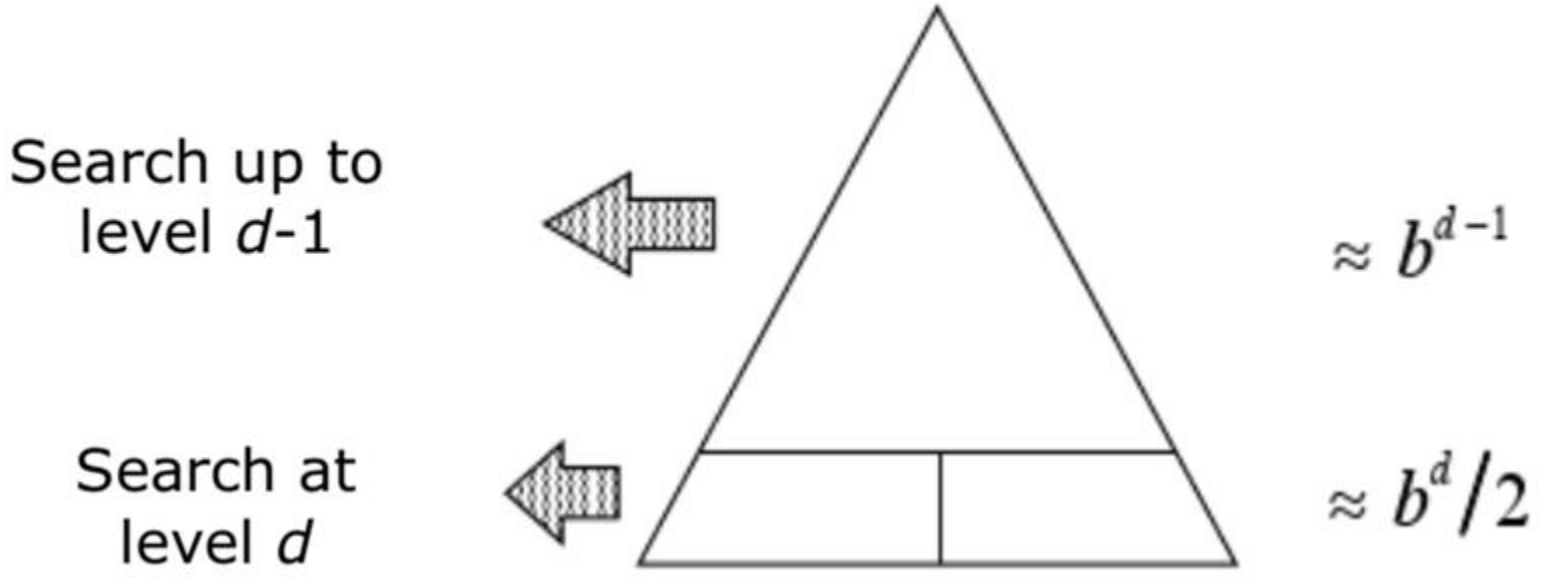
Evaluation of Breadth First Search (BFS)

- Simplifying hypotheses:
  - The search tree has constant branching factor **b**
  - The first goal is at depth **d**
  - **Observation: the number of nodes at level d is b^d**



Evaluation of Breadth First Search (BFS)

- The number of nodes to examine to reach depth **d** is:
- $1 + b^1 + b^2 + b^3 + \cdots + b^{d-1} = \frac{b^d - 1}{b-1}$
- **Time Complexity = Space Complexity = $O(b^d)$**
- **Note:** time complexity is dominated by search at the last level



BFS Disadvantages

- Consider a complete search tree of depth 15, where every node at depths 0 to14 has 10 children and every node at depth 15 is a leaf node.
- The complete search tree in this case will have $O(10^{15})$ nodes. If BFS expands 10000 nodes per second and each node uses 100 bytes of storage, then BFS will take **3500 years** to run in the worst case, and it will use **11100 terabytes** of memory.
- So you can see that the breadth first search algorithm cannot be effectively used unless the search space is quite small.
- You may also observe that even if you have all the time at your disposal, the search algorithm cannot run because it will run out of memory very soon.

BFS Disadvantages

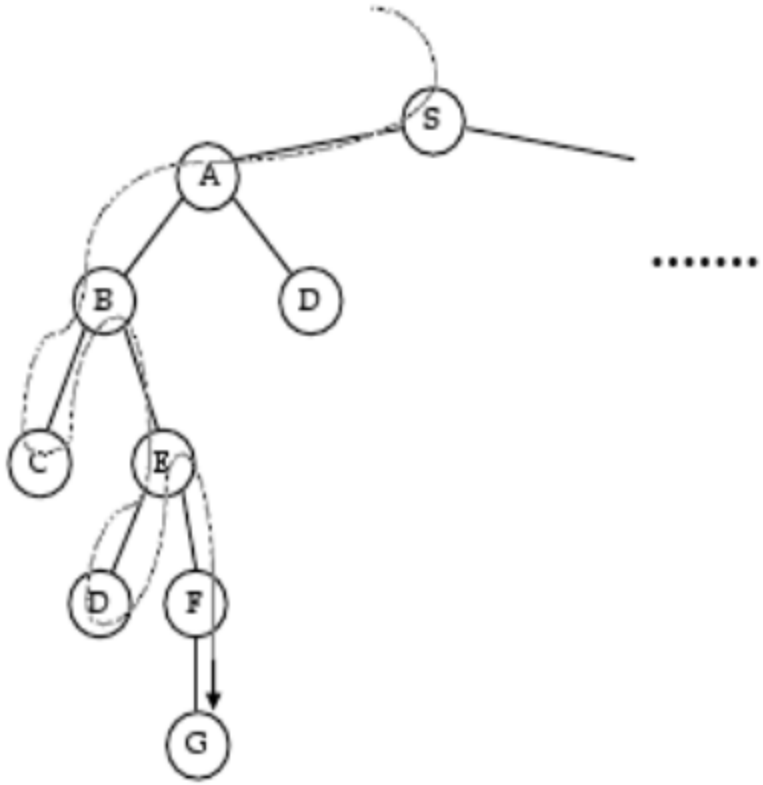
Depth	Nodes	Time	Memory
0	1	1 millisecond	100 kbytes
2	111	0.1 second	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Depth First Search (DFS) Algorithm

1. Initialize stack by adding root node (state)
2. If stack is empty then
    1. return “Failure”
3. else
    1. current_node = pop node from stack
    2. If current_node is goal_node then
        1. return current_node
    3. else
        1. child_list = generateAllSuccessors(Current_node,All_actions)
        2. Push all child_list in stack
    4. Continue Loop from step-2

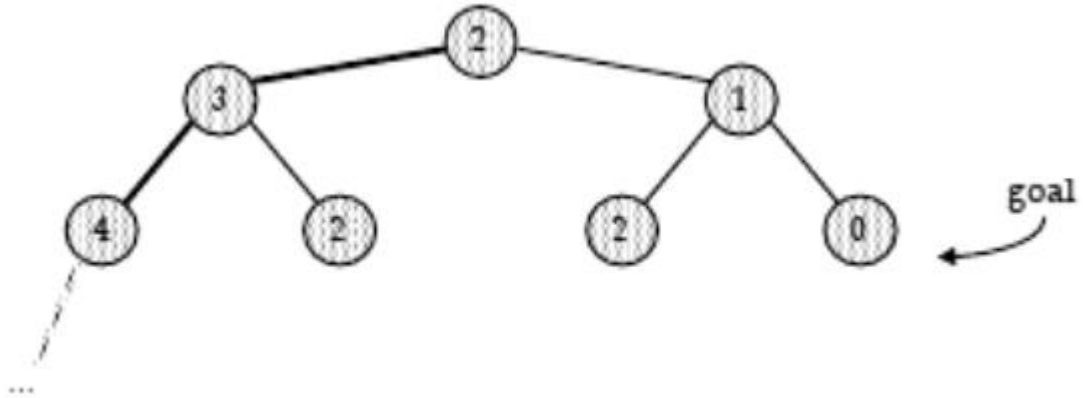
Depth First Search (DFS) Algorithm

- Search is performed by extracting and inserting from the front of the list (LIFO)



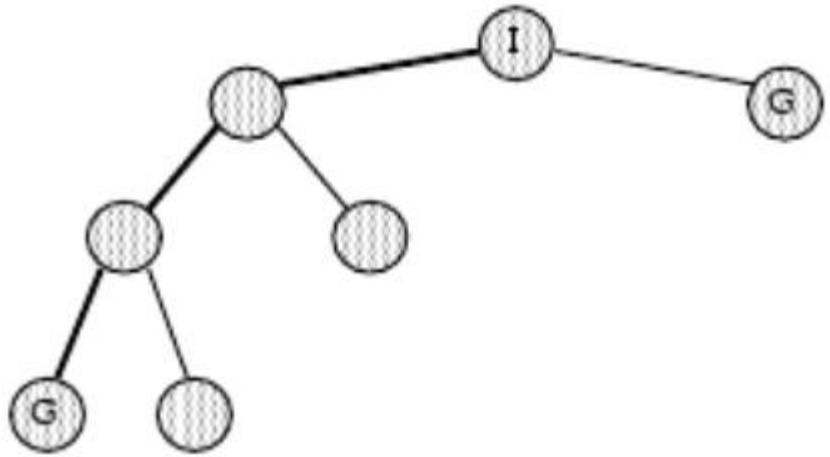
Evaluating Depth First Search (DFS) Algorithm

- If the tree has infinite depth, the algorithm might get stuck in an infinite branch and not find a solution even when one exists
- The algorithm is **not complete**



Evaluating Depth First Search (DFS) Algorithm

- If the problem has more than one solution the algorithm is not guaranteed to find the one at minimum depth
- The algorithm **is not optimal**



Evaluating Depth First Search (DFS) Algorithm

- Simplifying hypotheses:
- The search tree has constant **branching factor b**
- The tree has **depth d**
- There is only one goal and it is at **depth d**
- **Time Complexity =** $O(b^d)$
- **Space Complexity =** $O(db)$

Comparison of BFS & DFS

	Breadth-First	Depth-First
Complete	yes	no
Optimal	yes	no
Time	b^d	b^d
Space	b^d	bd

Problem-2
8 Puzzle Problem

8-Puzzle Problem

- In the 8-puzzle problem we have a 3×3 square board and 8 numbered tiles.
- The board has one blank position. Blocks can be slid to adjacent blank positions. We can alternatively and equivalently look upon this as the movement of the blank position up, down, left or right.
- The objective of this puzzle is to move the tiles starting from an initial position and arrive at a given goal configuration.
- The 15-puzzle problems is similar to the 8-puzzle. It has a 4×4 square board and 15 numbered tiles.

Initial State

Goal State

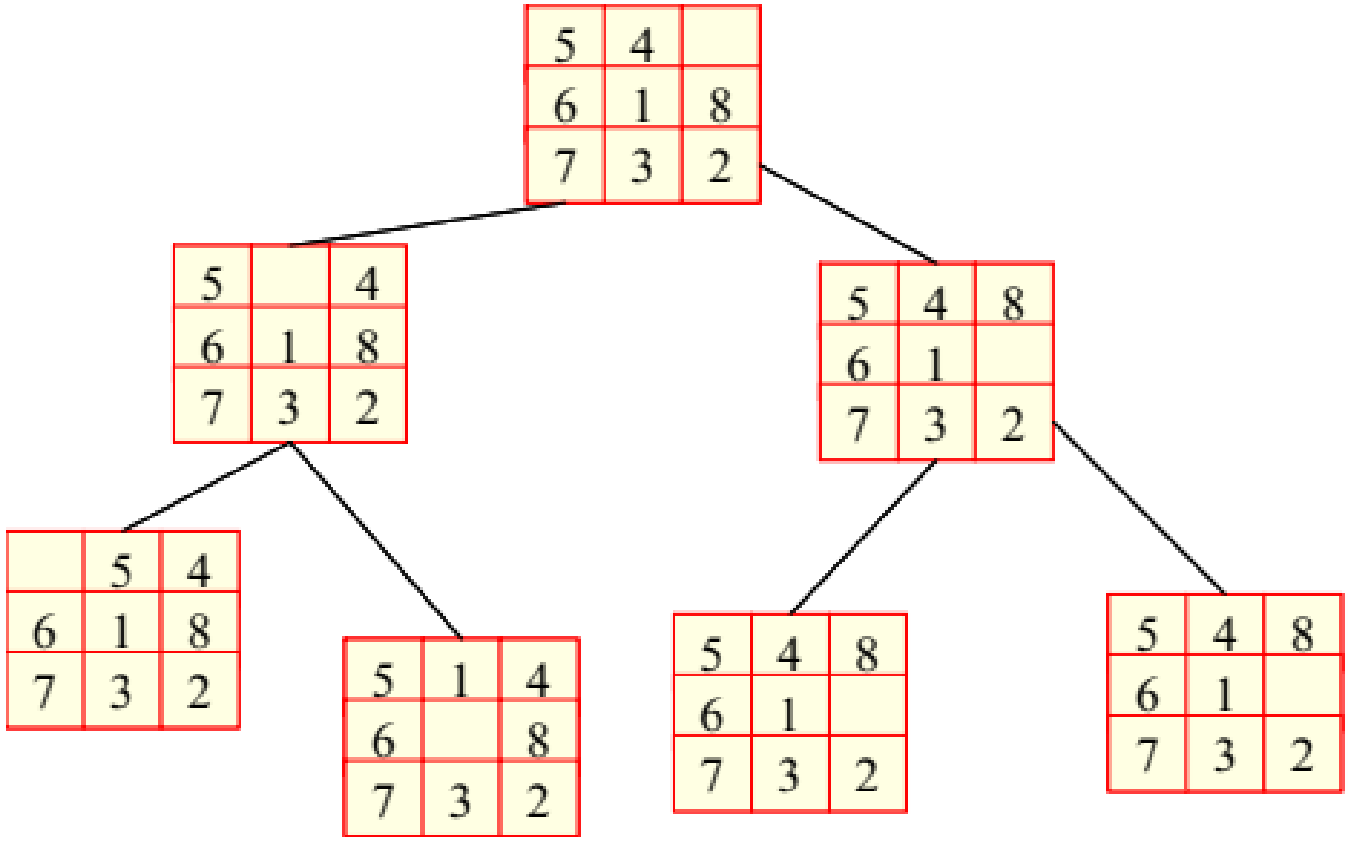
Problem formulation of 8-Puzzle

- **States:** A state is a description of each of the eight tiles in each location that it can occupy.
- **Operators/Action:** The blank moves left, right, up or down
- **Goal State:** The current state matches a certain state (e.g. one of the ones shown in image)
- **Path Cost:** Each move of the block costs 1

Initial State

Goal State

Search Tree of 8 Puzzle Problem (Not Fully generated tree)



Problem-3
N-Queen Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal. **N>3**

N queens problem formulation 1

- **States:** Any arrangement of 0 to N queens on the board
- **Initial state:** 0 queens on the board
- **Successor function:** Add a queen in any square
- **Goal State: N** queens on the board, none are attacked

N queens problem formulation 2

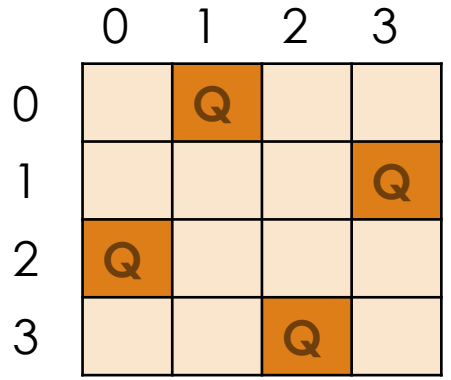
- **States:** Any arrangement of N queens on the board
- **Initial state:** All queens are at column 1
- **Successor function:** Change the position of any one queen
- **Goal State:** N queens on the board, none are attacked

N queens problem formulation 3

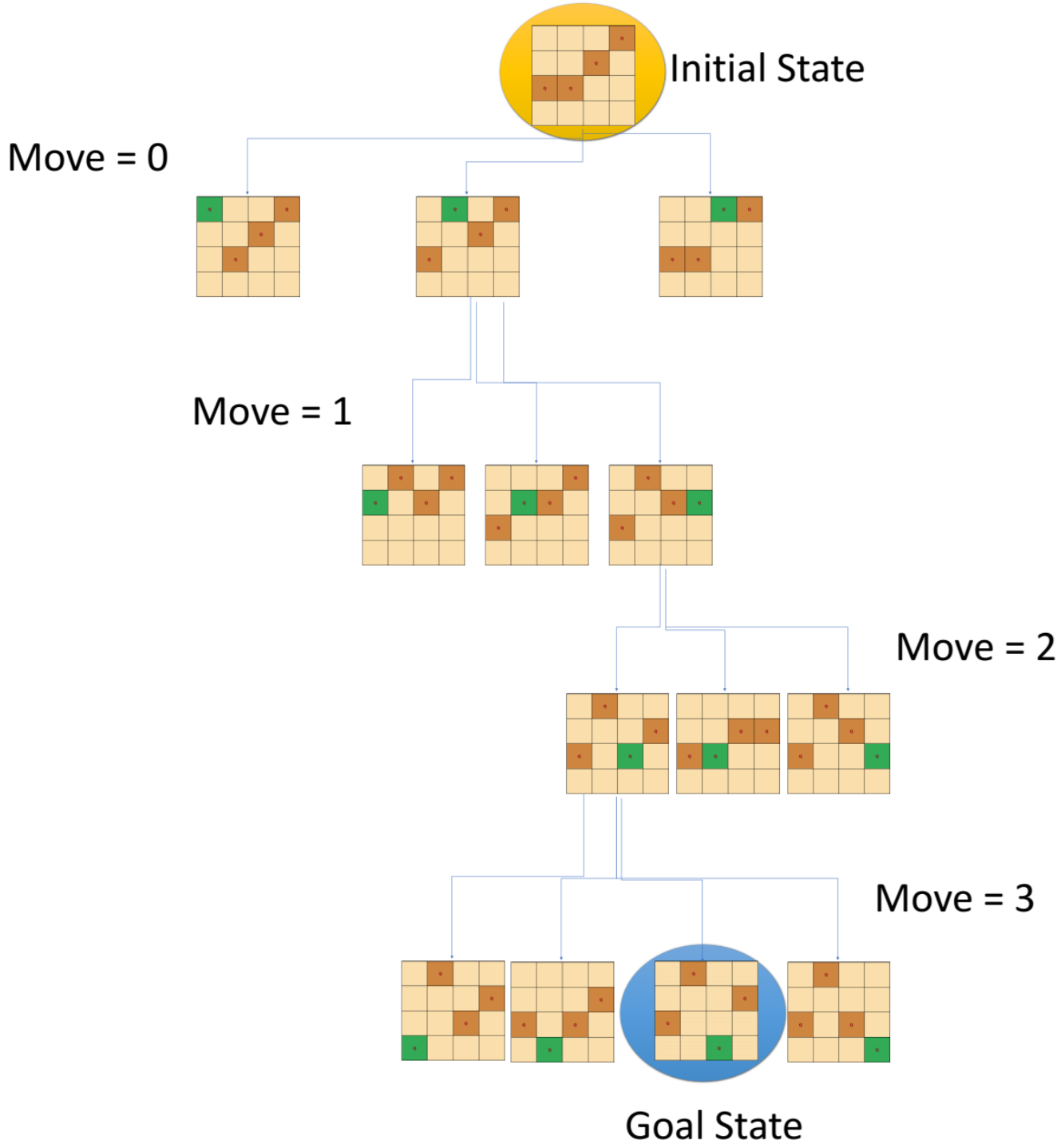
- **States:** Any arrangement of k queens in the first k rows such that none are attacked
- **Initial state:** 0 queens on the board
- **Successor function:** Add a queen to the (k+1)th row so that none are attacked.
- **Goal State : N** queens on the board, none are attacked

N queens problem formulation & Data Structure

- **States:** Any arrangement of N queens on the board
- **Initial state:** Random Arrangements of Queen
- **Successor function:** Add a queen to the (Move=k)th row so that none are attacked.
- **Goal State : N** queens on the board, none are attacked
- **Queen Data Structure:** Store value of Row, Column of Queen
- **State Data Structure:**
  1. List of Queens
  2. No. of Moves
  3. Parent pointer

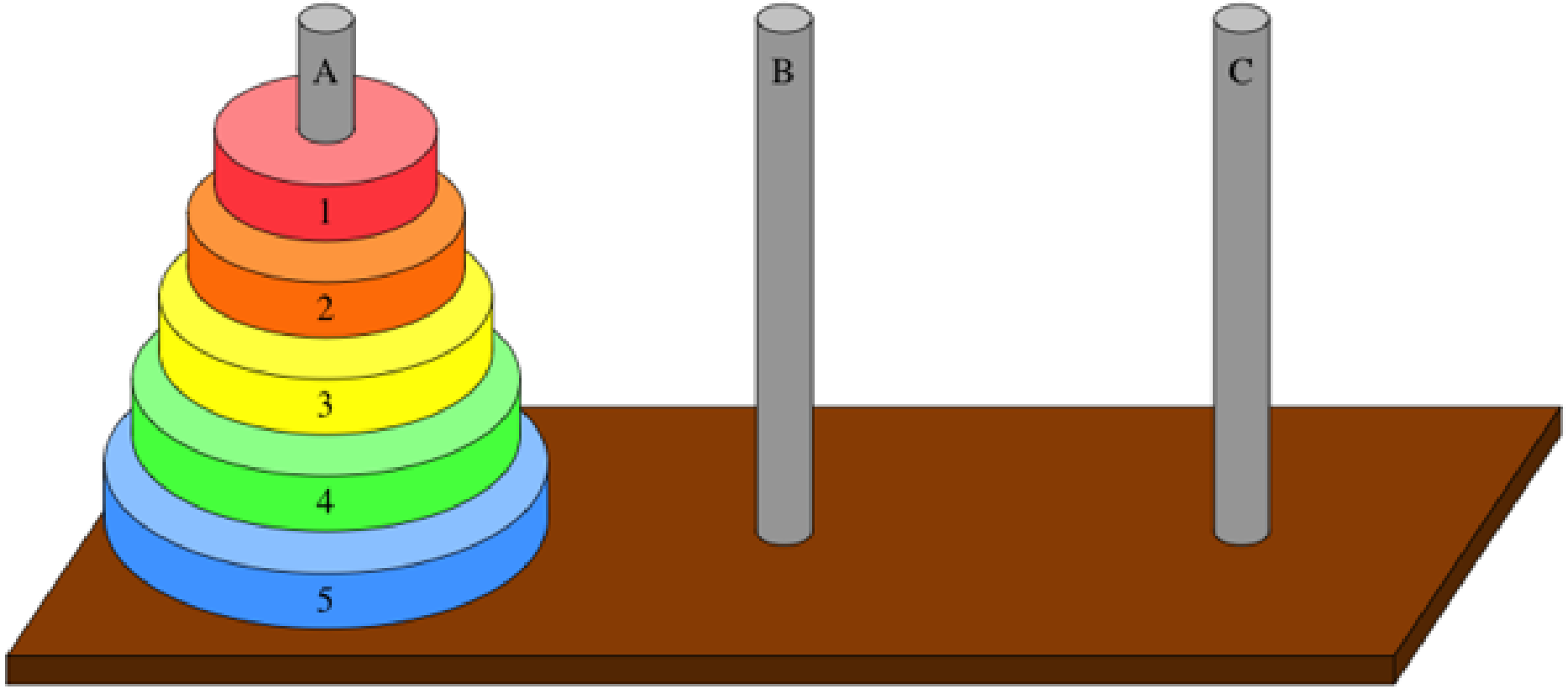


Search Tree N Queen



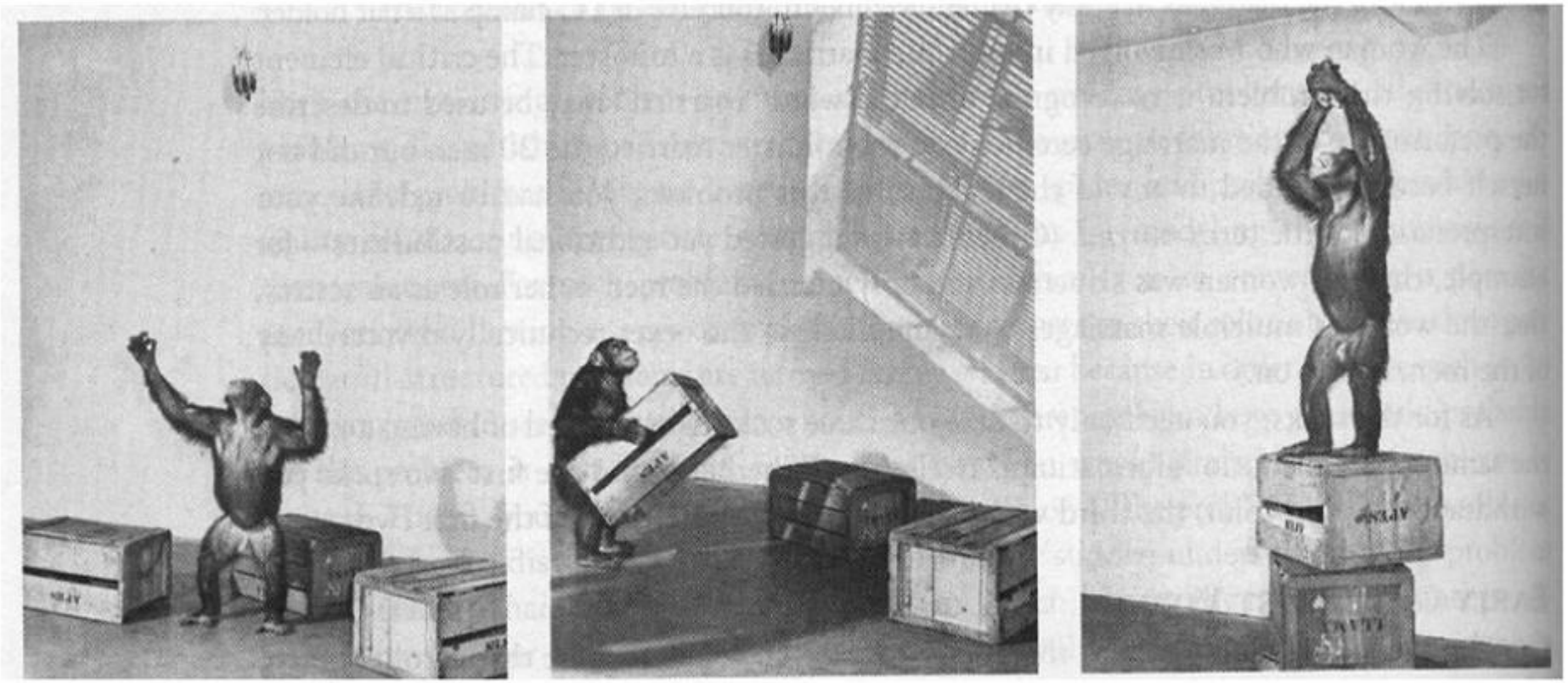
Problem-4
Tower of Hanoi

Tower of Hanoi



Problem-5
MONKEY BANANA PROBLEM

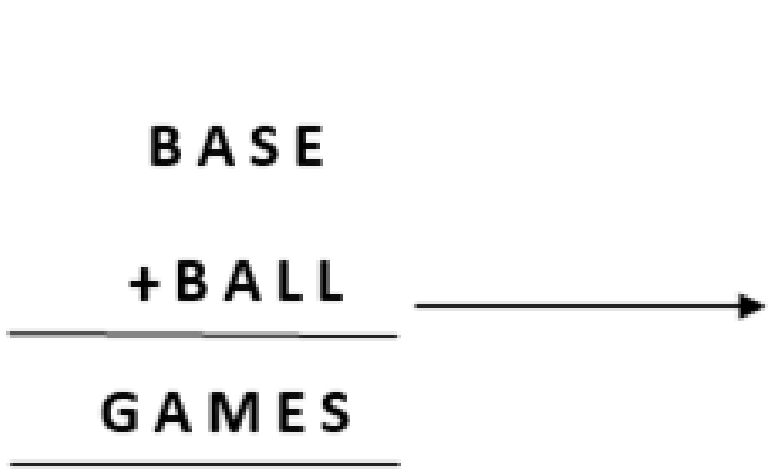
MONKEY BANANA PROBLEM



Kohler observed that chimpanzees appeared to have an insight into the problem before solving it

Problem-6
CRYPT ARITHMETIC PROBLEM

CRYPT ARITHMETIC

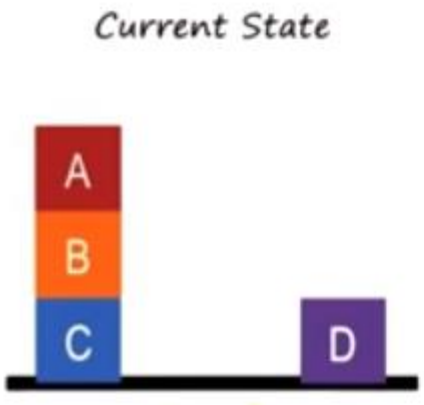


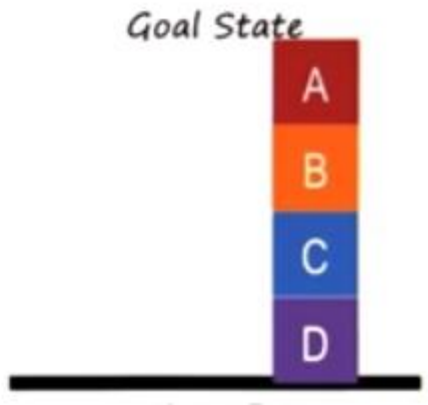
B	7
A	4
S	8
E	3
L	5
G	1
M	9

Problem-6
BLOCK WORLD PROBLEM

Block World Problem

- There is a table on which some blocks are placed.
- Some blocks may or may not be stacked on other blocks.
- We have a robot arm to pick up or put down the blocks.
- The robot arm can move only one block at a time, and no other block should be stacked on top of the block which is to be moved by the robot arm.



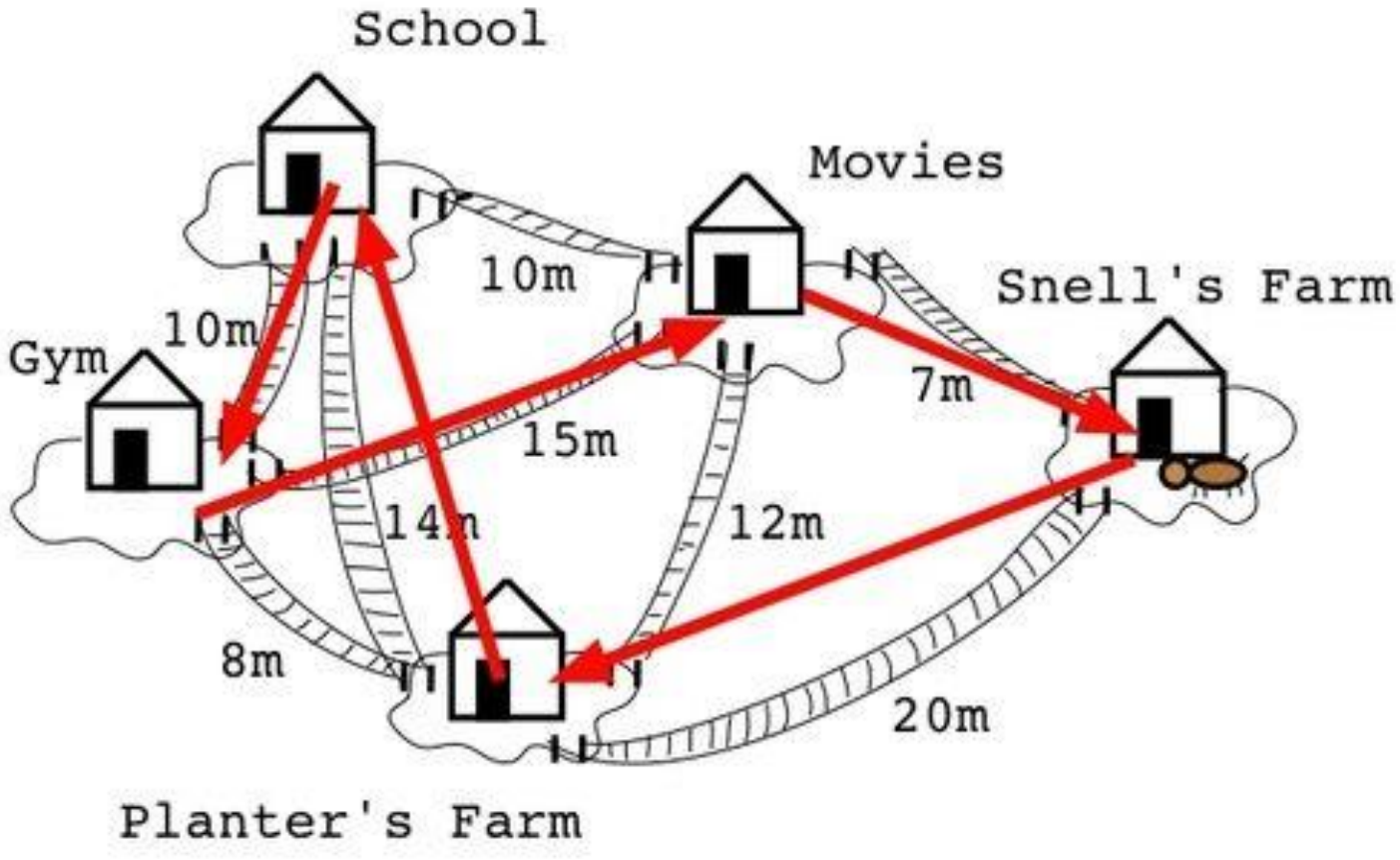


Problem-7
MISSIONARIES & CANNIBALS PROBLEM

MISSIONARIES & CANNIBALS PROBLEM

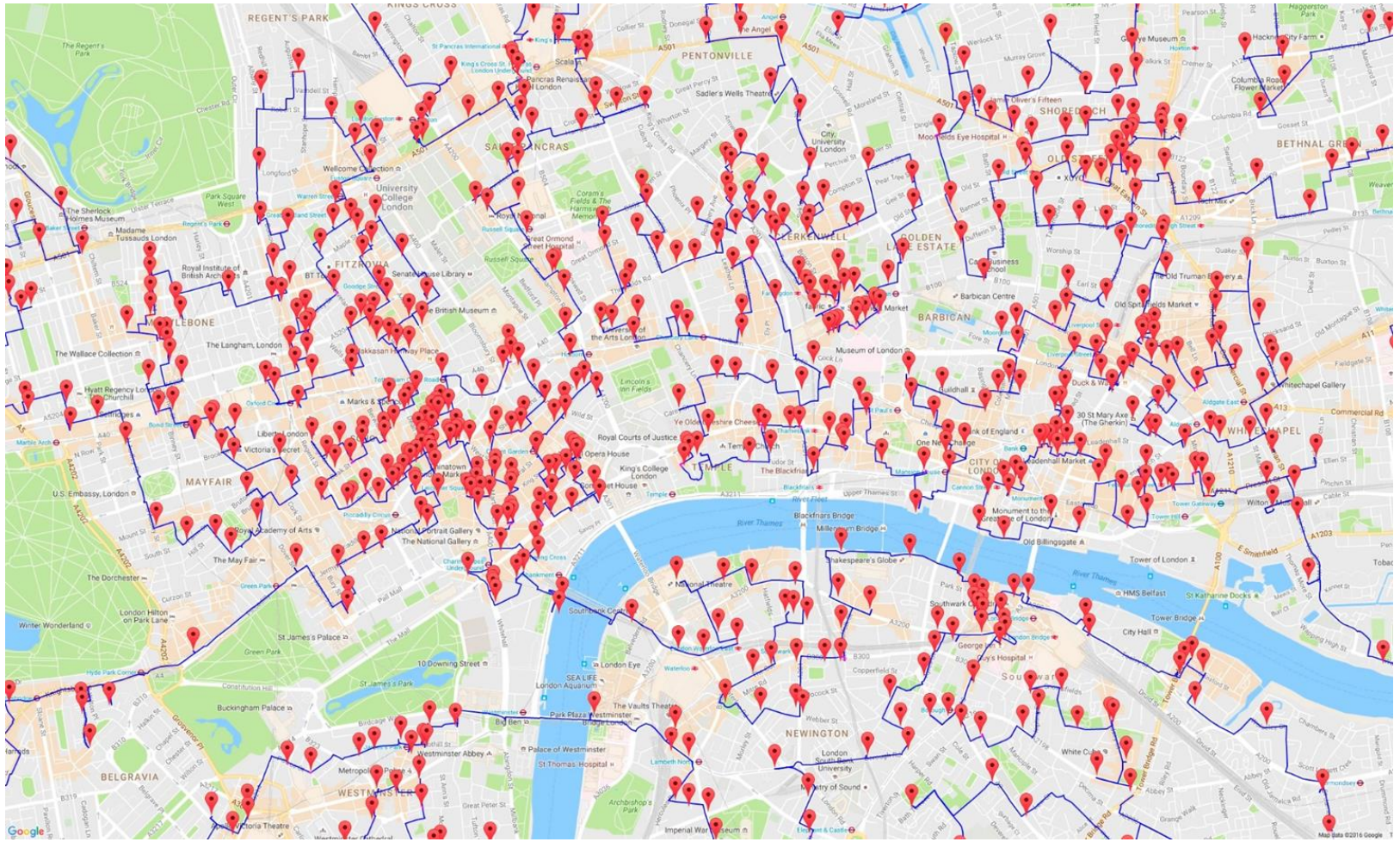
Problem-8
TRAVELING SALESMAN PROBLEM (TSP)

TRAVELING SALESMAN PROBLEM (TSP)



TRAVELING SALESMAN PROBLEM (TSP)

TRAVELING SALESMAN PROBLEM (TSP)



TRAVELING SALESMAN PROBLEM (TSP)

- A salesman has a list of cities, each of which he must visit exactly once.
- There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.
- A simple **BFS algorithm** can solve this problem.
- It would simply explore all possible paths in the tree and return the one with shortest length.
- But this approach breaks down quickly as the number of cities grows.

TRAVELING SALESMAN PROBLEM (TSP)

- If there are N cities, then the number of different paths among them is **1.2…..(N-1) or (N-1)!**
- The time to examine a single path is proportional to N. So the total time required to perform this search is proportional to N.
- Assuming there are only **10 cities**, **10! = 36,28,800**, which is a very large number.
- The salesman have easily have **25 cities** to visit. To solve this problem would take more time than he would be willing to spend.
- This phenomenon is called **combinatorial explosion**.
- It can be solved by **Heuristic technique**.

HEURISTIC SEARCH

- **Heuristic:** involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods. (Merriam-Webster's dictionary)
- Heuristic technique improves the efficiency of a search process, possibly by **sacrificing** claims of **completeness** or **optimality**.
- Heuristic is for combinatorial explosion.
- Optimal solutions are rarely needed.

NEAREST NEIGHBOR HEURISTIC FOR TSP

1. Select a starting city.
2. Select the one closest to the current city.
3. Repeat step 2 until all cities have been visited.

- $O(n^2)$ vs. $O(n!)$

HEURISTIC FUNCTION

- State descriptions → measures of desirability
- Simple Heuristic Functions
- **TSP:** the sum of the distances so far

HEURISTIC FUNCTION: 8-PUZZLE PROBLEM

- $h_1(n)$ =number of misplaced tiles = 6
- $h_2(n)$ =sum of the distance of every numbered tile to its goal position

  = 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 =13
- $h_3(n)$ =sum of permutation inversions

  = $n_5 + n_8 + n_4 + n_2 + n_1 + n_7 + n_3 + n_6$

  = 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0

  = 16

5		8
4	2	1
7	3	6

STATE(N)

1	2	3
4	5	6
7	8	

Goal state

Informed Search (Basic Heuristic Search Algorithm)

Limitations of Blind Uninformed Search Method

- As name suggests, the search is blind without any knowledge of domain and search proceeds with two basic information like start and goal node.
- The search methods also do not give optimal solution
- The search methods have exponential time complexity $O(b^d)$ if the state space grows exponentially and goal is tending towards the last level of state space search tree
- The behavior of search methods is decided by the domain specific implementation of GoalTest() and MoveGen() function used in the methods

Advantages of Informed / Heuristic Search Methods

- The Hill Climbing, Best First Search, A* algorithms are Informed / Heuristic search methods with following advantages.
- As name suggests, the search is Informed and not blind, using the knowledge of domain on which it works. The search proceeds with three basic information like start and goal node, and heuristic information / function, also called evaluation function
- The search methods may give optimal solution if domain specific Heuristic function is defined properly.

Advantages of Informed / Heuristic Search Methods

- The speed of search can be made faster by incorporating domain knowledge in heuristic / evaluation function.
- Space utilization is linear as search algorithm is guided by some domain specific knowledge
- The Heuristic or Evaluation Function is defined based on some domain specific knowledge so that it will lead the search algorithm to goal state

Advantages of Informed / Heuristic Search Methods

- Heuristic function must not be computationally expensive so that the overall cost of search would be less and this can easily be done by heuristic function as a static evaluation function.
- The heuristic function is represented as h(n), where ‘n’ is current node in question
- The heuristic value is implicitly evaluated with respect to goal and it is included with every node
- The heuristic value of Goal node is zero and all other nodes have non-zero heuristic value
- The problem on graph like TSP where shortest path and low cost solution is prime requirement, the heuristic function can be found using Euclidian distance OR Manhattan distance.

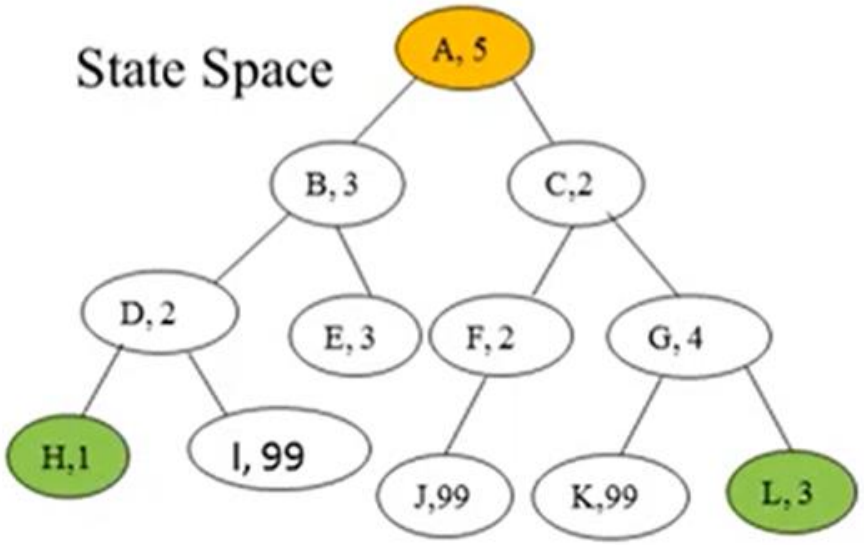
State Space Representation – Heuristic Search

Example: State Space Representation

Let A,B,C,D, …… represents a state in a solution space. The following moves are legal.
A5 to B3 and C2
B3 to D2 and E3
C2 to F2 and G4
D2 to H1 and I99
G4 to K99 and L3

Start = {A}
Goal = {H and L}

Note: The numeric value after a character represents its heuristic value.
A5 → A is node and 5 is its heuristic value.



S ={
(A,5):[(B,3),(C,2)],
(B,3):[(D,2),(E,3)],
(C,2):[(F,2),(G,4)],
(D,2):[(H,1),(I,99)],
(G,4):[(K,99),(L,3)]
}

Heuristic Search

- **Heuristic:** involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods. (Merriam-Webster's dictionary)
- Heuristic technique improves the efficiency of a search process, possibly by **sacrificing** claims of **completeness** or **optimality**.
- Heuristic is for combinatorial explosion.
- Optimal solutions are rarely needed.

Nearest Neighbor Heuristic for TSP

1. Select a starting city.
2. Select the one closest to the current city.
3. Repeat step 2 until all cities have been visited.

$O(n^2)$ vs. $O(n!)$

Heuristic Function: 8-puzzle problem

$h_1(n)$ =number of misplaced tiles = 6

$h_2(n)$ =sum of the distance of every numbered tile to its goal position

= 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 =13

$h_3(n)$ =sum of permutation inversions

$= n_5 + n_8 + n_4 + n_2 + n_1 + n_7 + n_3 + n_6$

= 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0

= 16

Basic Heuristic Search Algorithm (Simple Hill Climbing)

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
- **If** it is goal state, then return success and quit.
- **Else if** it is better than the current state then assign new state as a current state.
- **Else** if not better than the current state, then return to step2.
- **Step 5:** Exit.

Basic Heuristic Search Algorithm (Simple Hill Climbing)

```
state ← Initial State
value ← f(state)
while not goal (state):
    M ← applicable Rules (state)
    for each  m ε M
        nextState ← applyRule (m, state)
        if f(nextState) > value
            value ← f(nextState )
            r ← m
    state ← applyRule(r, state)
```

Heuristic Function: Block World Problem

- The block world problem comprises of some fix number of blocks with the start and goal configuration. The blocks either placed on infinitely large table or can be put on another but only one block over the other.

- Given the start configuration, one has to find the sequence of moves to get correct goal configuration. This is a classic example to be solved using some heuristic methods.

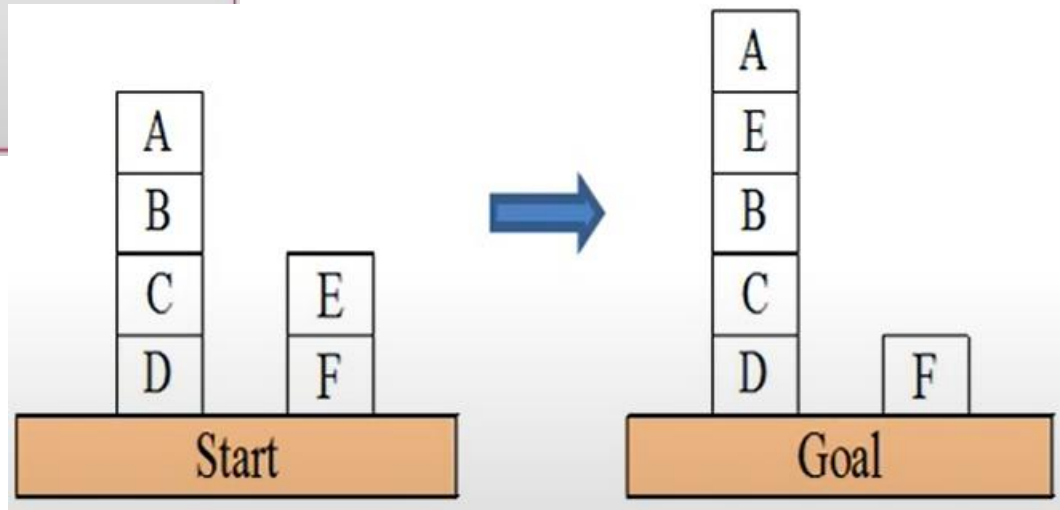
Heuristic Function: Block World Problem

Heuristic Method-1:

It is define as $h_1(n)$ with respect to the Goal configuration:

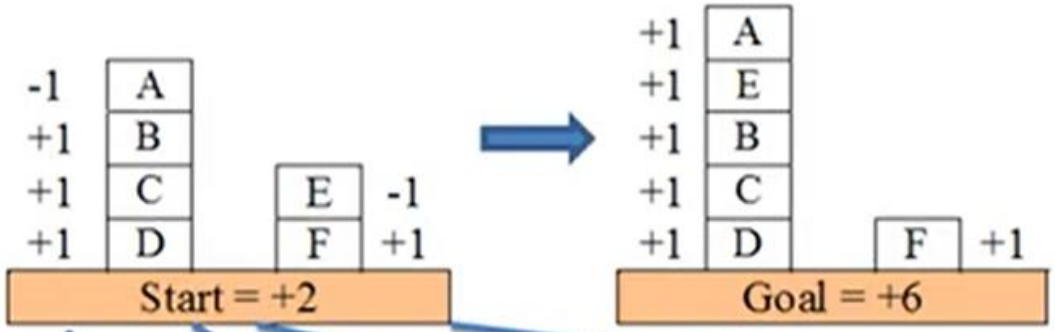
1. If a given block is on the correct block or place then assign weight +1 to the block.
2. If a given block is not on the correct block or place then assign weight -1 to the block.

$h_1(n)$ = val-block1 + val-block2+val-block3 + ----------



Heuristic Function: Block World Problem

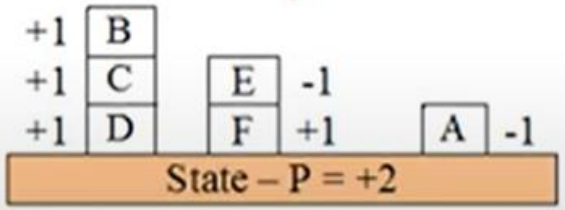
For the given block world problem the $h_1(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

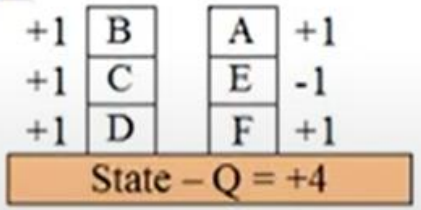


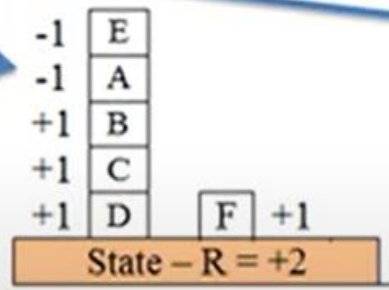
$h_1(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

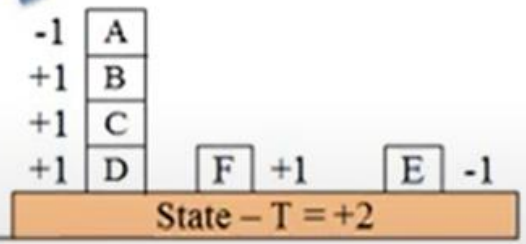
$h_1(Start) = (-1) + 1 + 1 + 1 + (-1) + 1 = +2$

$h_1(Goal) = 1 + 1 + 1 + 1 + 1 + 1 = +6$









$h_1(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_1(P) = (-1) + 1 + 1 + 1 + (-1) + 1 = +2$

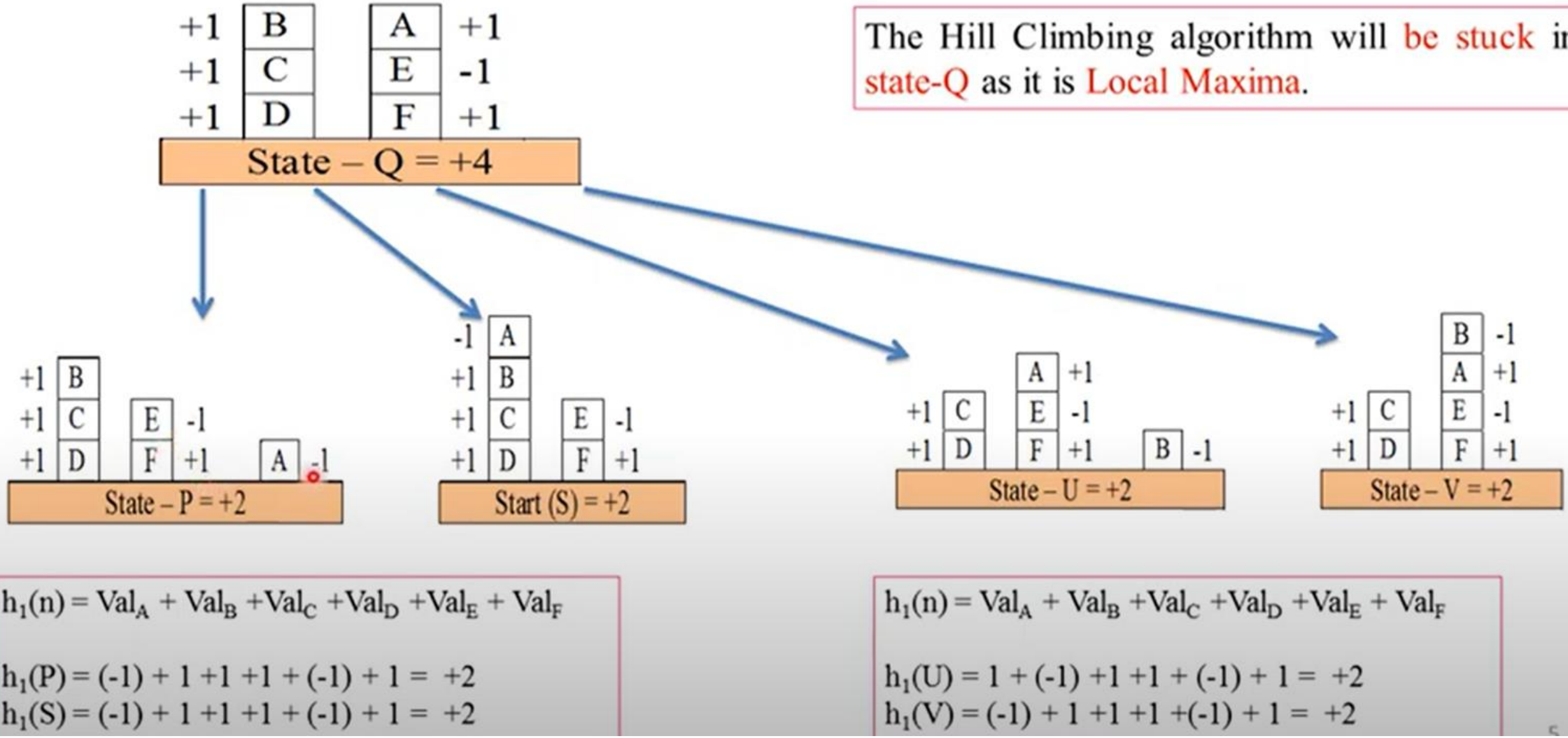
$h_1(Q) = 1 + 1 + 1 + 1 + (-1) + 1 = +4$

$h_1(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_1(R) = (-1) + 1 + 1 + 1 + (-1) + 1 = +2$

$h_1(T) = (-1) + 1 + 1 + 1 + (-1) + 1 = +2$

Heuristic Function: Block World Problem



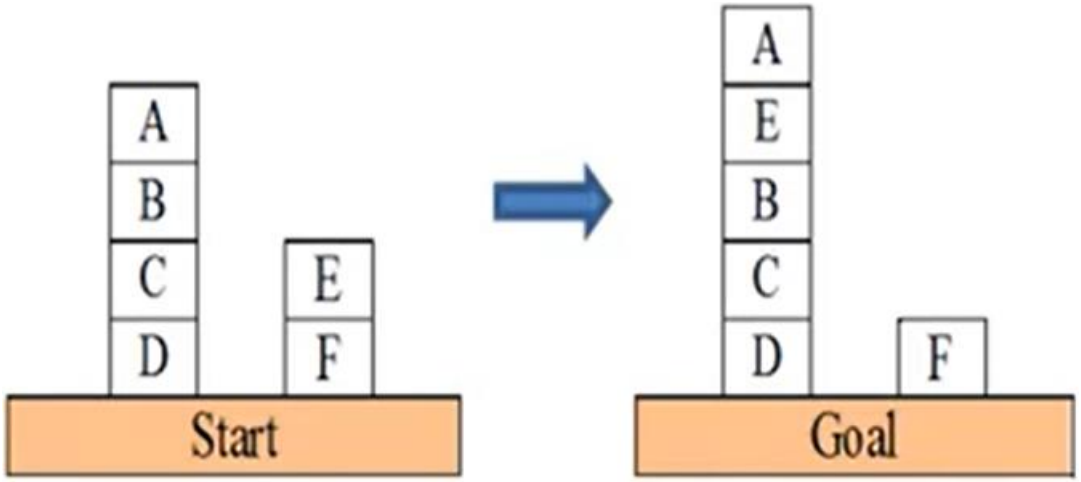
Heuristic Function: Block World Problem

Heuristic Method-2:

It is define as $h_2(n)$ with respect to the Goal configuration:

1. If a given block is on the correct block or place then assign +ve weight to the block.
2. If a given block is not on the correct block or place then assign -ve weight to the block.

$h_2(n)$ = val-block1 + val-block2+val-block3 + ----------



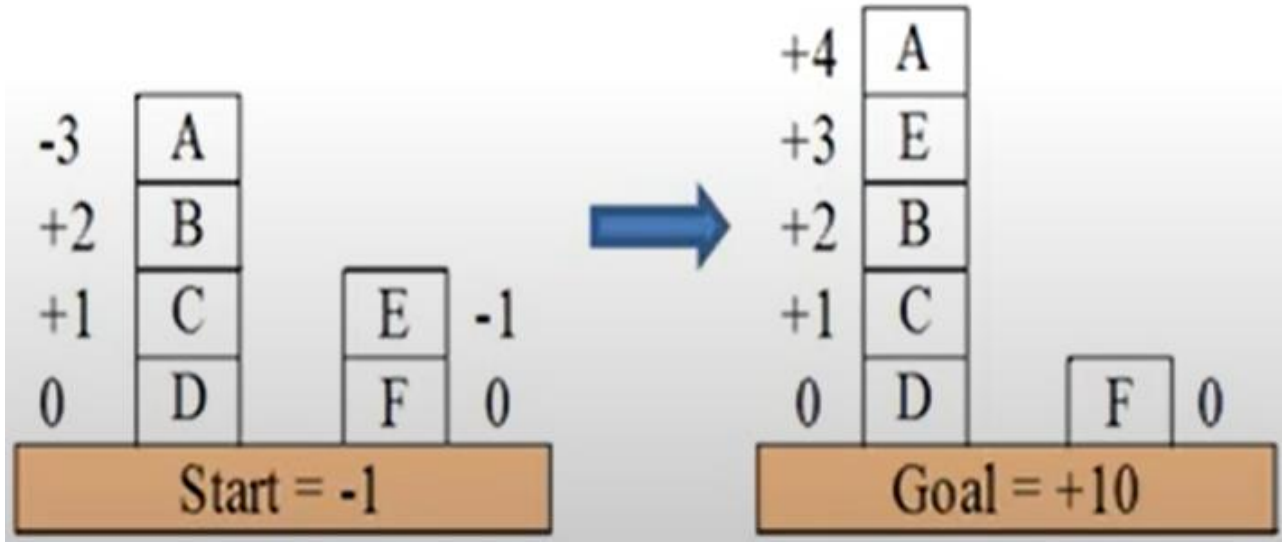
Heuristic Function: Block World Problem

For given Block World problem:

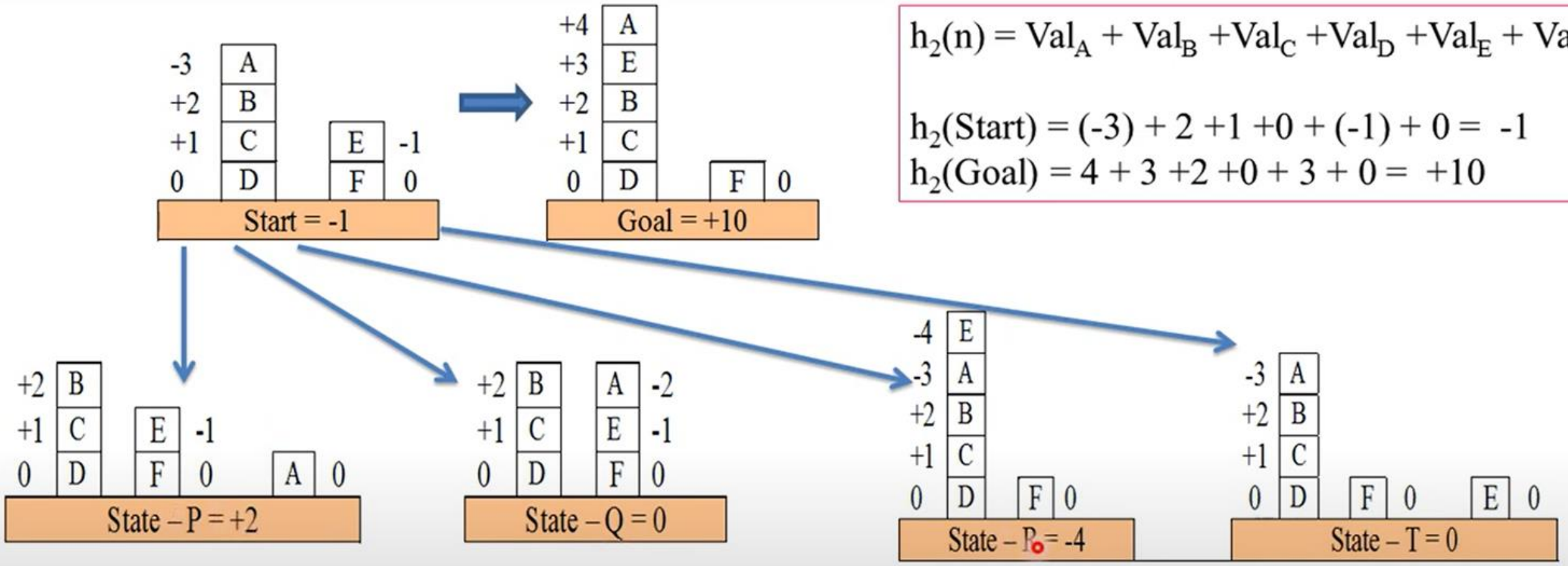
$h_2(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_2(Start) = (-3) + 2 + 1 + 0 + (-1) + 0 = -1$

$h_2(Goal) = 4 + 3 + 2 + 0 + 3 + 0 = +10$



Heuristic Function: Block World Problem



$h_2(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_2(Start) = (-3) + 2 + 1 + 0 + (-1) + 0 = -1$
$h_2(Goal) = 4 + 3 + 2 + 0 + 3 + 0 = +10$

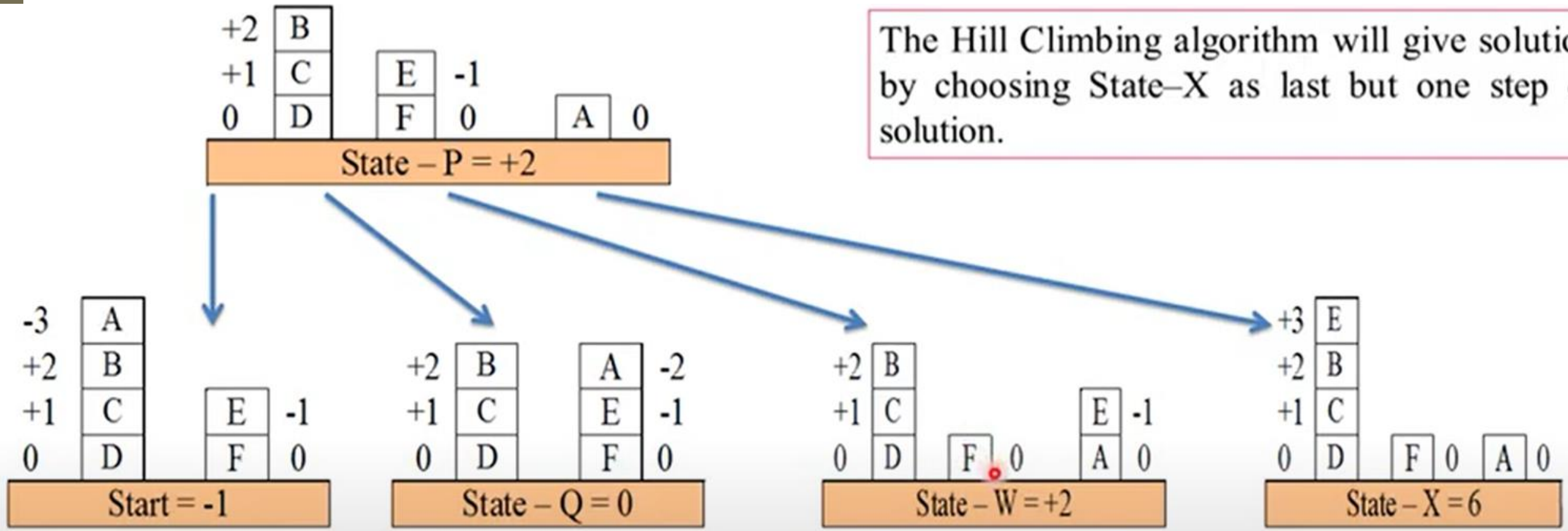
$h_2(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_2(P) = 0 + 2 + 1 + 0 + (-1) + 0 = +2$
$h_2(Q) = (-2) + 2 + 1 + 0 + (-1) + 0 = 0$

$h_2(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_2(R) = (-3) + 2 + 1 + 0 + (-4) + 0 = -4$
$h_2(T) = (-3) + 2 + 1 + 0 + 0 + 0 = 0$

Heuristic Function: Block World Problem



The Hill Climbing algorithm will give solution by choosing State–X as last but one step of solution.

$h_2(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_2(Start) = (-3) + 2 + 1 + 0 + (-1) + 0 = -1$
$h_2(Q) = (-2) + 2 + 1 + 0 + (-1) + 0 = 0$

$h_2(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_2(W) = 0 + 2 + 1 + 0 + (-1) + 0 = +2$
$h_2(X) = 0 + 2 + 1 + 0 + 3 + 0 = +6$

Heuristic Function: Block World Problem

The choice correct ***Heuristic Function*** plays a vital role while using search methods. In case of Block World domain , function $h_2(n)$ is more discriminating as compare to $h_1(n)$ and it looks the entire pile that the block is resting on.

$h_1(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

$h_1(Start) = (-1) + 1 + 1 + 1 + (-1) + 1 = +2$
$h_1(Goal) = 1 + 1 + 1 + 1 + 1 + 1 = +6$
$h_1(P) = (-1) + 1 + 1 + 1 + (-1) + 1 = +2$
$h_1(Q) = 1 + 1 + 1 + 1 + (-1) + 1 = +4$
$h_1(R) = (-1) + 1 + 1 + 1 + (-1) + 1 = +2$
$h_1(T) = (-1) + 1 + 1 + 1 + (-1) + 1 = +2$

$h_2(n) = Val_A + Val_B + Val_C + Val_D + Val_E + Val_F$

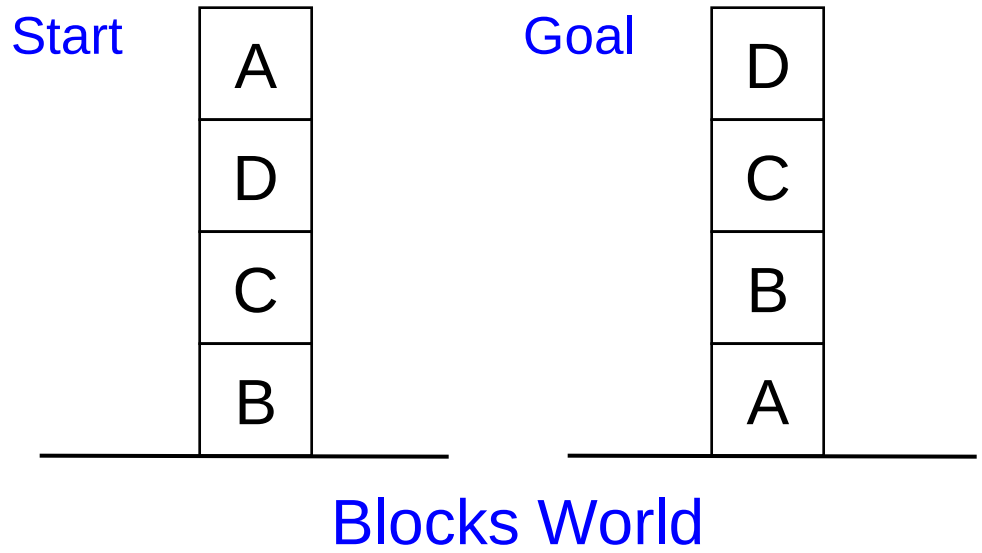
$h_2(Start) = (-3) + 2 + 1 + 0 + (-1) + 0 = -1$
$h_2(Goal) = 4 + 3 + 2 + 0 + 3 + 0 = +10$
$h_2(P) = 0 + 2 + 1 + 0 + (-1) + 0 = +2$
$h_2(Q) = (-2) + 2 + 1 + 0 + (-1) + 0 = 0$
$h_2(R) = (-3) + 2 + 1 + 0 + (-4) + 0 = -4$
$h_2(T) = (-3) + 2 + 1 + 0 + 0 + 0 = 0$

The function $h_1(n)$ had a little choice and it led the search to local maxima.

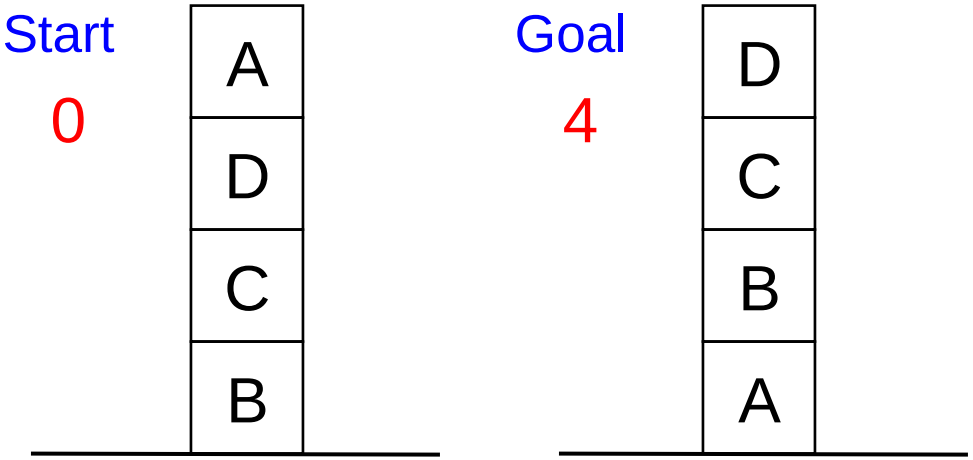
The function $h_2(n)$ had selected proper sequence of steps and led to the goal state.

Note: If the goal is of **maximization** then the Search algorithm may get stuck in **local maxima** due to ***steepest gradient ascent*** and if the goal is **minimization** then it may get stuck in **local minima** due to ***steepest gradient decent***.

HEURISTIC Function: BLOCK WORLD PROBLEM



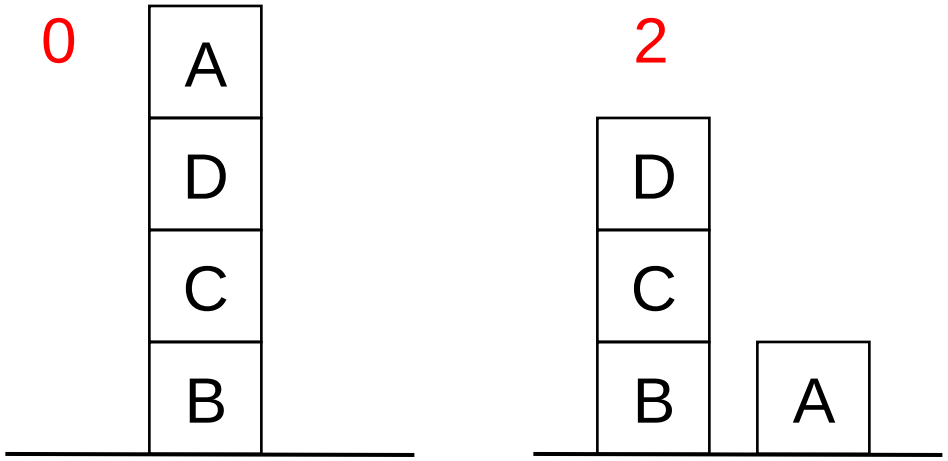
HEURISTIC Function: BLOCK WORLD PROBLEM



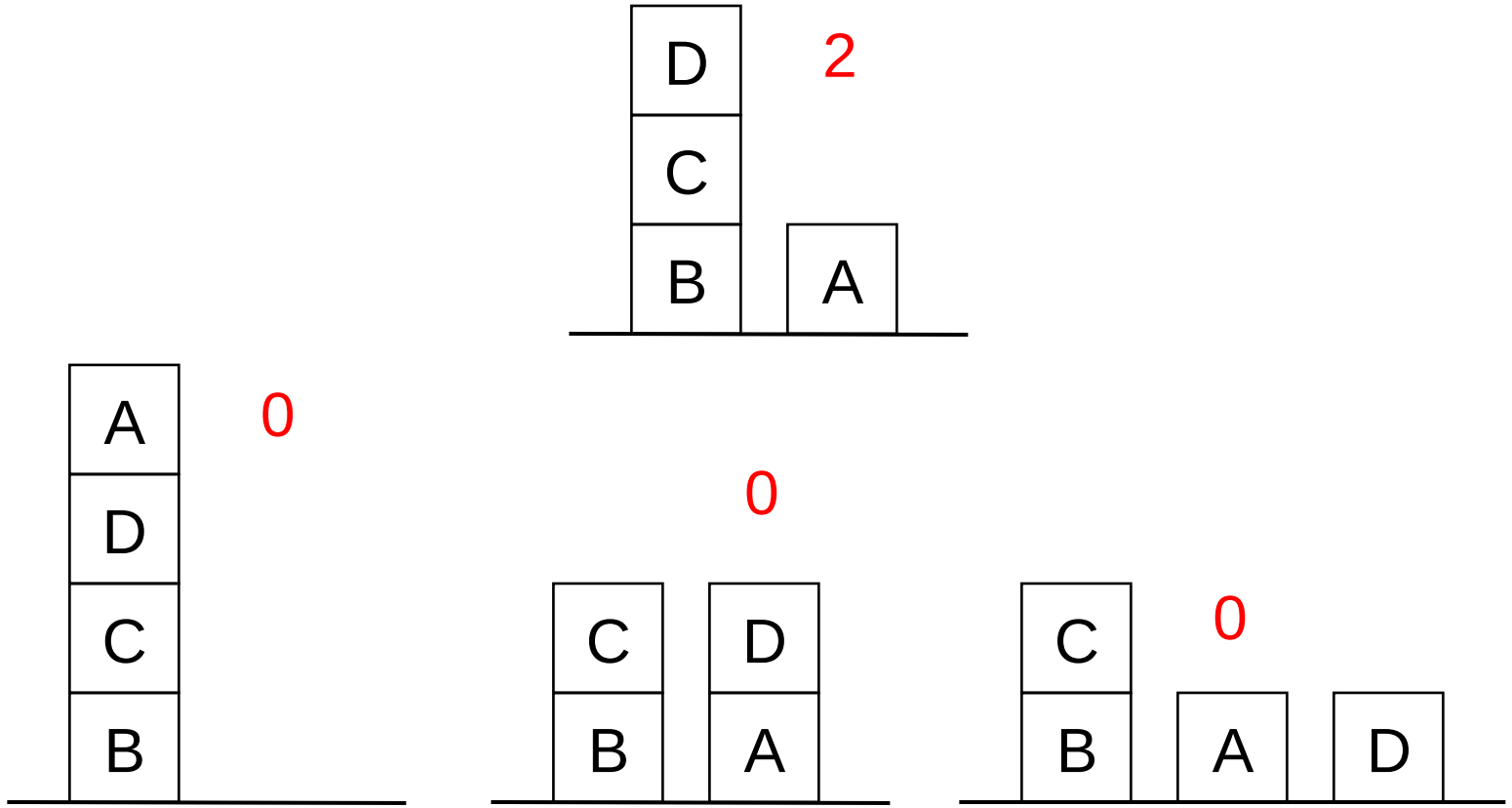
Blocks World

Local heuristic:
+1 for each block that is resting on the thing it is supposed to be resting on.
-1 for each block that is resting on a wrong thing.

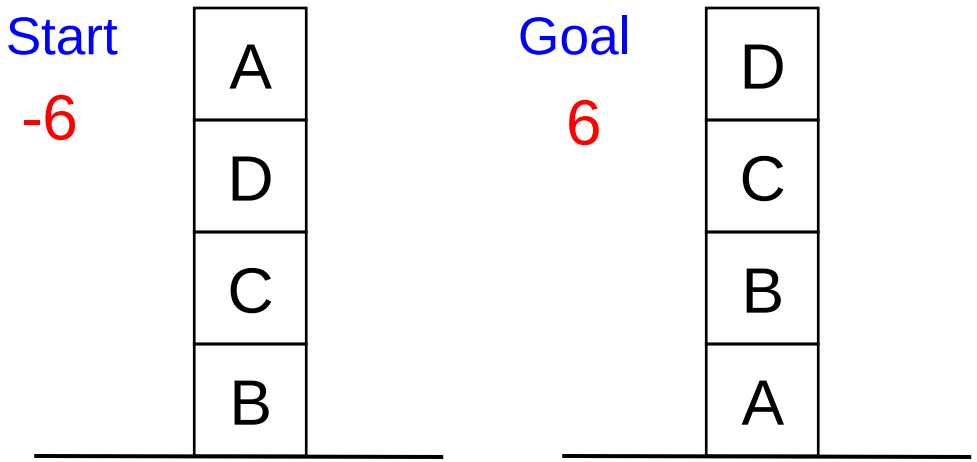
HEURISTIC Function: BLOCK WORLD PROBLEM



HEURISTIC Function: BLOCK WORLD PROBLEM



HEURISTIC Function: BLOCK WORLD PROBLEM



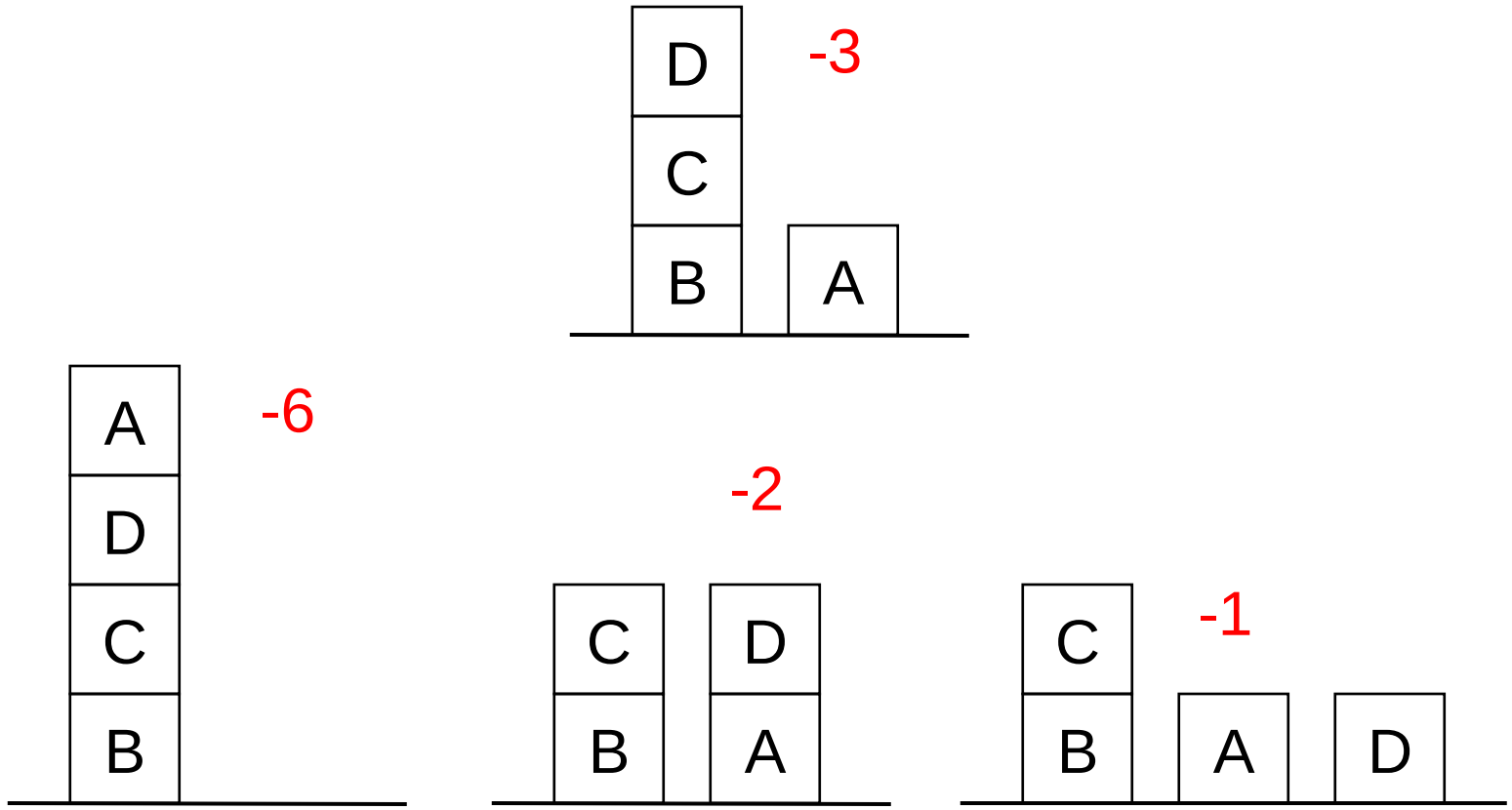
Blocks World

Global heuristic:

For each block that has the correct support structure: +1 to every block in the support structure.

For each block that has a wrong support structure: -1 to every block in the support structure.

HEURISTIC Function: BLOCK WORLD PROBLEM



Heuristic Function

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

Heuristic Function

- The admissibility of the heuristic function is given as:
- $h(n) \leq h^*(n)$
- Here, $h(n)$ is heuristic cost and $h^*(n)$ is estimated cost. Hence, Heuristic cost should be less than or equal to the estimated cost.

Best First Search Algorithm

Advantages of BFS & DFS

- Advantages of BFS
  - not getting trapped on dead-end paths
- Advantages of DFS
  - not all competing branches having to be expanded

Best First Search(Greedy Search)

- Best First Search is combination of Depth First Search Breadth First Search algorithms.
- Combining the two is to follow a single path at a time, but switch paths whenever some competing path look more promising than the current one.
- In Best First Search Algorithm, it expands node which is closest to the goal node and closest cost is estimated by heuristic function,
- Evaluation function $f(n) = h(n)$
- $Where, h(n) = estimated\ cost\ from\ node\ n\ to\ goal\ node$
- This algorithm can be implemented by priority Queue.

OPEN List and CLOSED List

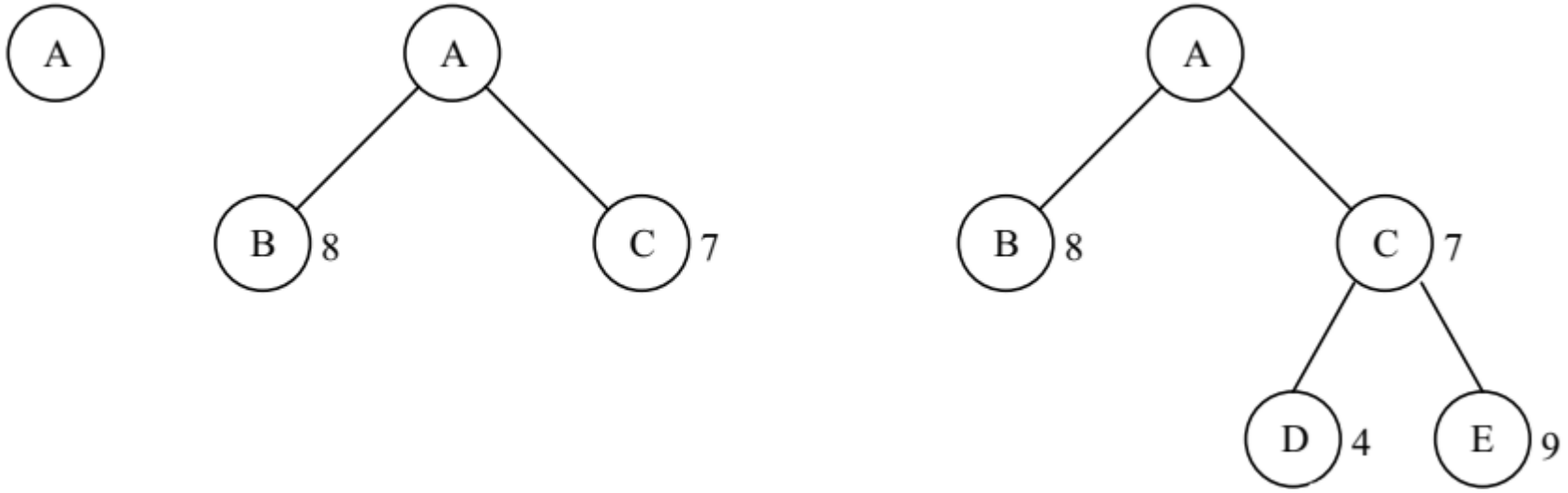
- **OPEN:** nodes that have been generated, but have not explored. Open list can be implemented in the form of a queue in which the nodes will be arranged in the order of decreasing priority from the front i.e., the node with the most promising heuristic value (i.e., the highest priority node) will be at the first place in the list..
- This is organized as a **priority queue**.
- **CLOSED:** nodes that have already been explored.
- Whenever a new node is generated, check whether it has been generated before.

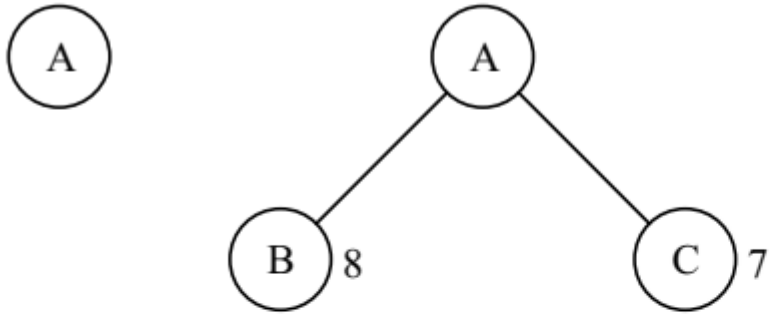
Best First Search Algorithm

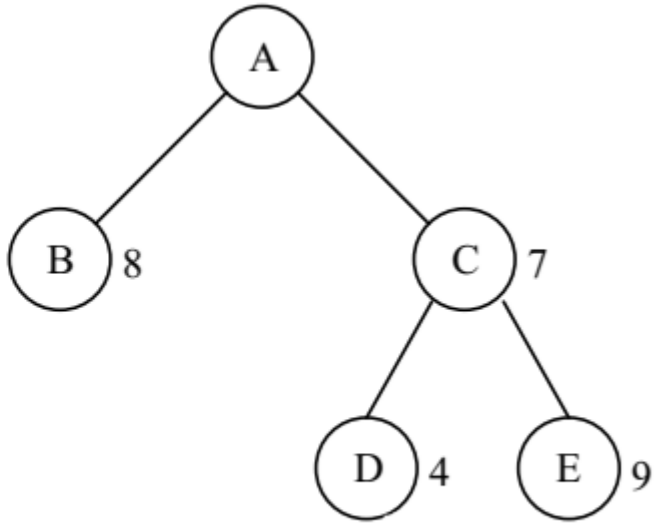
1. Push Start node on **OPEN**
2. **CLOSED** = []
3. If **OPEN** is Empty then
    1. Return search ends unsuccessfully
4. CURRENT_NODE = Pop from **OPEN**
5. Push CURRENT_NODE in **CLOSED**
6. If CURRENT_NODE == GOAL_NODE
    1. Return search ends successfully
7. N = Generate All Successor of CURRENT_NODE
8. For every successor n' on N:
    1. Calculate f (n') by equation $f(n') = h(n')$
    2. If n' was neither on OPEN nor on CLOSED, add it to OPEN. Attach a pointer from n' back to N. Assign the newly computed f(n') to node n'.
    3. else If n' already resided on OPEN or CLOSED, compare the newly computed f(n) with the value previously assigned to n'. If the old value is lower, discard the newly generated node. If the new value is lower, substitute it for the old (n' now points back to N instead of to its previous predecessor).
    4. Sort All nodes of OPEN
9. Go to Step 3

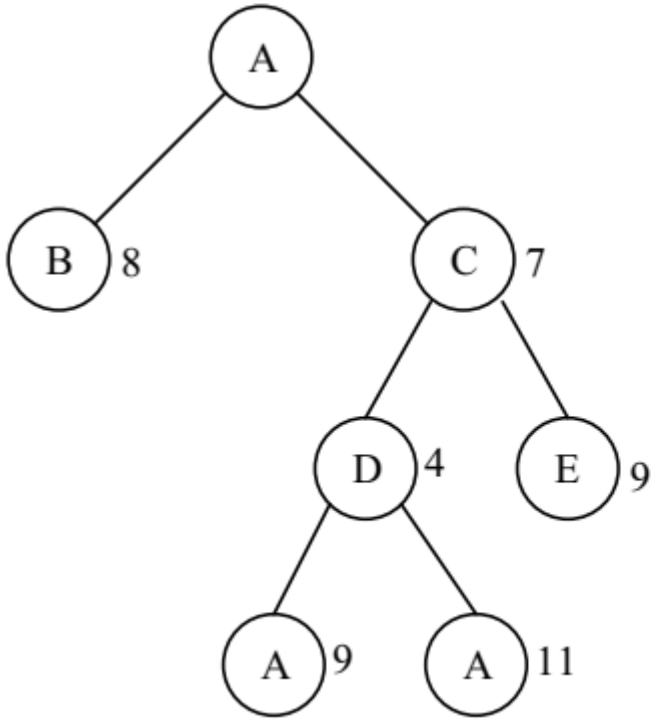
Example, Best First Search Algorithm

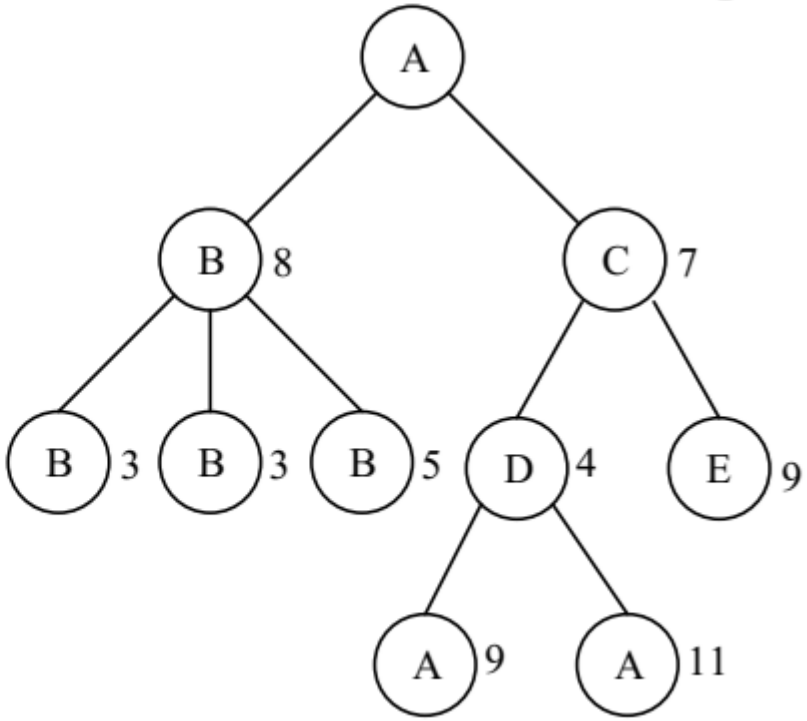
- In this example, each node has a heuristic value showing the estimated cost of getting to a solution from this node. This example shows part of the search process using best first search.



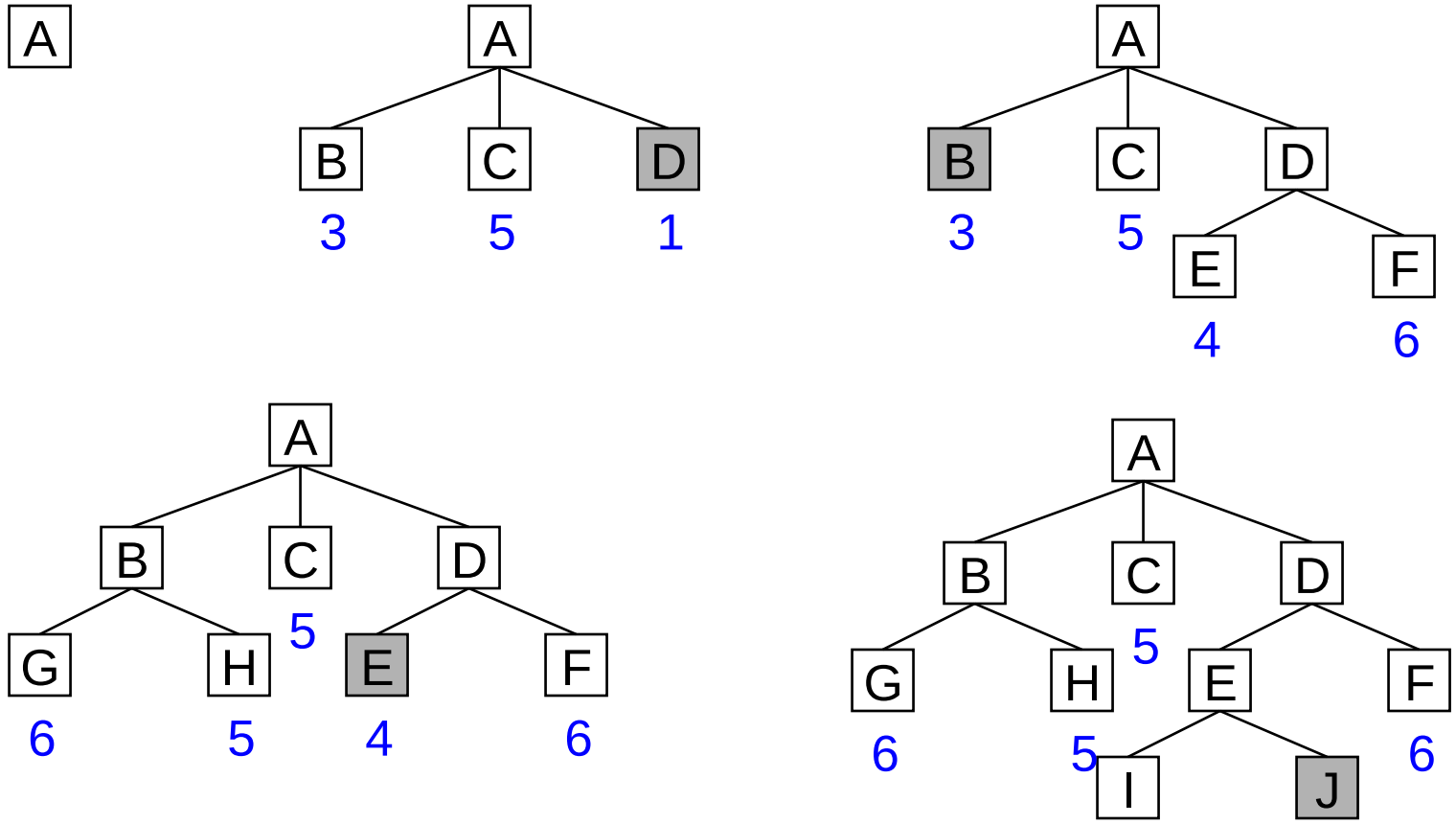

A
A
B 8
C 7


A
B 8
C 7
D 4
E 9


A
B 8
C 7
D 4
E 9
A 9
A 11


A
B 8
C 7
B 3
B 3
B 5
D 4
E 9
A 9
A 11

Example, Best First Search Algorithm



Evaluating Criteria of Greedy Best First Search

- Complete? – No
- Optimal? – No
- Time Complexity - $O(b^d)$ in worst case. Where d is max depth of search space
- Space Complexity - $O(b^d)$

A* Algorithm

BEST-FIRST SEARCH-A* algorithm

- **Greedy search:**
- h(n) = estimated cost of the cheapest path from node n to a goal state.
- Neither optimal nor complete

BEST-FIRST SEARCH-A* algorithm

- **Uniform-cost search:**
- g(n) = cost of the cheapest path from the initial state to node n.
- Optimal and complete, but very inefficient

A* algorithm

- Algorithm A* (Hart et al., 1968):
- $f(n) = g(n) + h(n)$

- $h(n)$ = cost of the cheapest path from node n to a goal state.
- $g(n)$ = cost of the cheapest path from the initial state to node n.

A* Algorithm

1. Push Start node on **OPEN**
2. **CLOSED** = []
3. If **OPEN** is Empty then
    1. Return search ends unsuccessfully
4. CURRENT_NODE = Pop from **OPEN**
5. Push CURRENT_NODE in **CLOSED**
6. If CURRENT_NODE == GOAL_NODE
    1. Return search ends successfully
7. N = Generate All Successor of CURRENT_NODE
8. For every successor n' on N:
    1. Calculate f (n') by equation $f(n') = g(n') + h(n')$
    2. If n' was neither on OPEN nor on CLOSED, add it to OPEN. Attach a pointer from n' back to N. Assign the newly computed f(n') to node n'.
    3. else If n' already resided on OPEN or CLOSED, compare the newly computed f(n) with the value previously assigned to n'. If the old value is lower, discard the newly generated node. If the new value is lower, substitute it for the old (n' now points back to N instead of to its previous predecessor).
    4. IF OPEN contains paths: P, Q AND P ends in node Ni && Q contains node Ni AND cost_P ≥ cost_Q THEN remove P from OPEN
    5. Sort All nodes of OPEN
9. Go to Step 3

A* Algorithm

(a) Set *SUCCESSOR* to point back to *BESTNODE*. These backwards links will make it possible to recover the path once a solution is found.

(b) Compute $g(SUCCESSOR) = g(BESTNODE)$ + the cost of getting from *BESTNODE* to *SUCCESSOR*.

(c) See if *SUCCESSOR* is the same as any node on *OPEN* (i.e., it has already been generated but not processed). If so, call that node *OLD*. Since this node already exists in the graph, we can throw *SUCCESSOR* away and add *OLD* to the list of *BESTNODE's* successors. Now we must decide whether *OLD's* parent link should be reset to point to *BESTNODE*. It should be if the path we have just found to *SUCCESSOR* is cheaper than the current best path to *OLD* (since *SUCCESSOR* and *OLD* are really the same node). So see whether it is cheaper to get to *OLD* via its current parent or to *SUCCESSOR* via *BESTNODE* by comparing their g values. If *OLD* is cheaper (or just as cheap), then we need do nothing. If *SUCCESSOR* is cheaper, then reset *OLD's* parent link to point to *BESTNODE*, record the new cheaper path in $g(OLD)$, and update f'(OLD).

(d) If *SUCCESSOR* was not on *OPEN*, see if it is on *CLOSED*. If so, call the node on *CLOSED OLD* and add *OLD* to the list of *BESTNODE's*, successors. Check to see if the new path or the old path is better just as in step *2(c)*, and set the parent link-and g and f' values appropriately. If we have just found a better path to *OLD*, we must propagate the improvement to *OLD's* successors. This is a bit tricky. *OLD* points to its successors. Each successor in turn points to its successors, and so forth, until each branch terminates with a node that either is still on *OPEN* or has no successors. So to propagate the new cost downward, do a depth-first traversal of the tree starting at *OLD*, changing each node's g value (and thus also its f' value), terminating each branch when you reach either a node with no successors or a node to which an equivalent or better path has already been found.[4] This condition is easy to check for. Each node's parent link points back to its best known parent. As we propagate down to a node, see if its parent points to the node we are coming from. If so, continue the propagation. If not, then its g value already reflects the better path of which it is part. So the propagation may stop here. But it is possible that with the new value of g being propagated downward, the path we are following may become better than the path through the current parent. So compare the two. If the path through the current parent is still better, stop the propagation. If the path we are propagating through is now better, reset the parent and continue propagation.

(e) If *SUCCESSOR* was not already on either *OPEN* or *CLOSED*, then put it on *OPEN*, and add it to the list of *BESTNODE's* successors. Compute $f'(.SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)$.

A* Algorithm

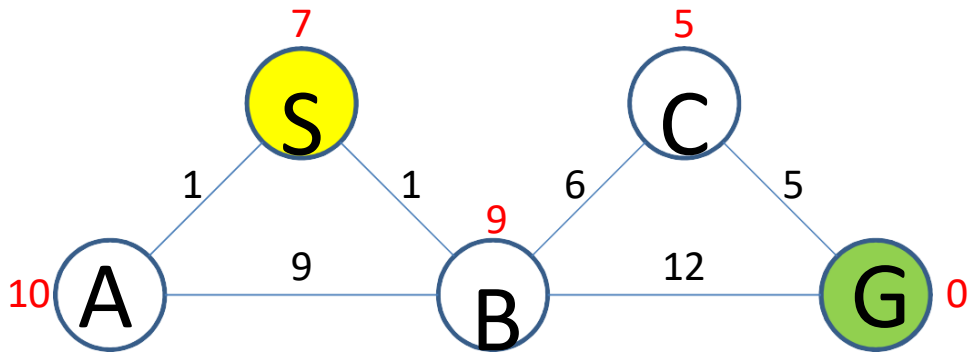
- *Input:*
  - QUEUE: Path only containing root
- *Algorithm:*
  - WHILE (QUEUE not empty && first path not reach goal) DO
    - Remove *first path* from QUEUE
    - Create paths to all children
    - Reject paths with loops
    - Add paths and sort QUEUE (by f = cost + heuristic)
    - IF QUEUE contains paths: P, Q
      AND P ends in node N_i && Q contains node N_i
      AND cost_P ≥ cost_Q
      THEN remove P
  - IF goal reached THEN success ELSE failure

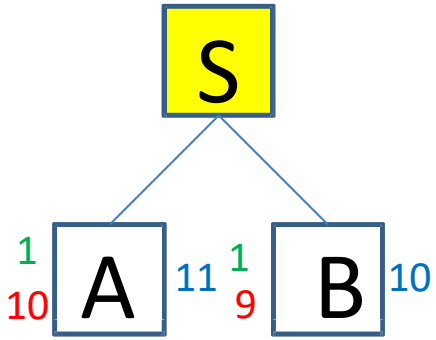
FIRST EXAMPLE ON A*

f = accumulated path cost + heuristic

QUEUE = *path containing root*

OPEN_LIST/QUEUE: <S>

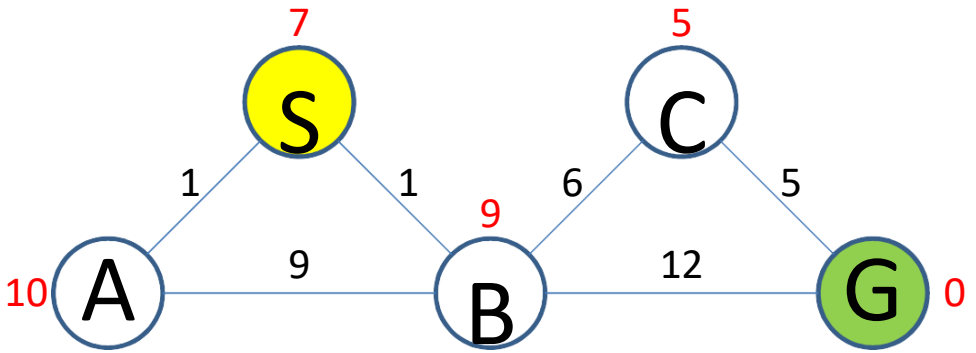


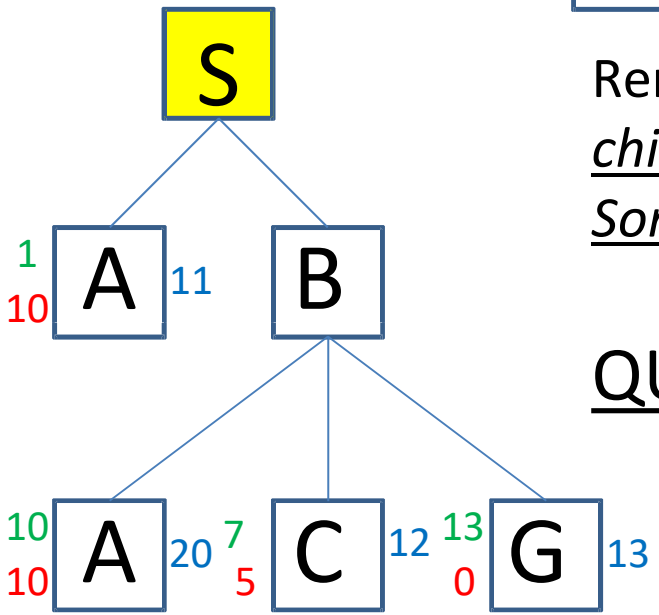


f = accumulated path cost + heuristic

Remove *first path*, Create *paths to all children*, Reject *loops* and *Add paths*.
Sort QUEUE *by f*

QUEUE: <SB,SA>

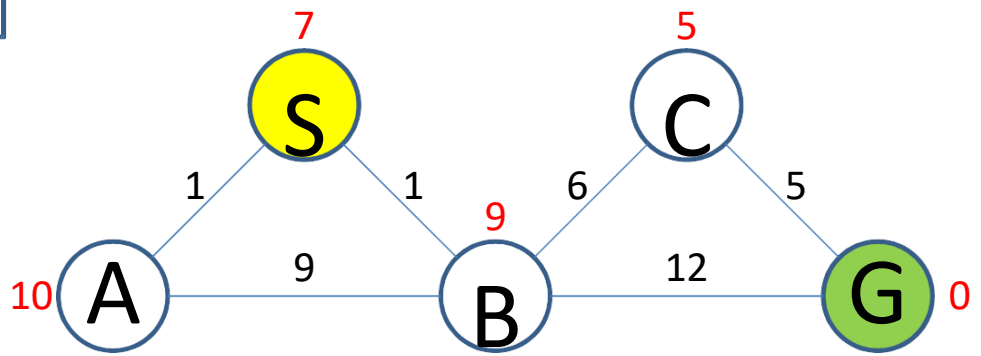




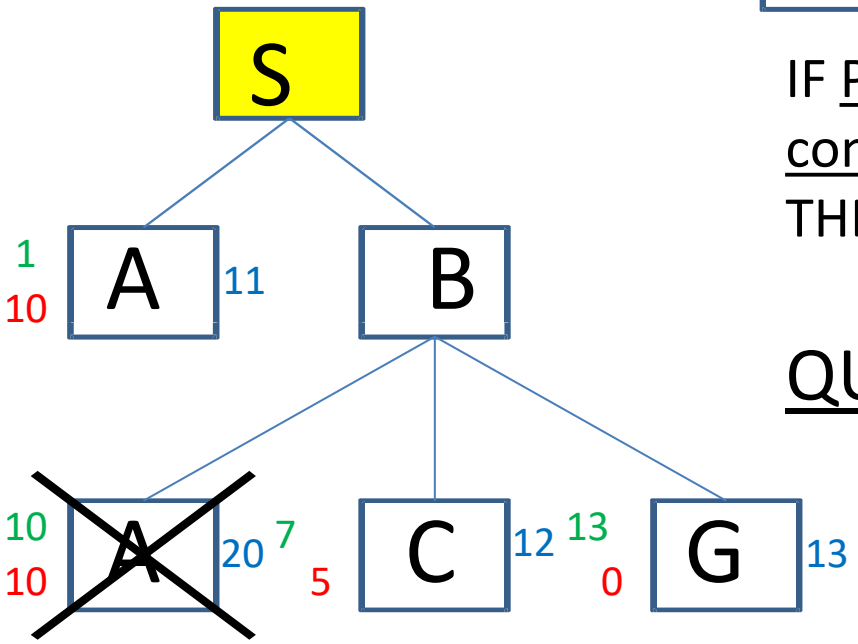
f = accumulated path cost + heuristic

Remove *first path*, Create *paths to all children*, Reject *loops* and *Add paths*.
Sort QUEUE *by f*

QUEUE: <SA,SBC,SBG,SBA>



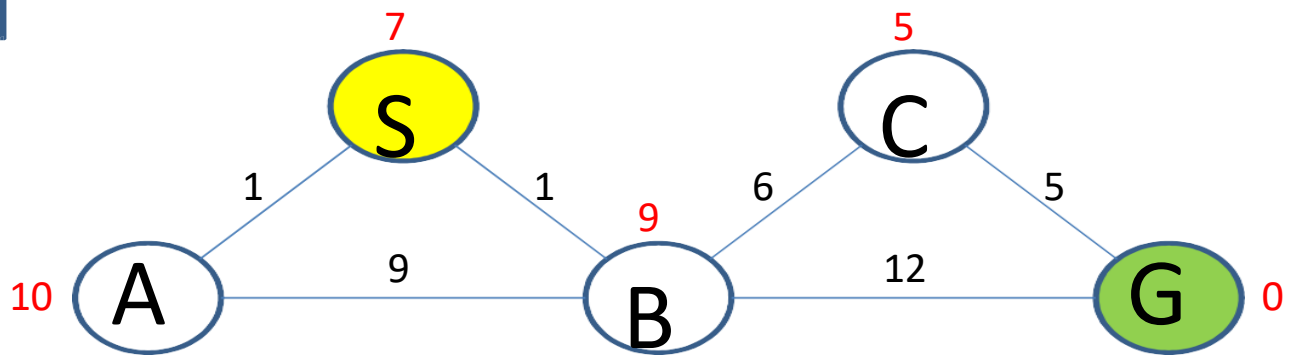
A* ALGORITHM BY EXAMPLE



f = accumulated path cost + heuristic

IF P terminating in I with cost_P && Q containing I with cost_QAND cost_P ≥ cost_Q THEN remove P

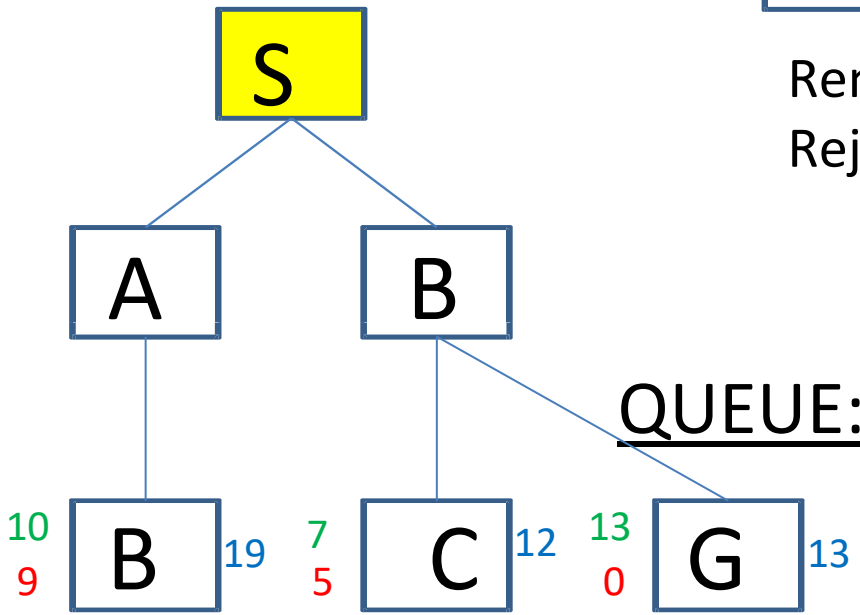
QUEUE: <SA,SBC,SBG,SBA>



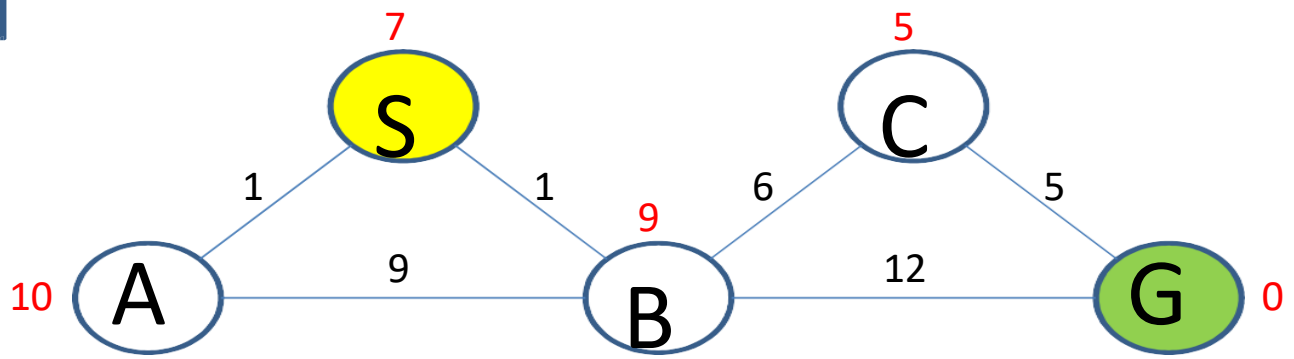
A* ALGORITHM BY EXAMPLE

f = accumulated path cost + heuristic

Remove *first path*, Create *paths to all children*,
Reject *loops* and *Add paths*. *Sort* QUEUE *by f*



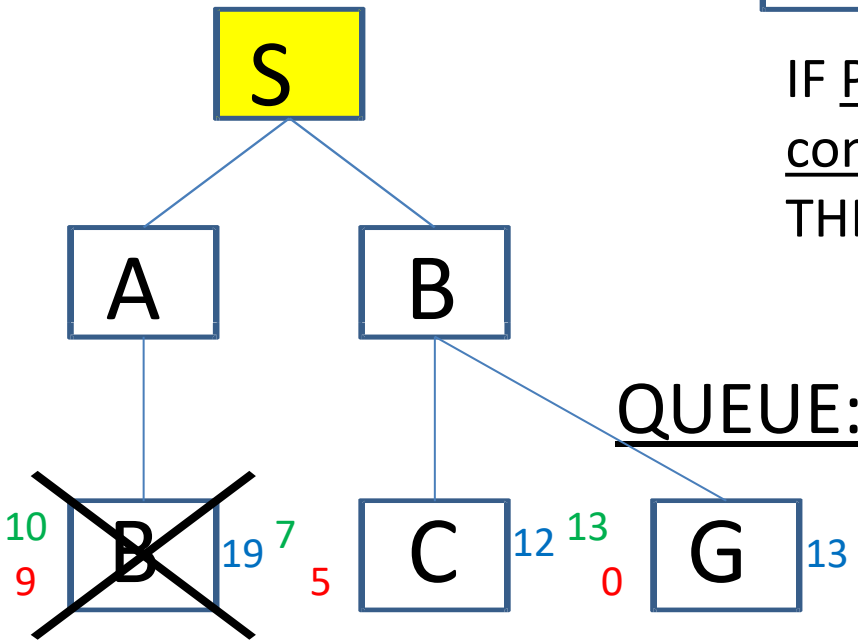
QUEUE: <SBC,SBG,SAB>



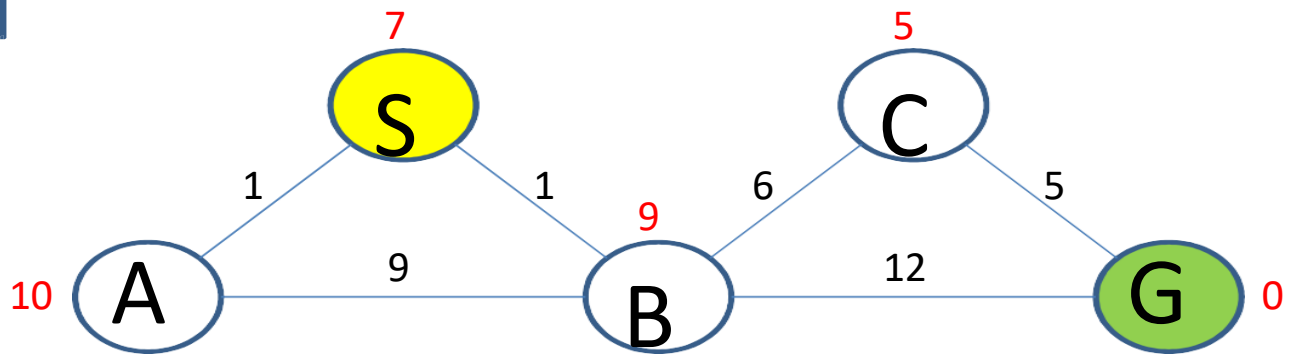
A* ALGORITHM BY EXAMPLE

f = accumulated path cost + heuristic

IF P terminating in I with cost_P && Q containing I with cost_QAND cost_P ≥ cost_Q THEN remove P



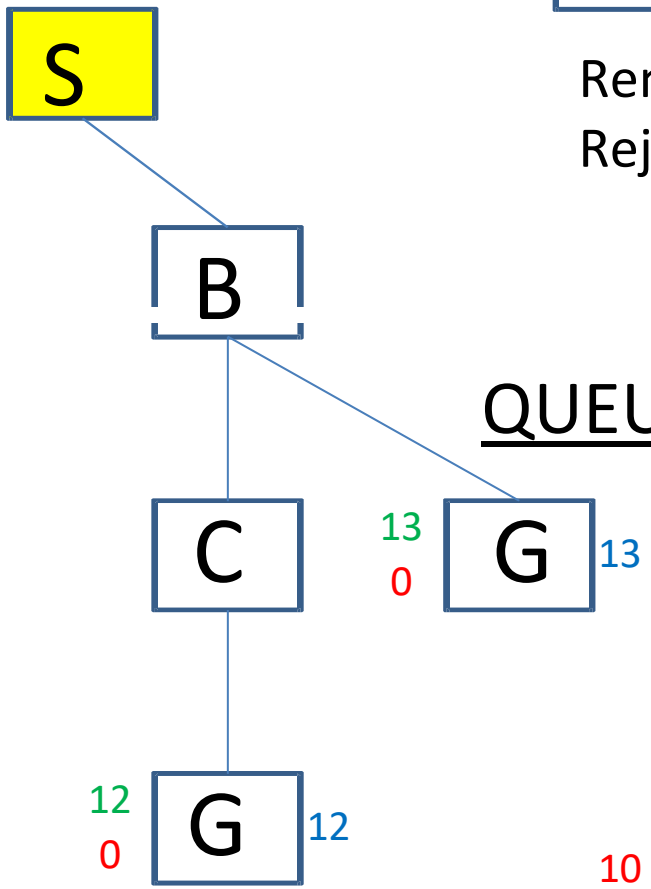
QUEUE: <SBC,SBG,SAB>



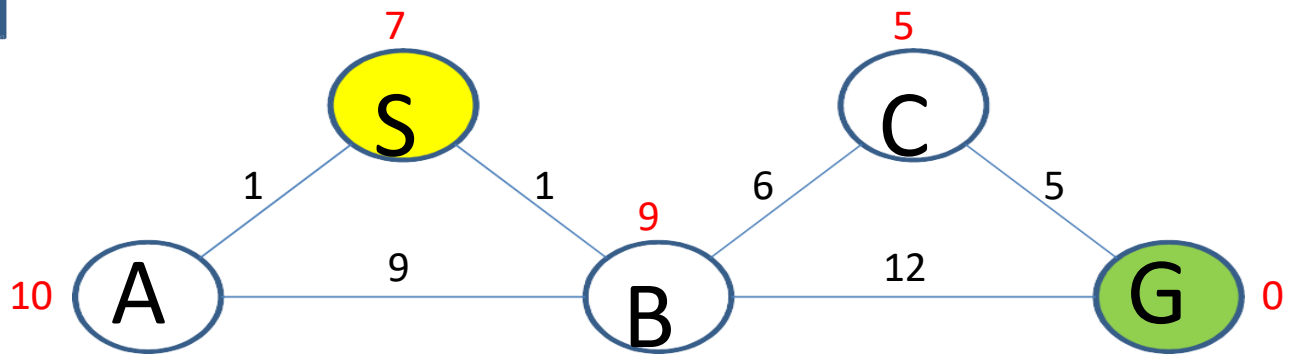
A* ALGORITHM BY EXAMPLE

f = accumulated path cost + heuristic

Remove *first path*, Create *paths to all children*,
Reject *loops* and *Add paths*. *Sort* QUEUE *by f*



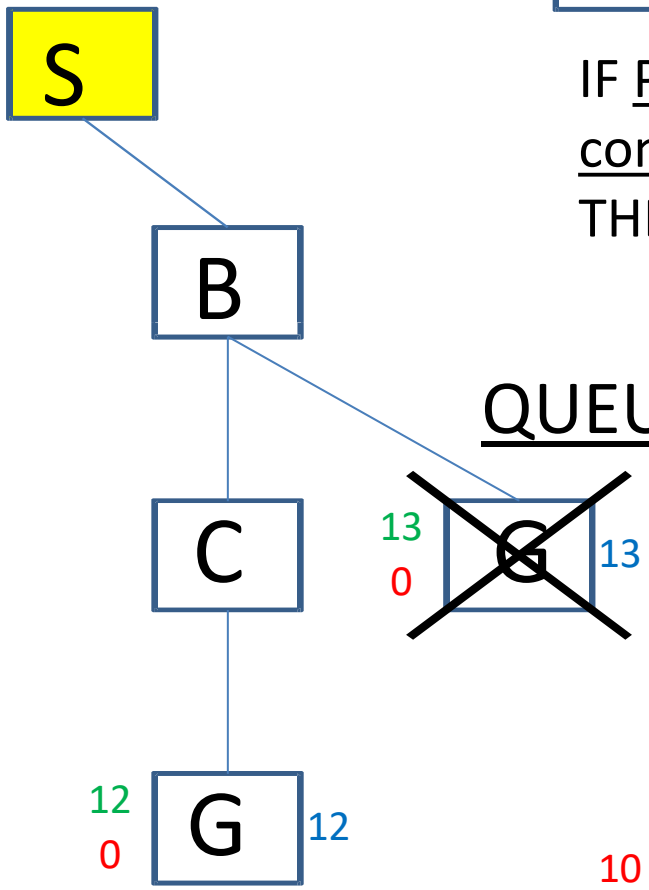
QUEUE: <SBCG,SBG>



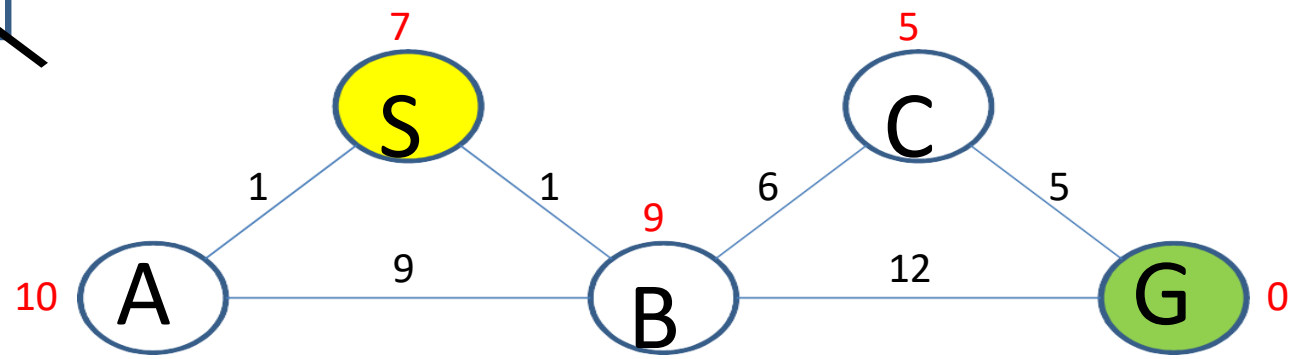
A* ALGORITHM BY EXAMPLE

f = accumulated path cost + heuristic

IF P terminating in I with cost_P && Q containing I with cost_QAND cost_P ≥ cost_Q THEN remove P



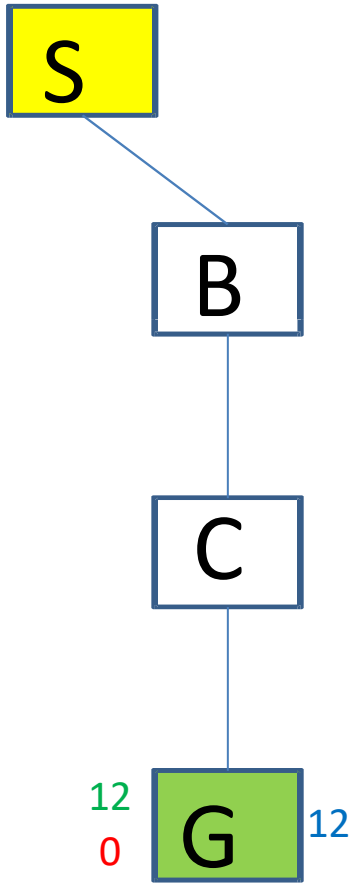
QUEUE: <SBCG,SBG>



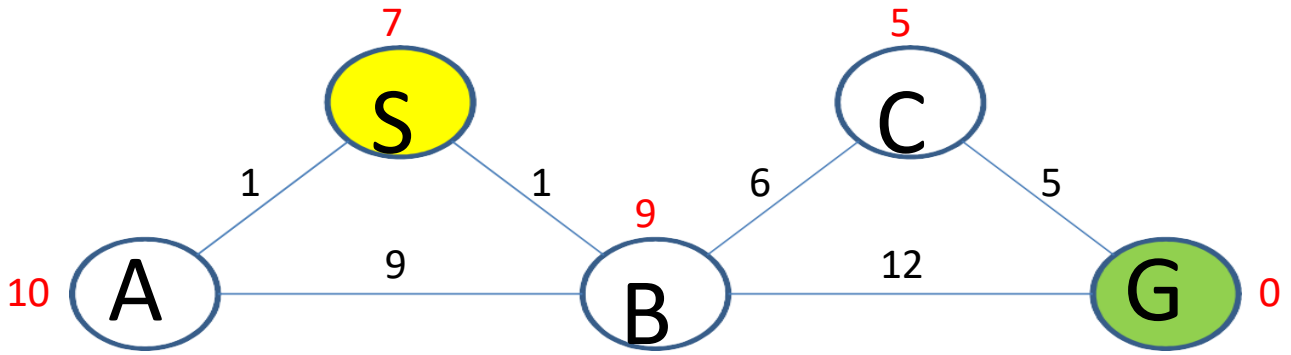
A* ALGORITHM BY EXAMPLE

f = accumulated path cost + heuristic

SUCCESS

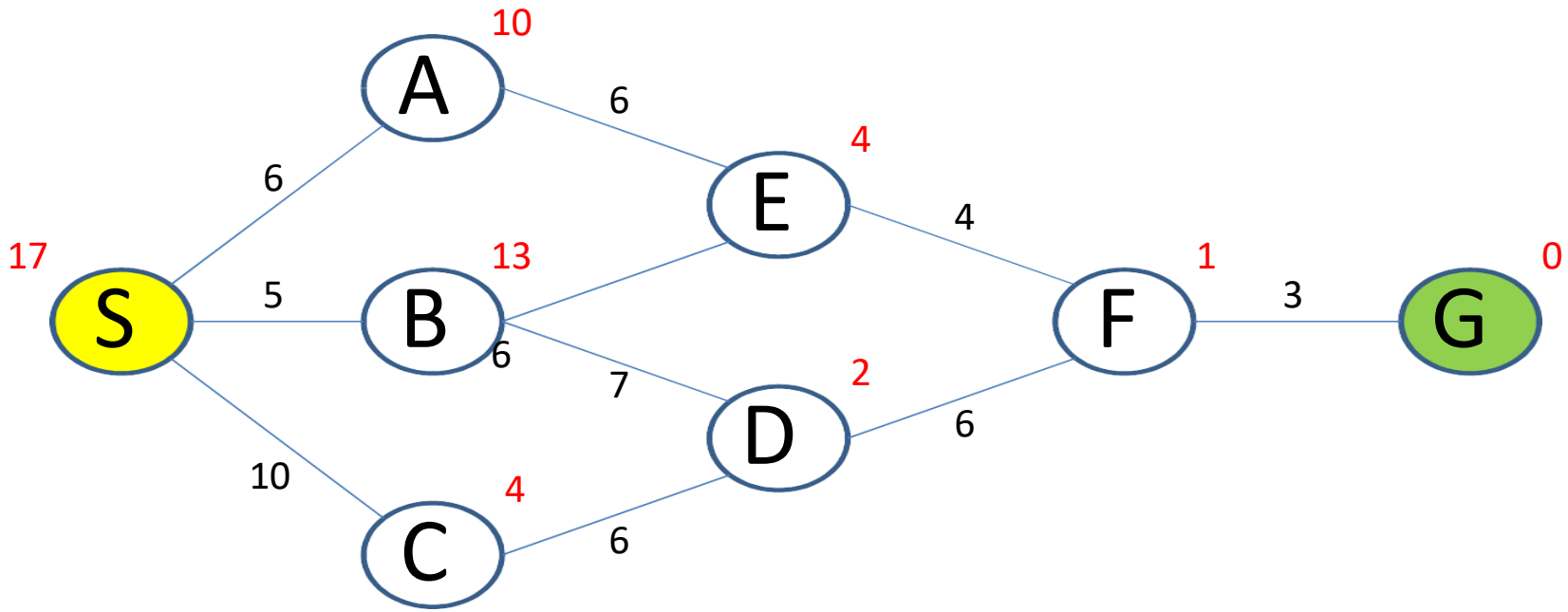


QUEUE: <SBCG>



PROBLEM

- Perform the A* Algorithm on the following figure. Explicitly write down the queue at each step.



Difference between Best First Search and A*

DIFFERENCE BETWEEN BEST FIRST SEARCH AND A*

Best First Search is Greedy search method. So not considering previous knowledge.

$f(n)=h'(n)$

A* algorithm gives optimal solution. It considers previous knowledge also…

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

Admissibility Condition of A* Algorithm

$h(n) \leq h^*(n)$ ∴ Underestimation
$h(n) \geq h^*(n)$ ∴ Overestimation

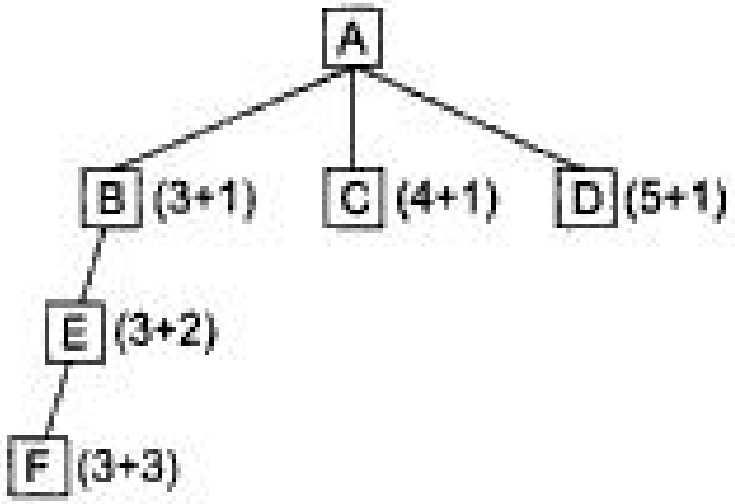


Fig. 3.4 *h' Underestimates h*

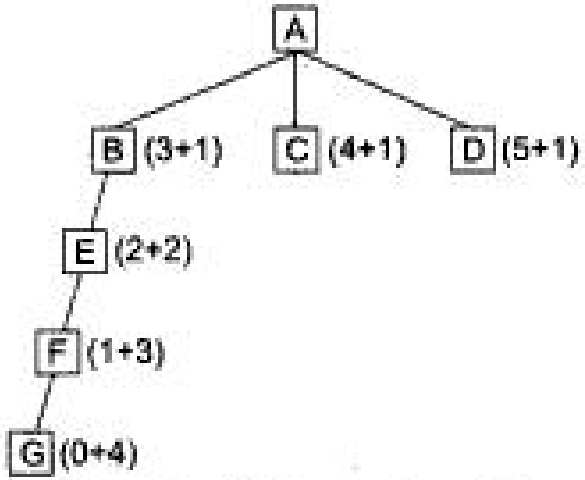


Fig. 3.5 *h' Overestimates h*

Game Playing

Game Playing

- Game playing was one of the first tasks undertaken in Artificial Intelligence.
- The very first game that is been tackled in AI is chess.

Types of Game

- Perfect Information Game
  - In which player knows all the possible moves of himself and opponent and their results
  - Ex: Chess
- Imperfect Information Game
  - In which player does not know all the possible moves of the opponent.
  - Bridge since all the cards are not visible to player

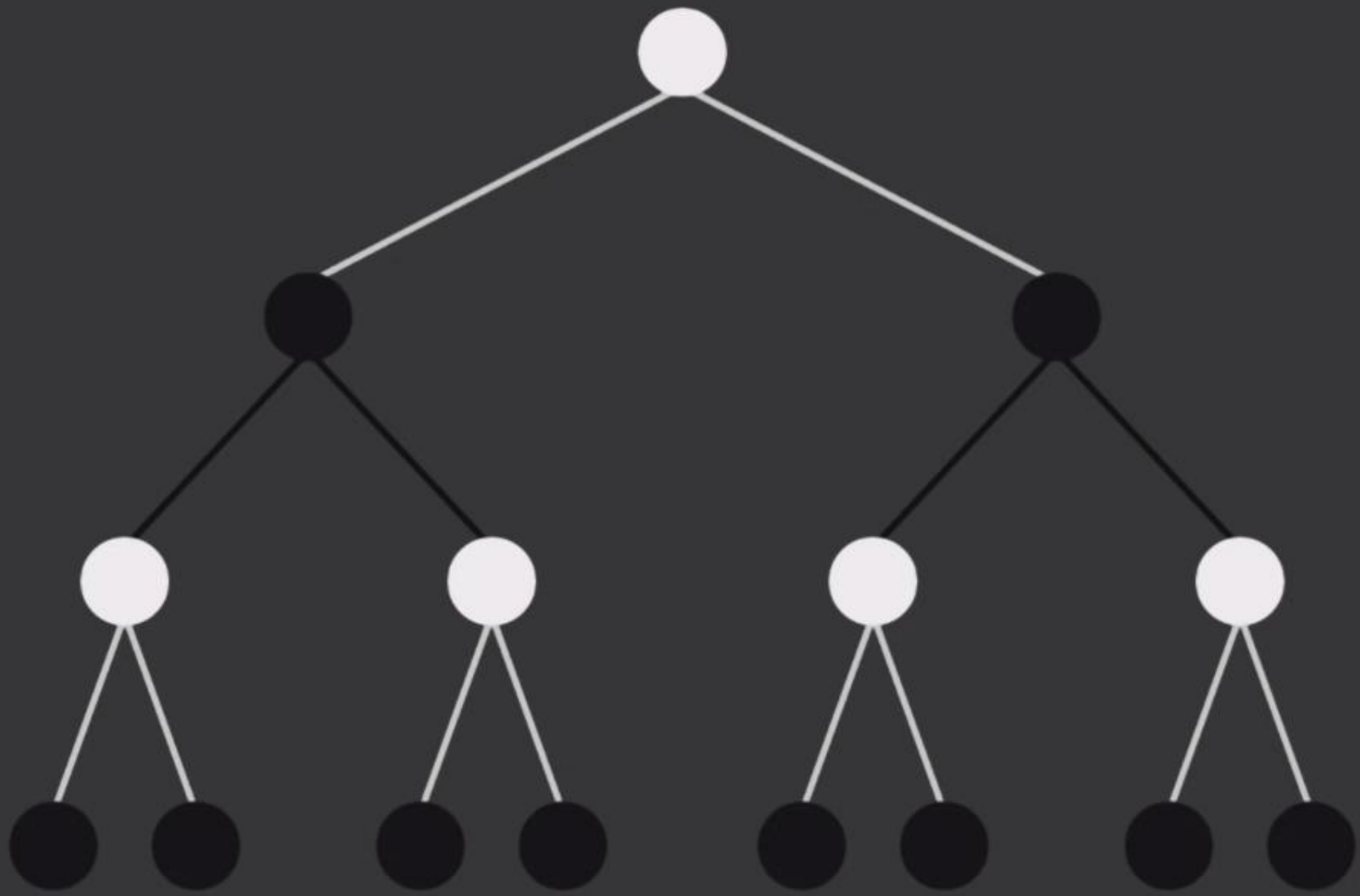
What is Game Playing in AI?

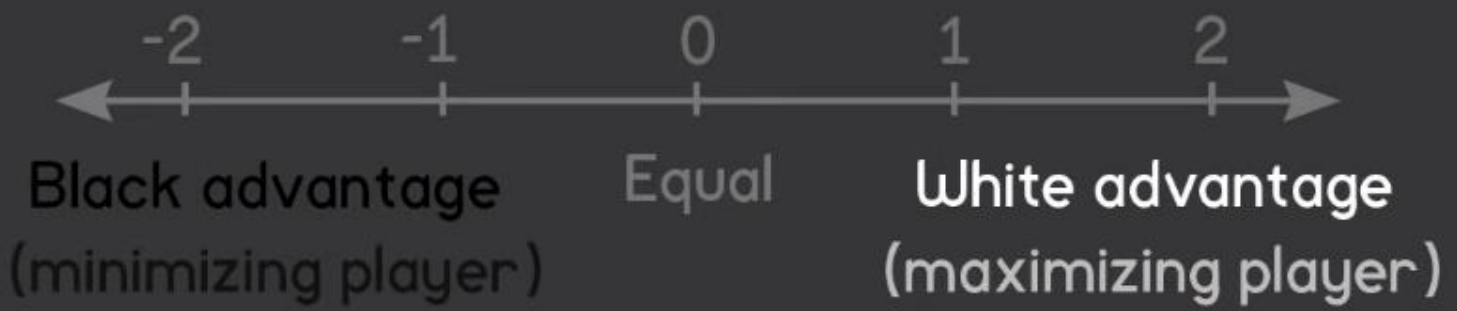
- Game playing is a search problem defined by following components:
- **Initial state:** This defines initial configuration of the game and identifies first payer to move.
- **Successor function:** This identifies which are the possible states that can be achieved from the current state. This function returns a list of (move, state) pairs, each indicating a legal move and the resulting state.
- **Goal test:** Which checks whether a given state is a goal state or not. States where the game ends are called as terminal states.
- **Path cost / utility / payoff function:** Which gives a numeric value for the terminal states? In chess, the outcome is win, loss or draw, with values +1, -1, or 0. Some games have wider range of possible outcomes.

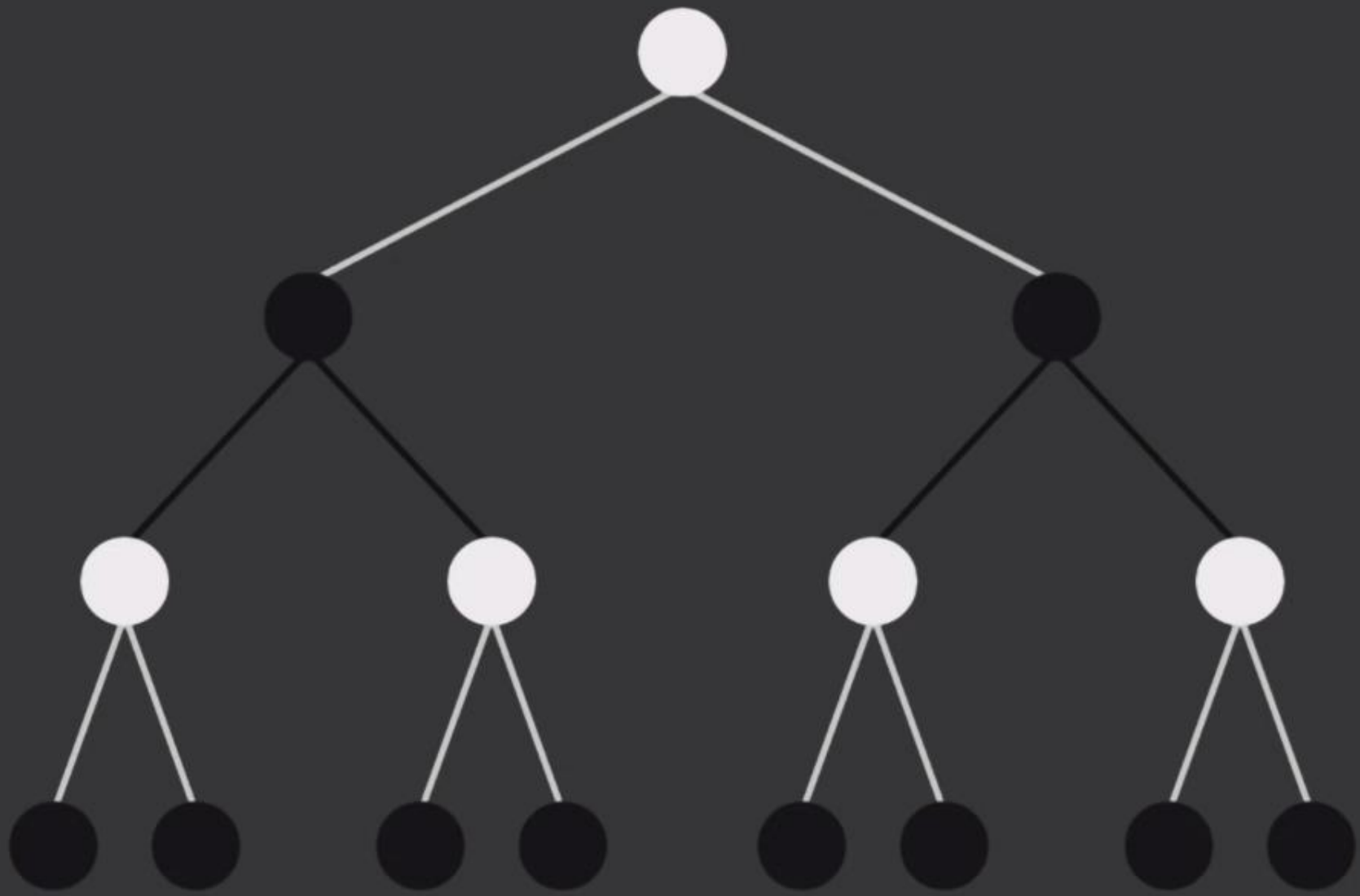
Characteristics of game playing

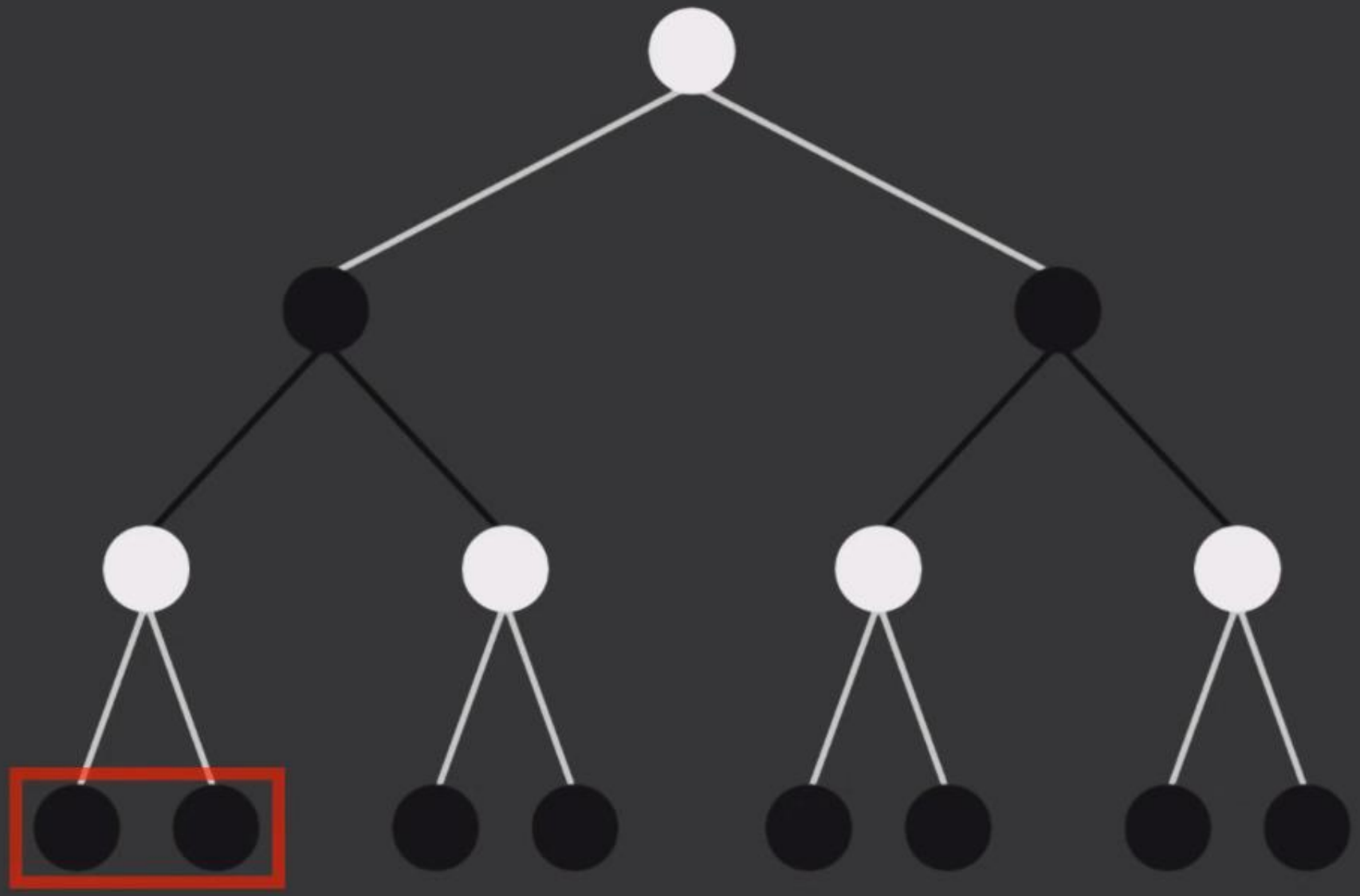
- **Unpredictable Opponent:** Generally we cannot predict the behavior of the opponent. Thus we need to find a solution which is a strategy specifying a move for every possible opponent move or every possible state.
- **Time Constraints:** Every game has a time constraints. Thus it may be infeasible to find the best move in this time.

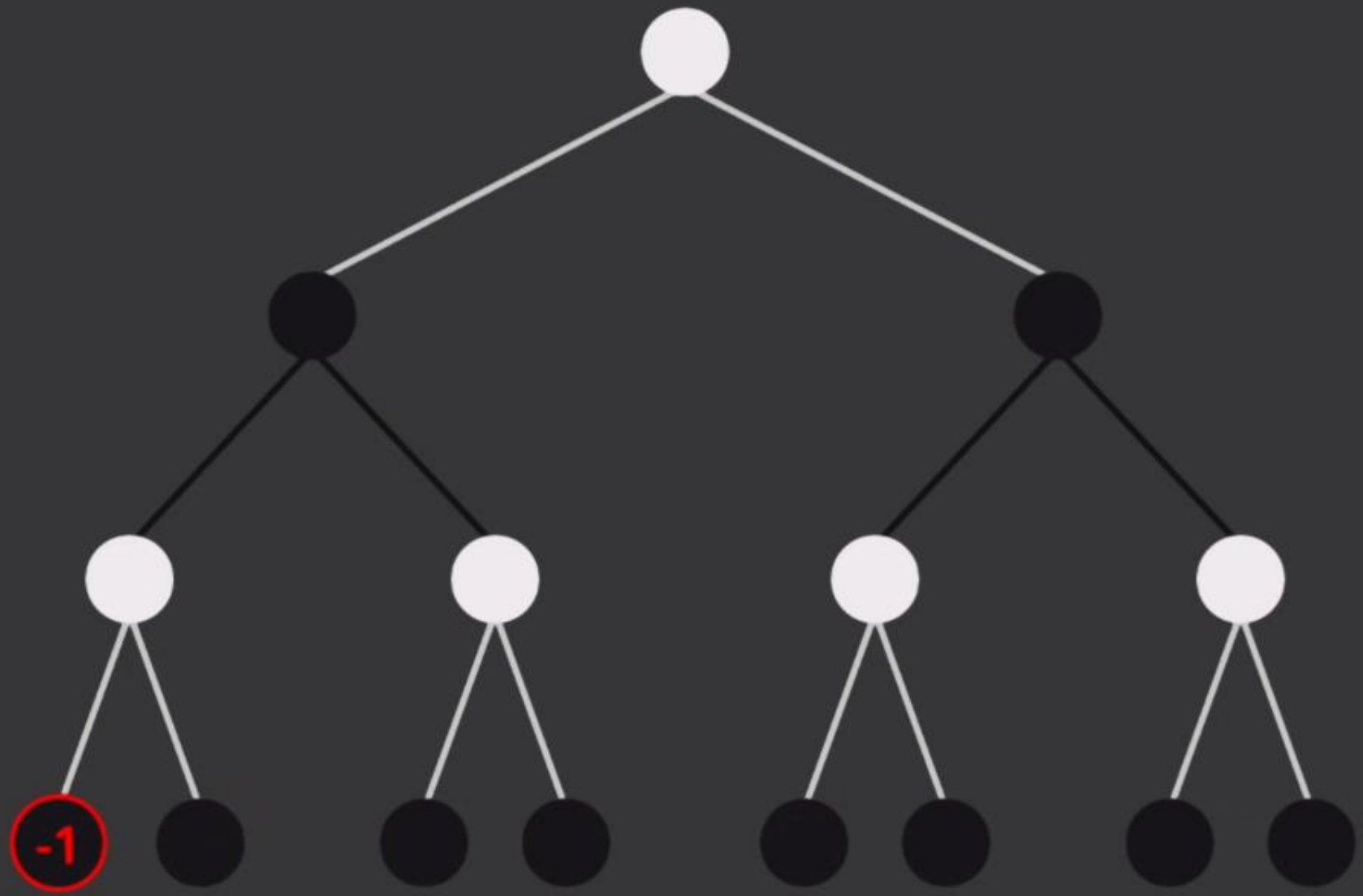
Minimax game playing

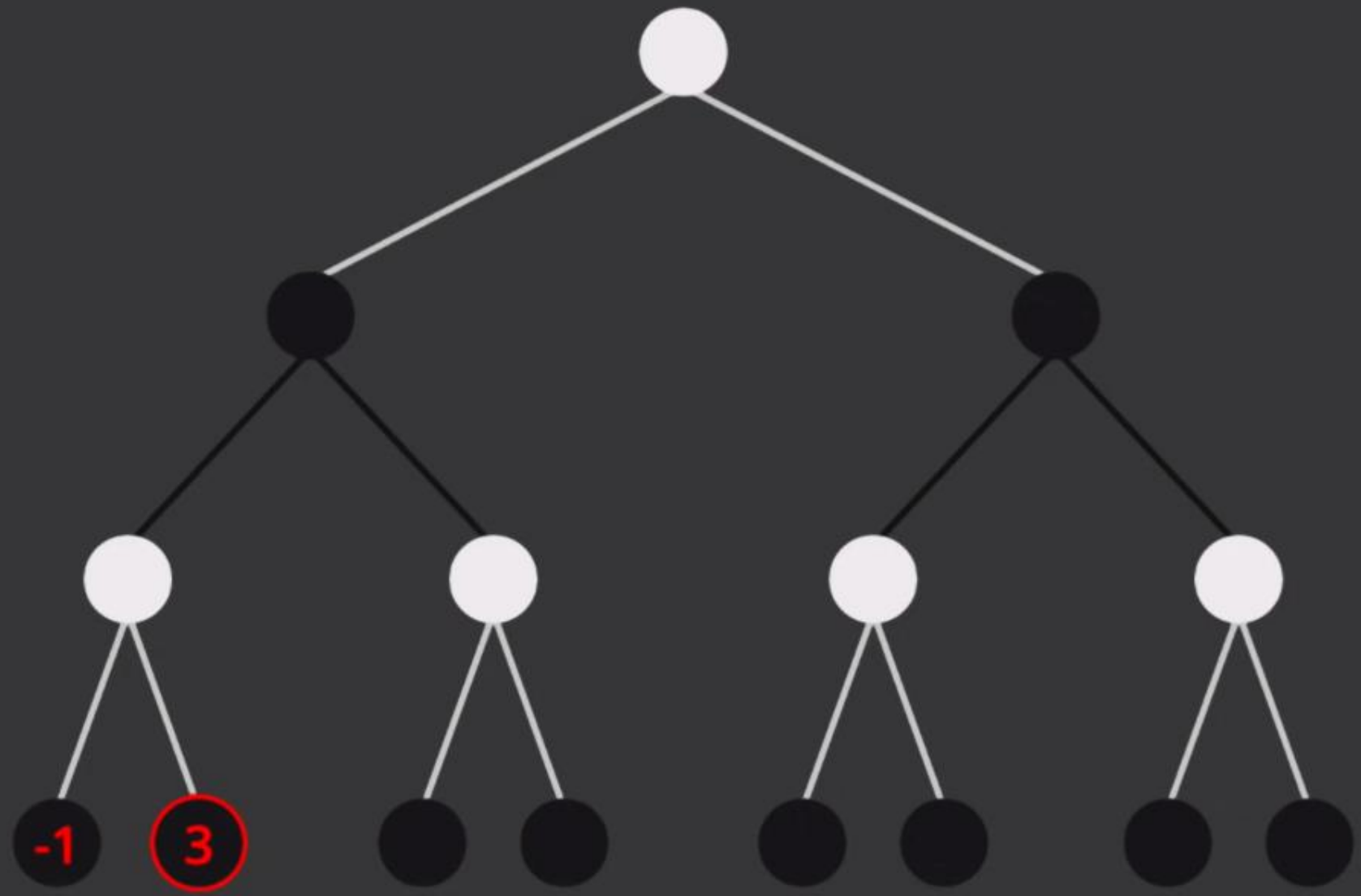


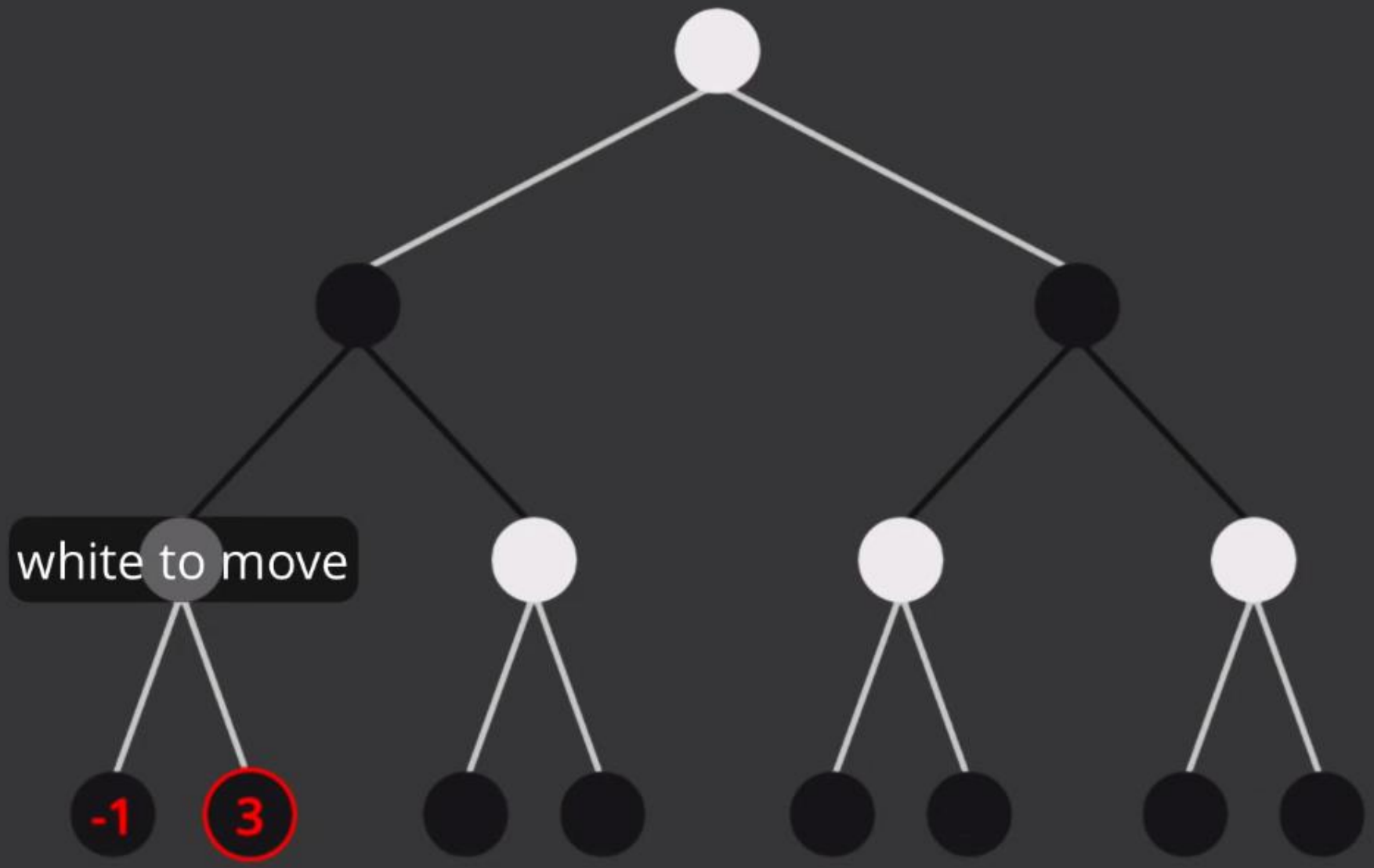

-2
-1
0
1
2
Black advantage
(minimizing player)
Equal
White advantage
(maximizing player)

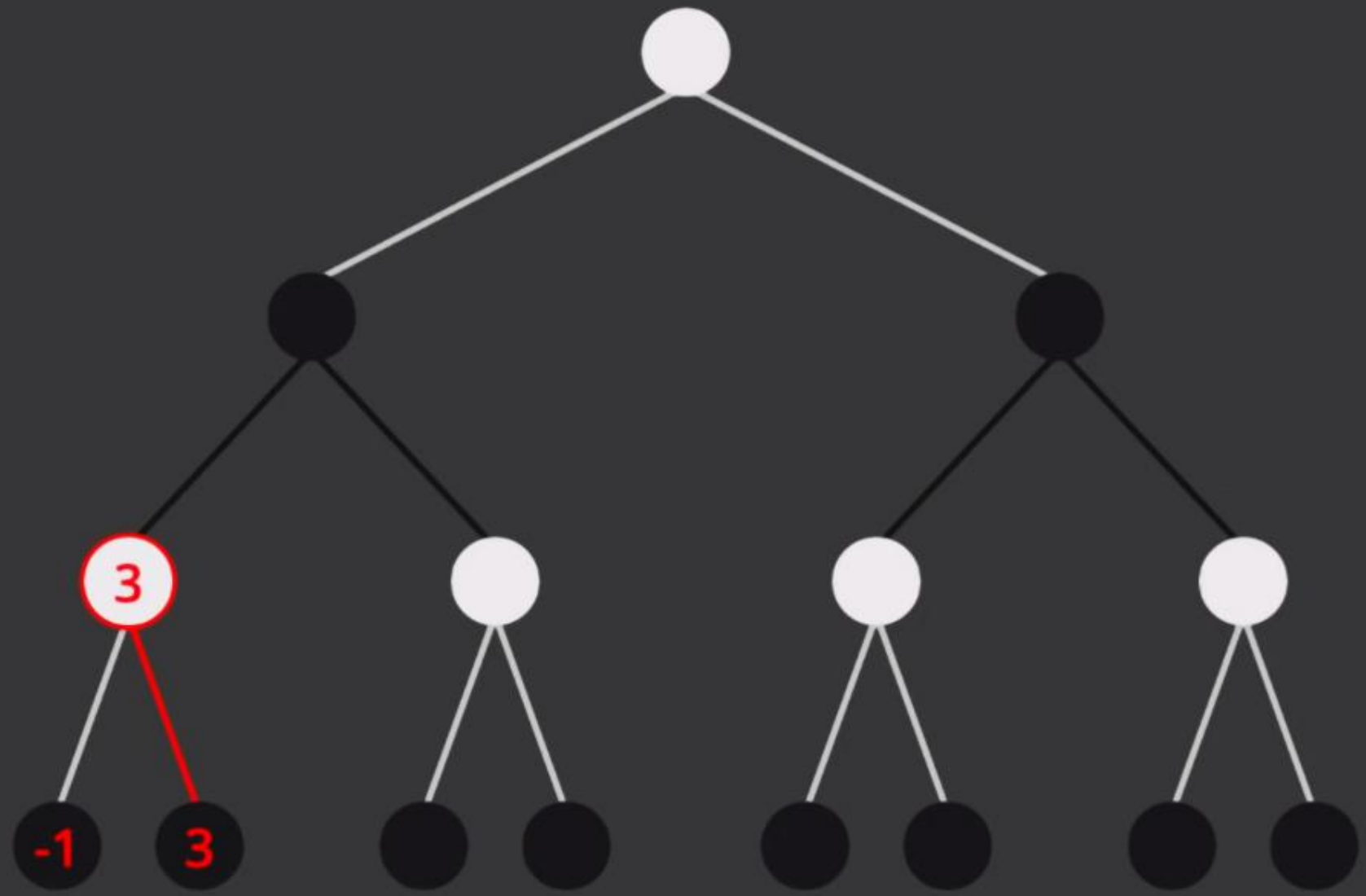


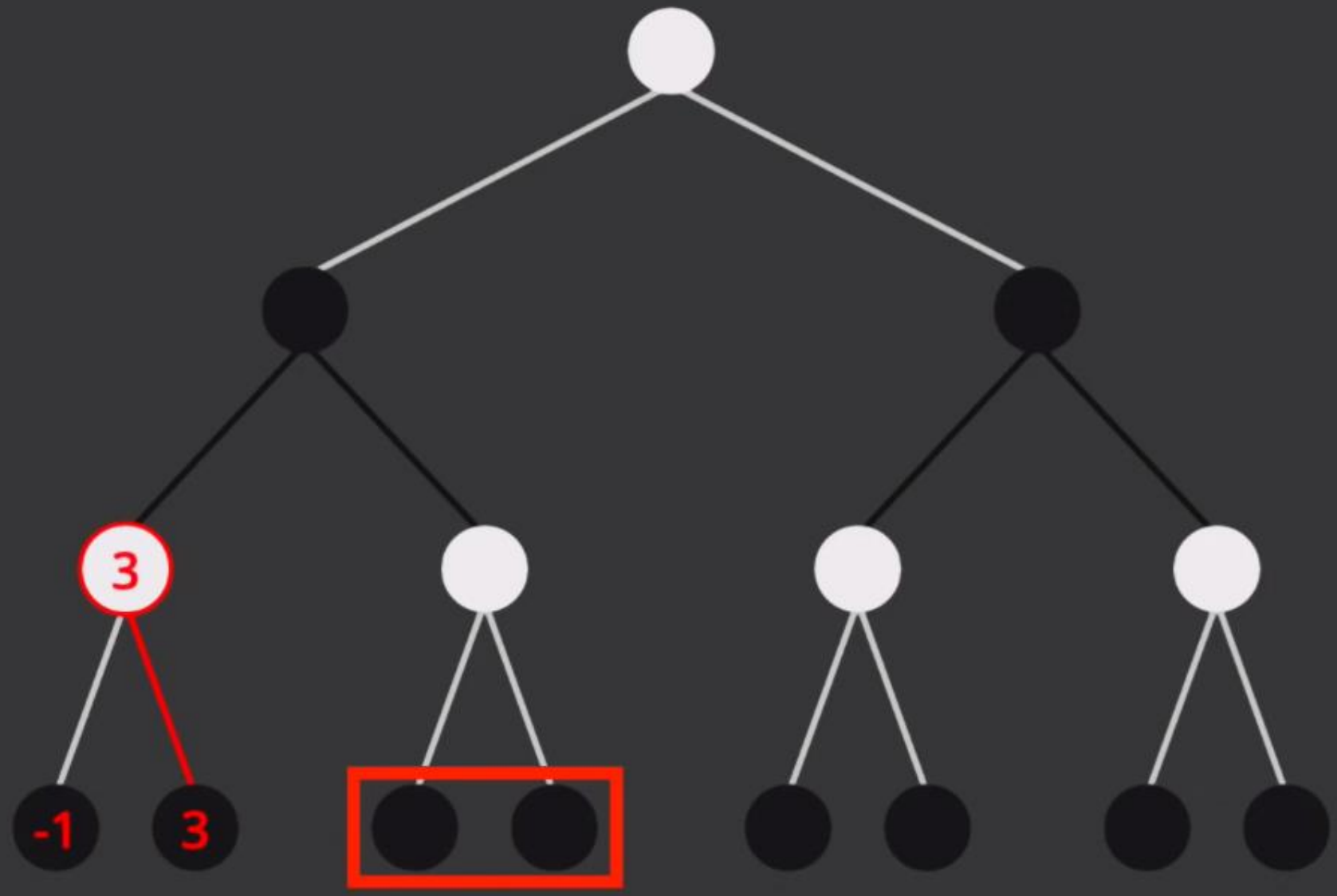


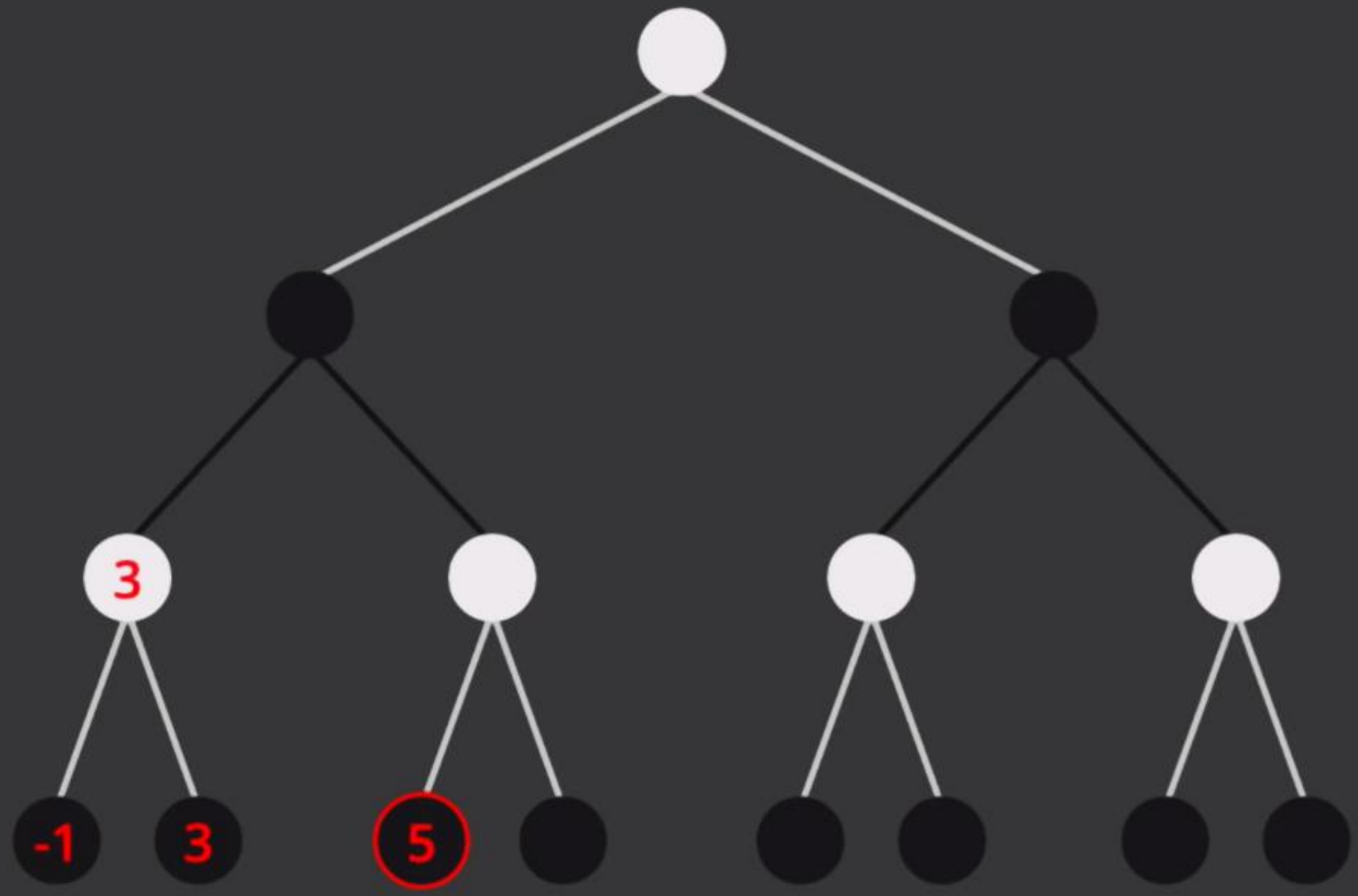

-1

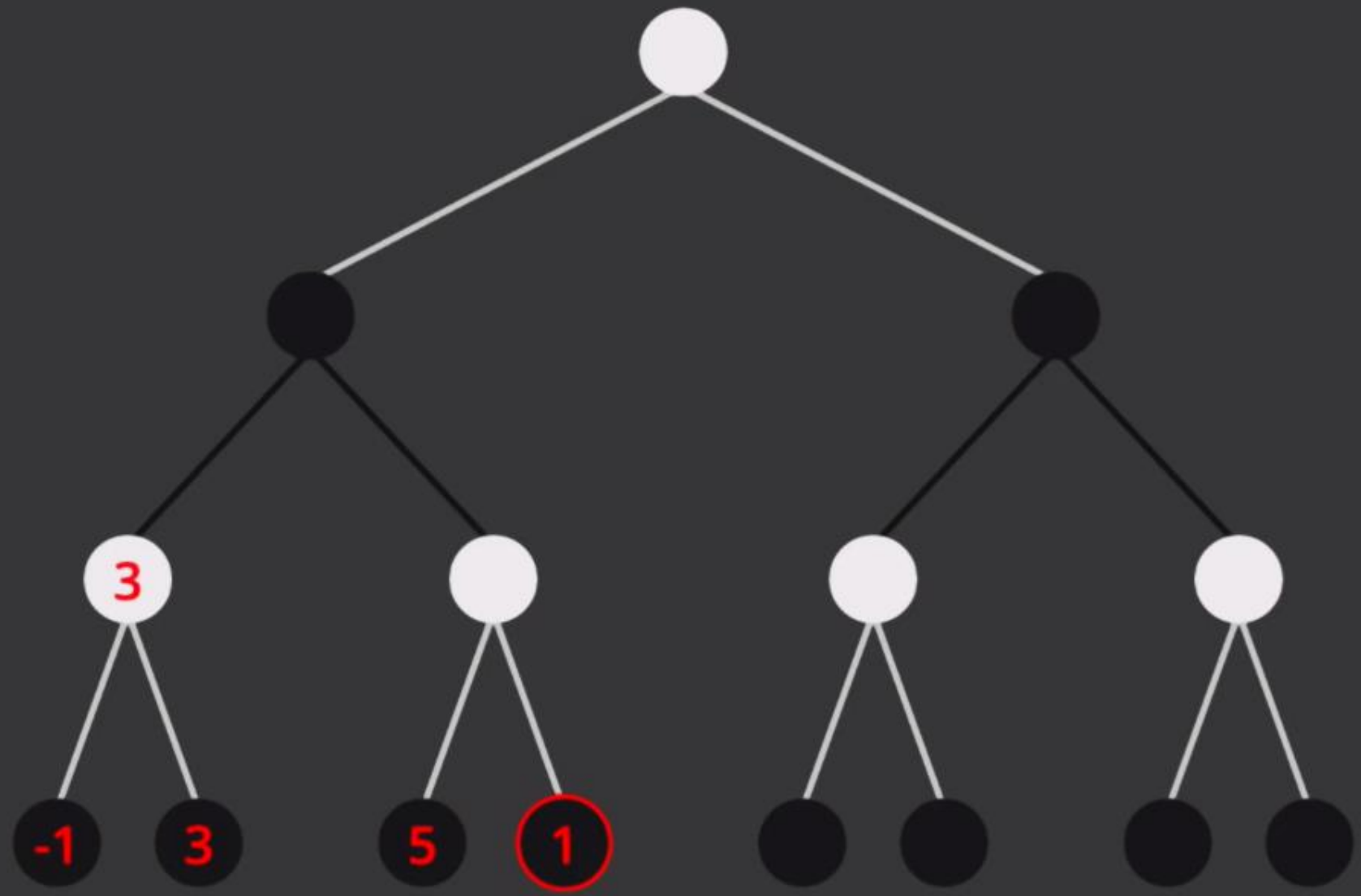

-1
3

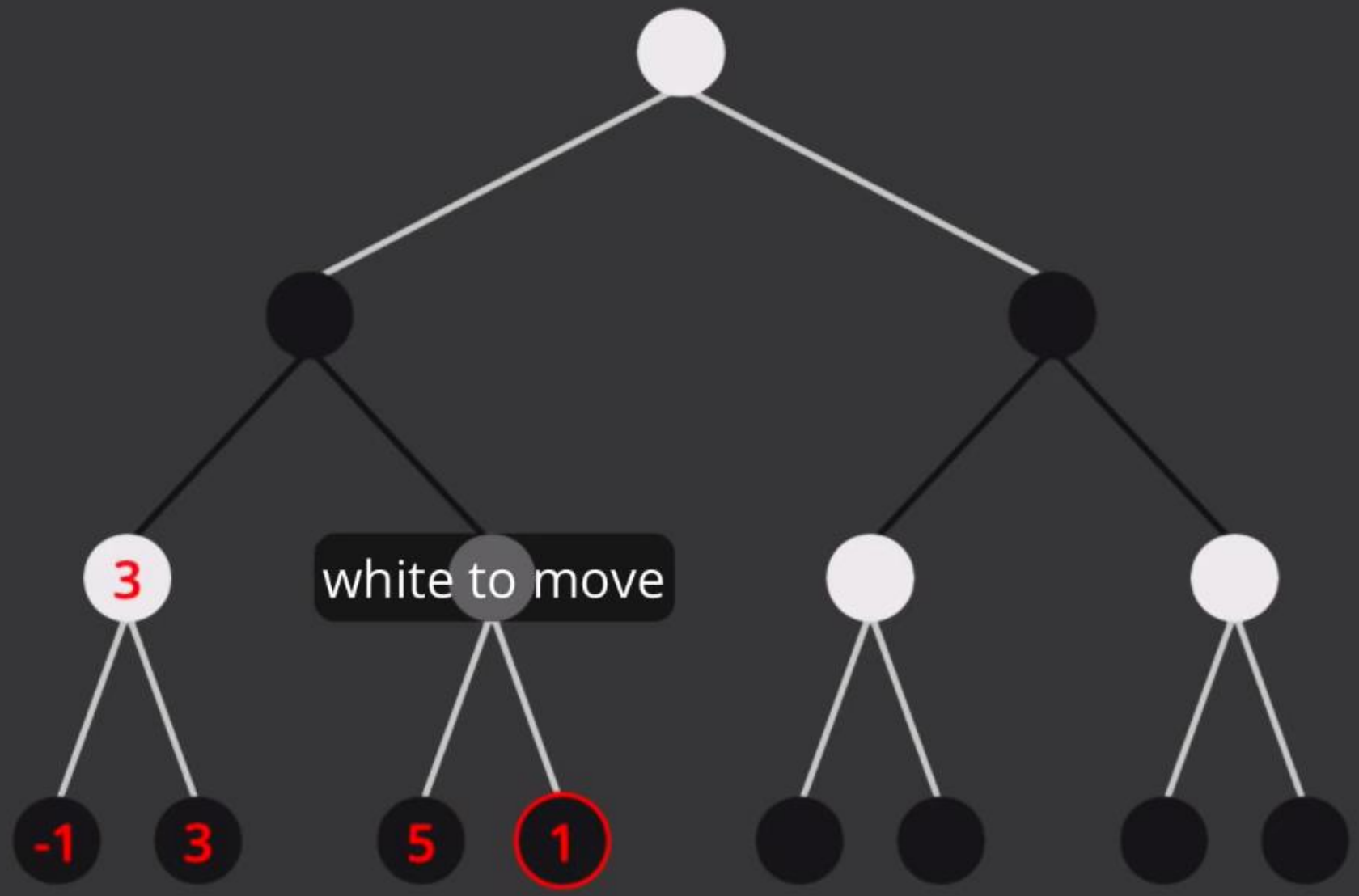

white to move
-1
3

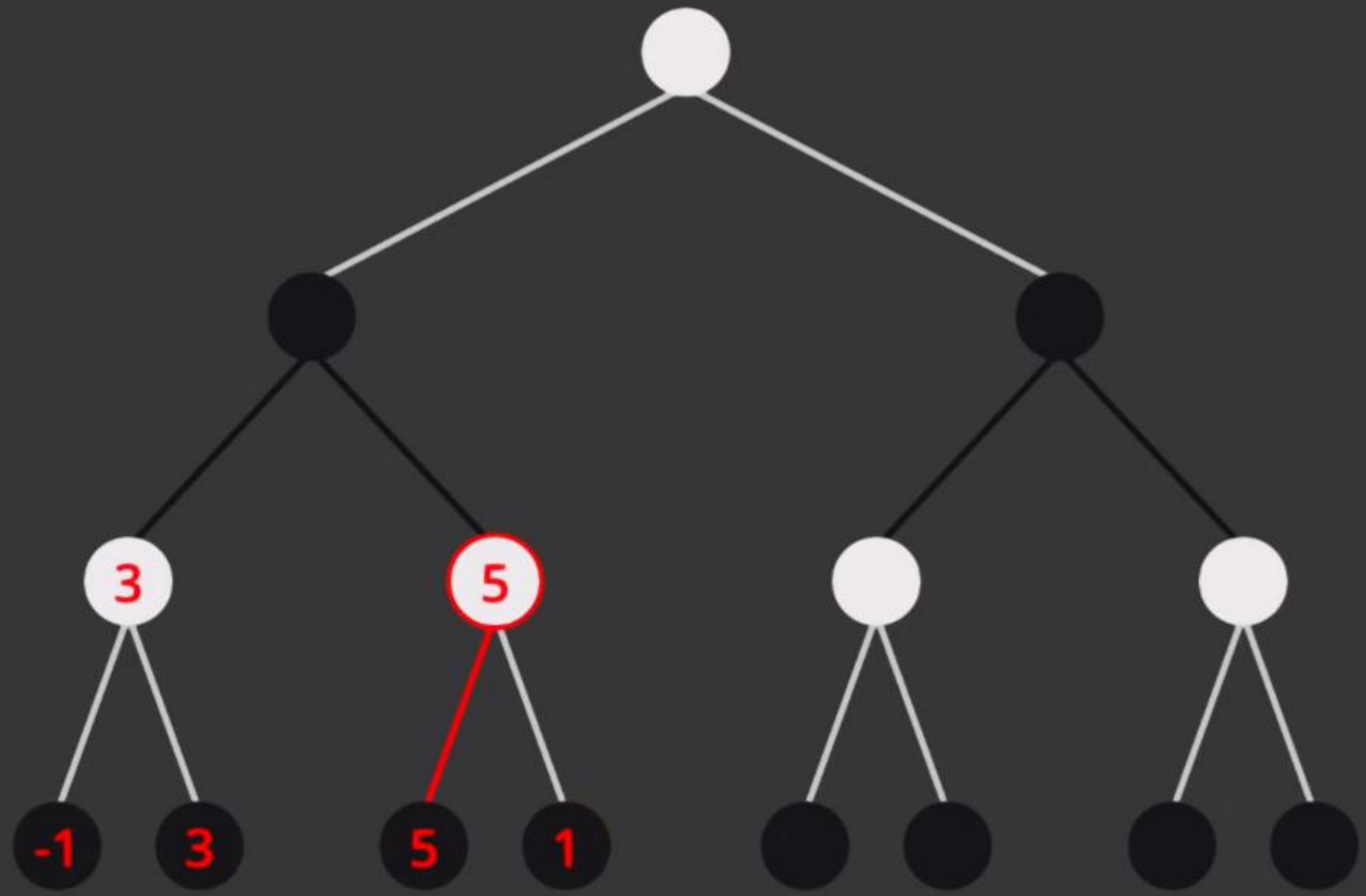

3
-1
3

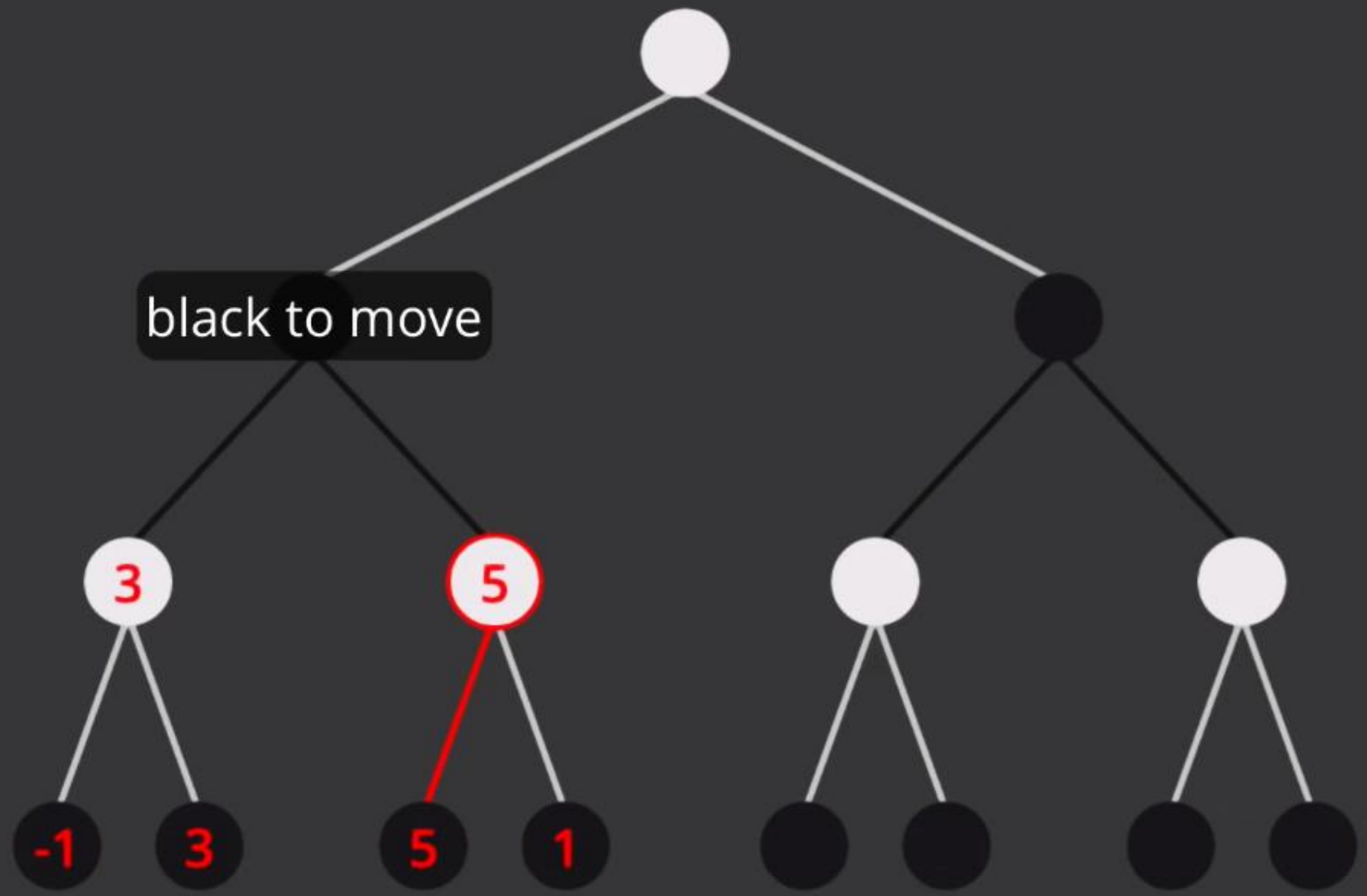

3
-1
3

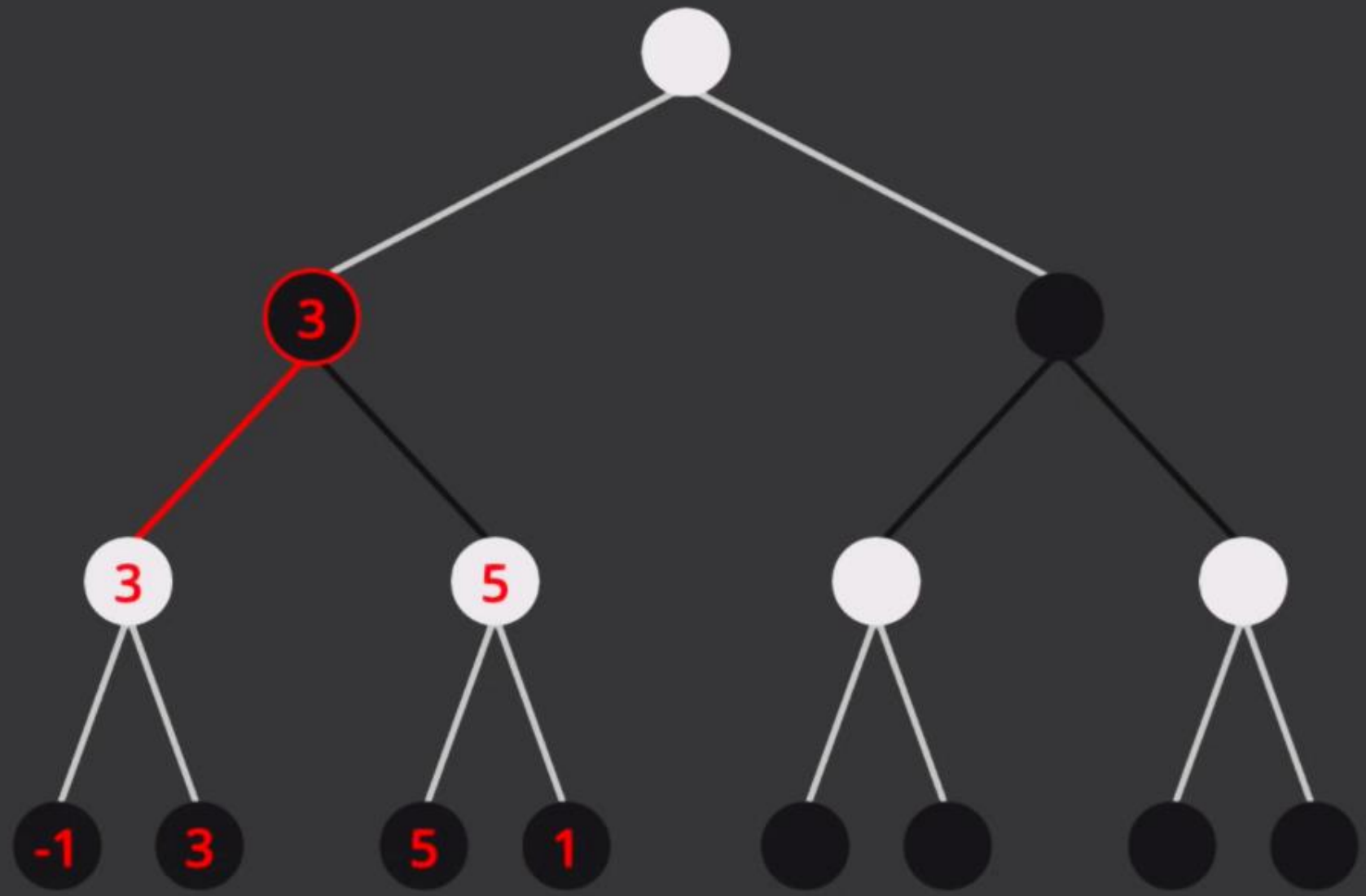

3
-1
3
5

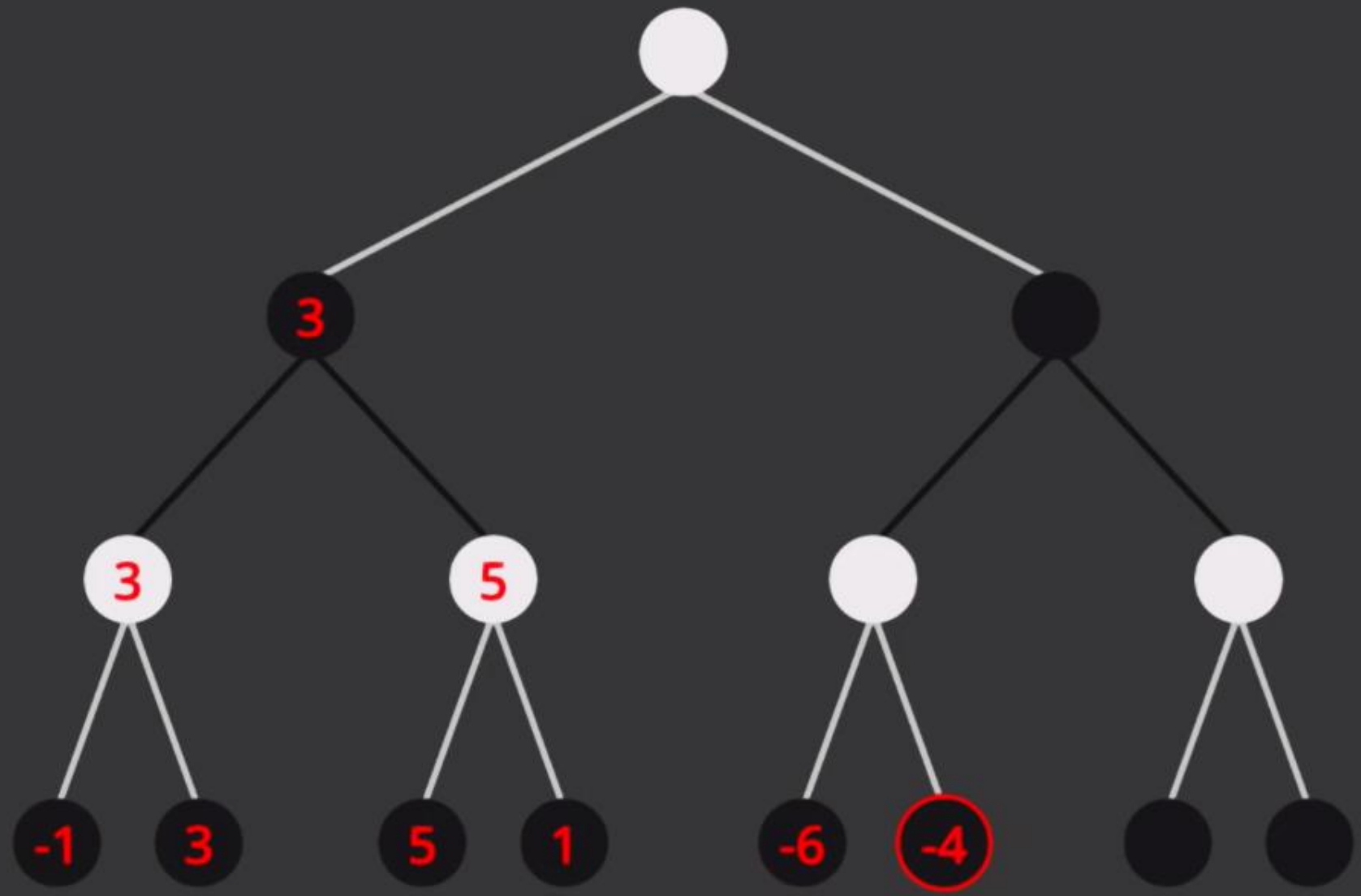

3
-1
3
5
1

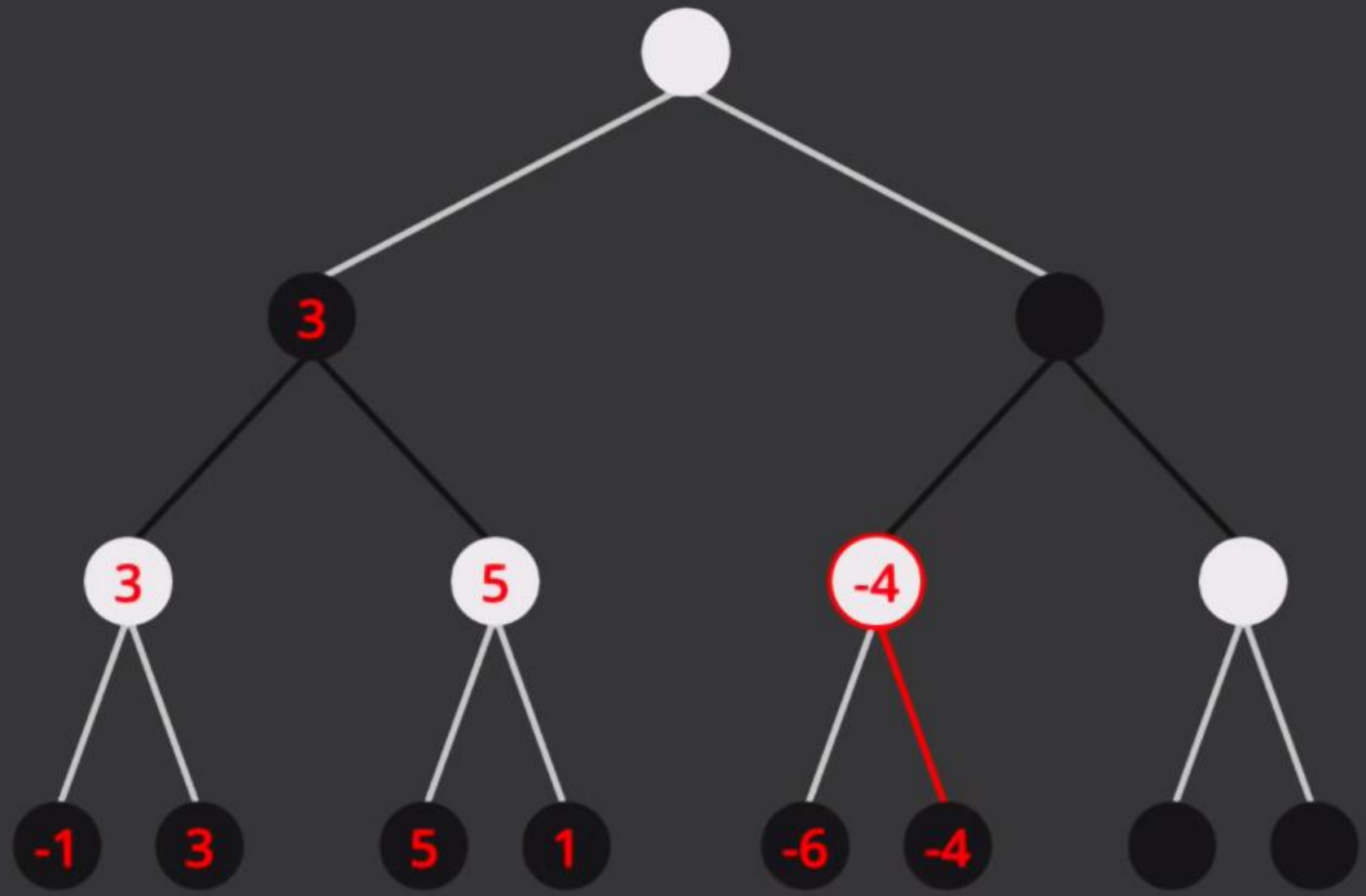

3
white to move
-1
3
5
1

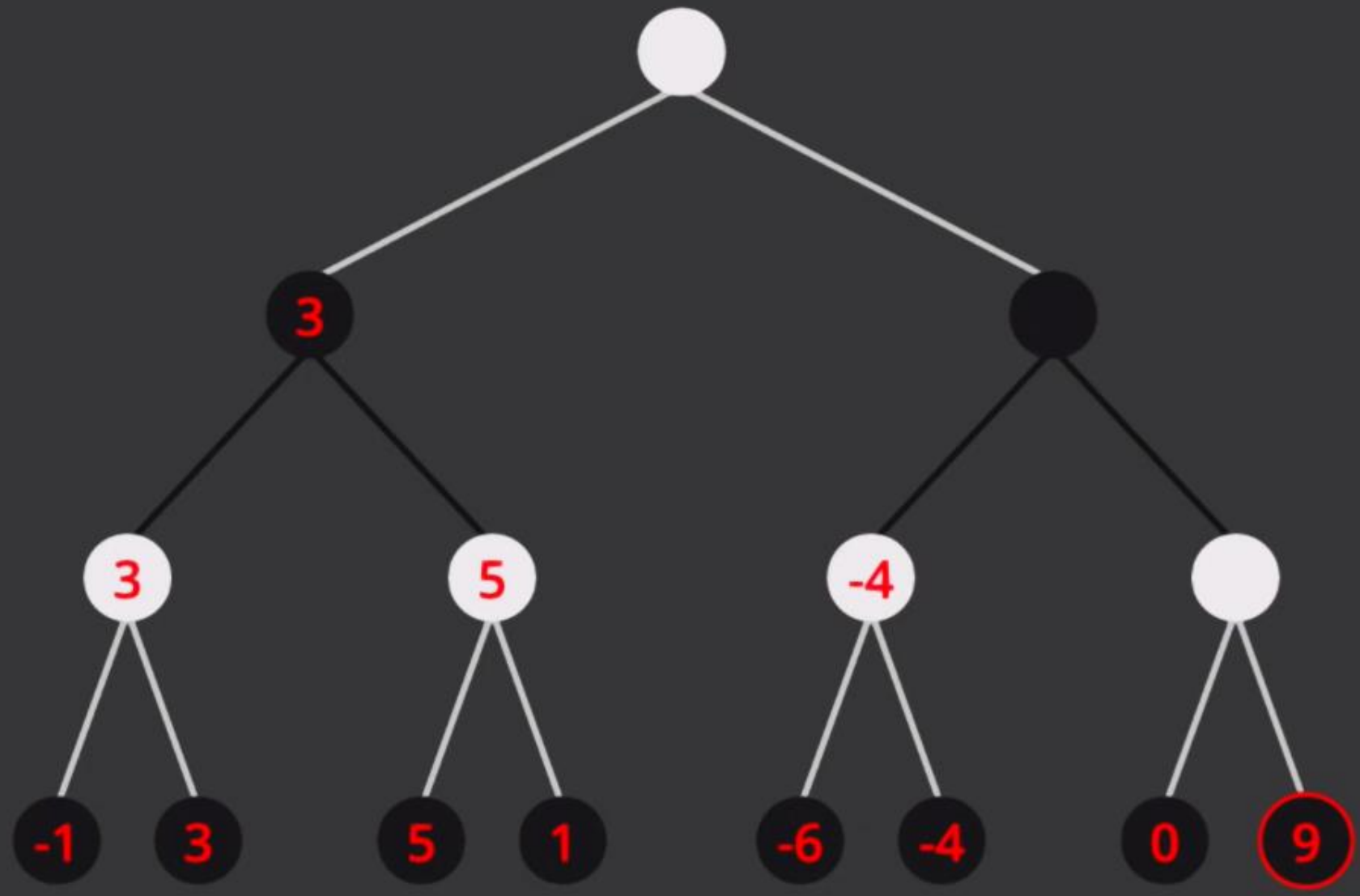

3
5
-1
3
5
1

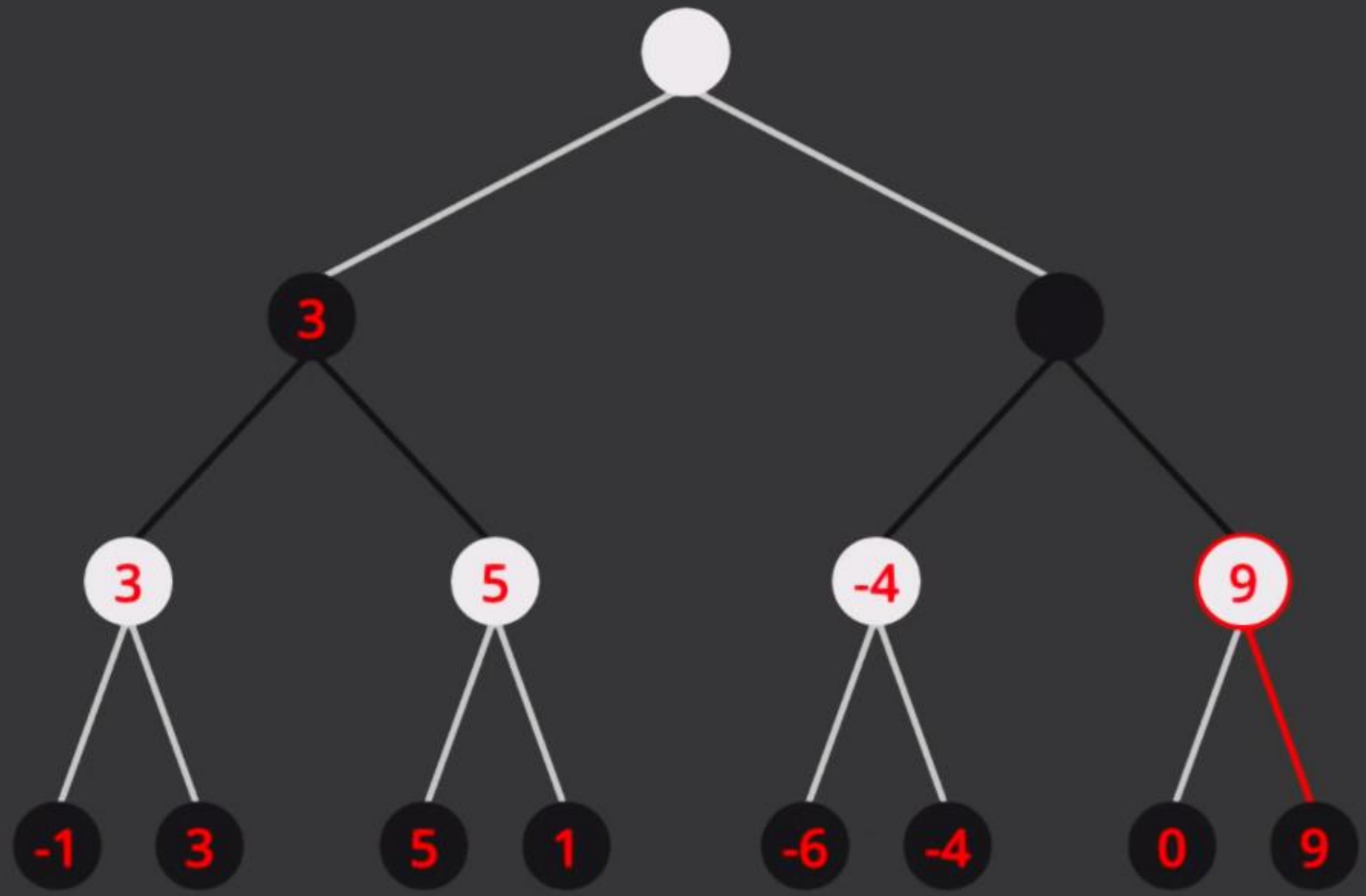

black to move
3
5
-1
3
5
1

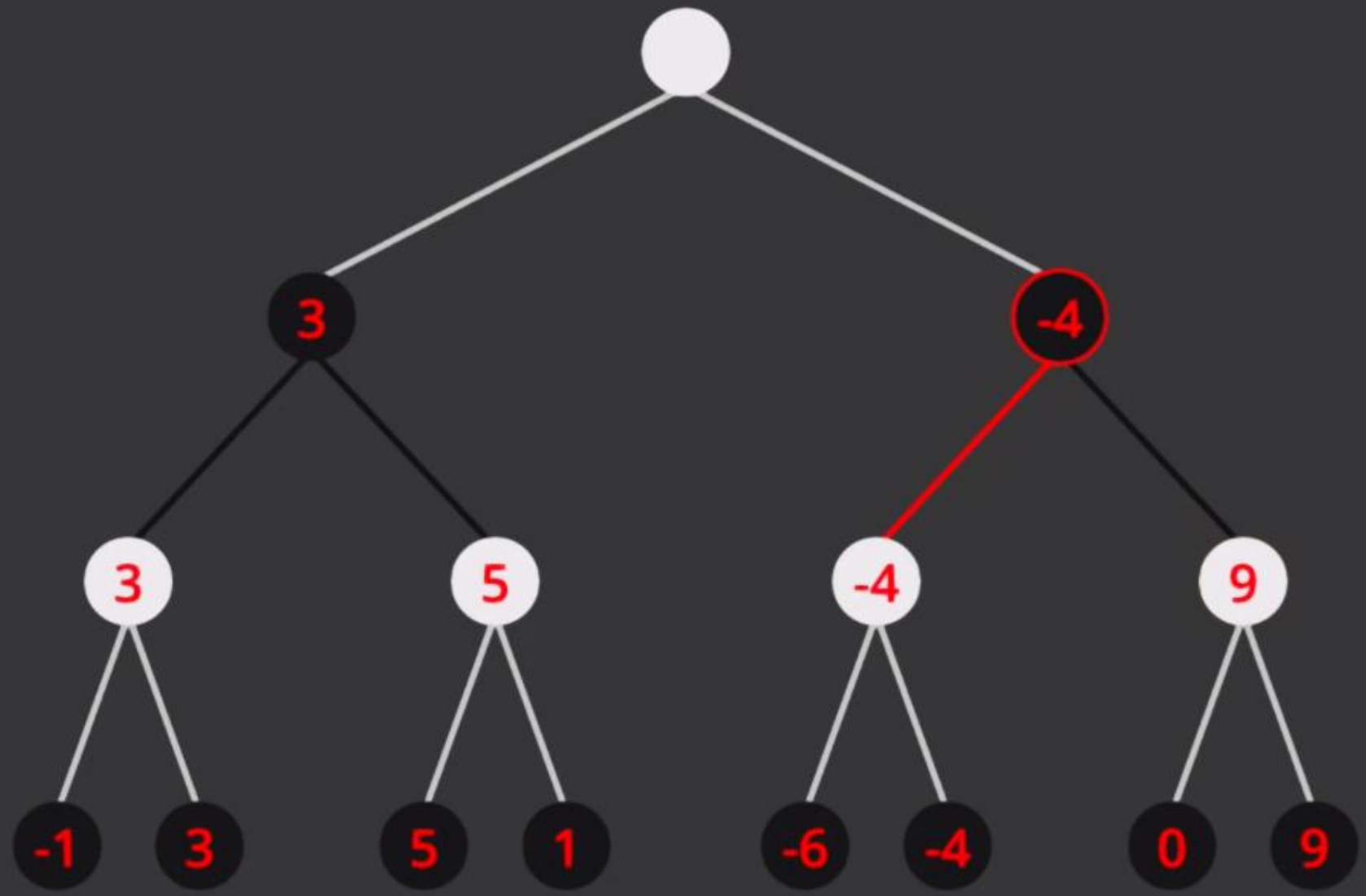

3
3
5
-1
3
5
1

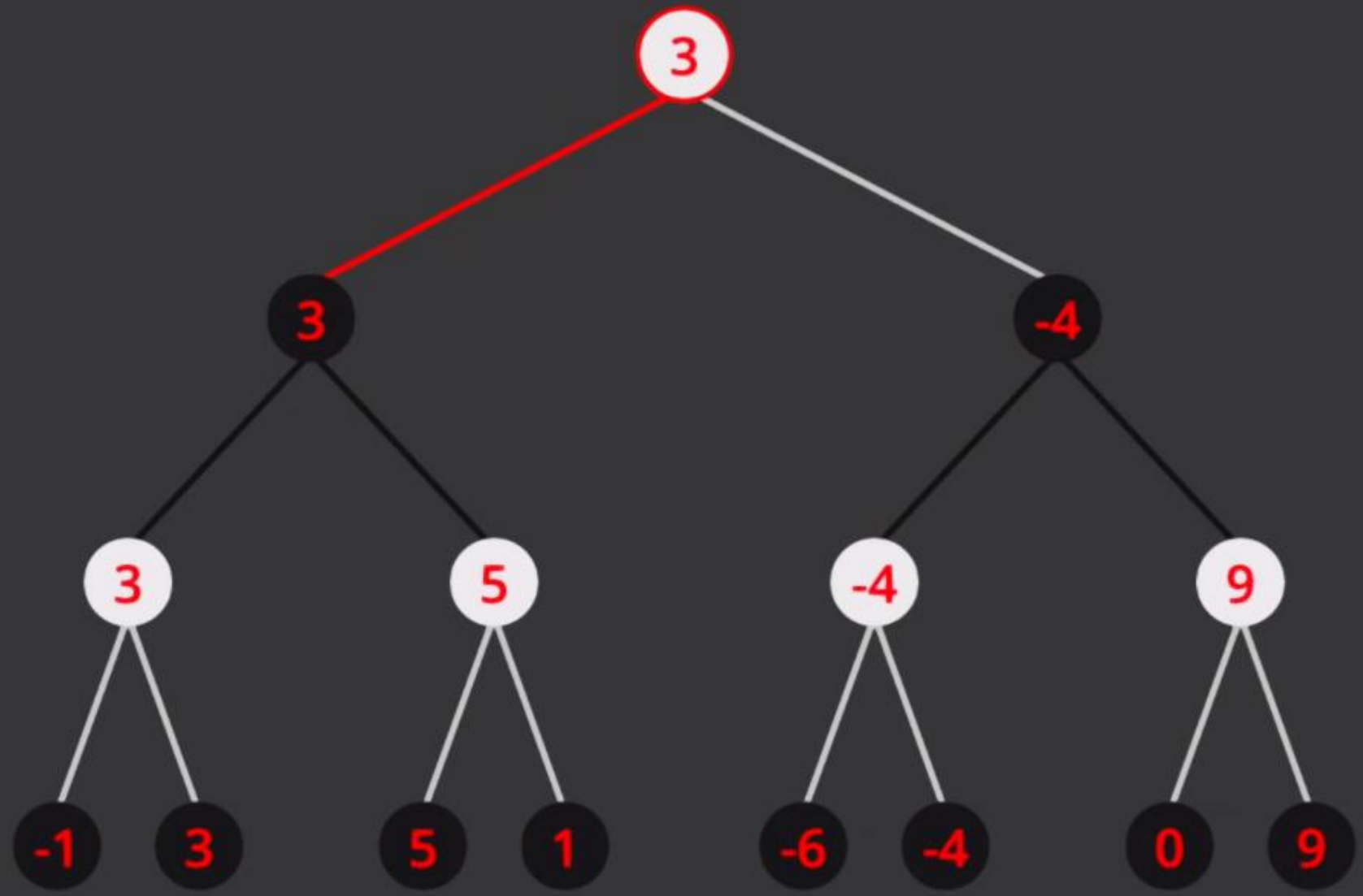

3
3
5
-1
3
5
1
-6
-4


3
3
5
-4
-1
3
5
1
-6
-4


3
3
5
-4
-1
3
5
1
-6
-4
0
9


3
3
5
-4
9
-1
3
5
1
-6
-4
0
9


3
-4
3
5
-4
9
-1
3
5
1
-6
-4
0
9


3
3
-4
3
5
-4
9
-1
3
5
1
-6
-4
0
9

```
function minimax(position, depth)
```

```
function minimax(position, depth, maximizingPlayer)
```

```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position
```

```
function minimax(position, depth, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
```

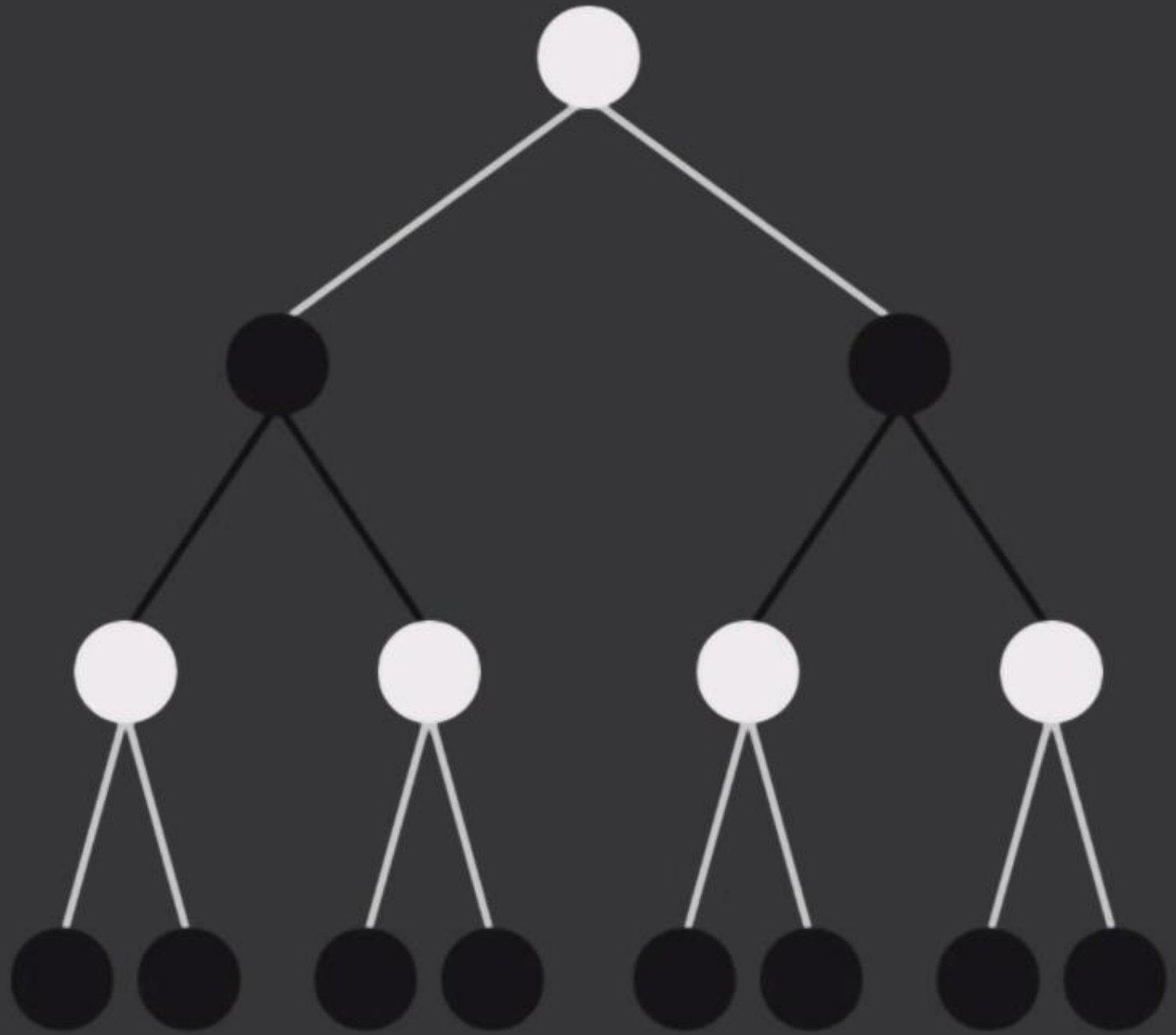
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
```

```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval
```

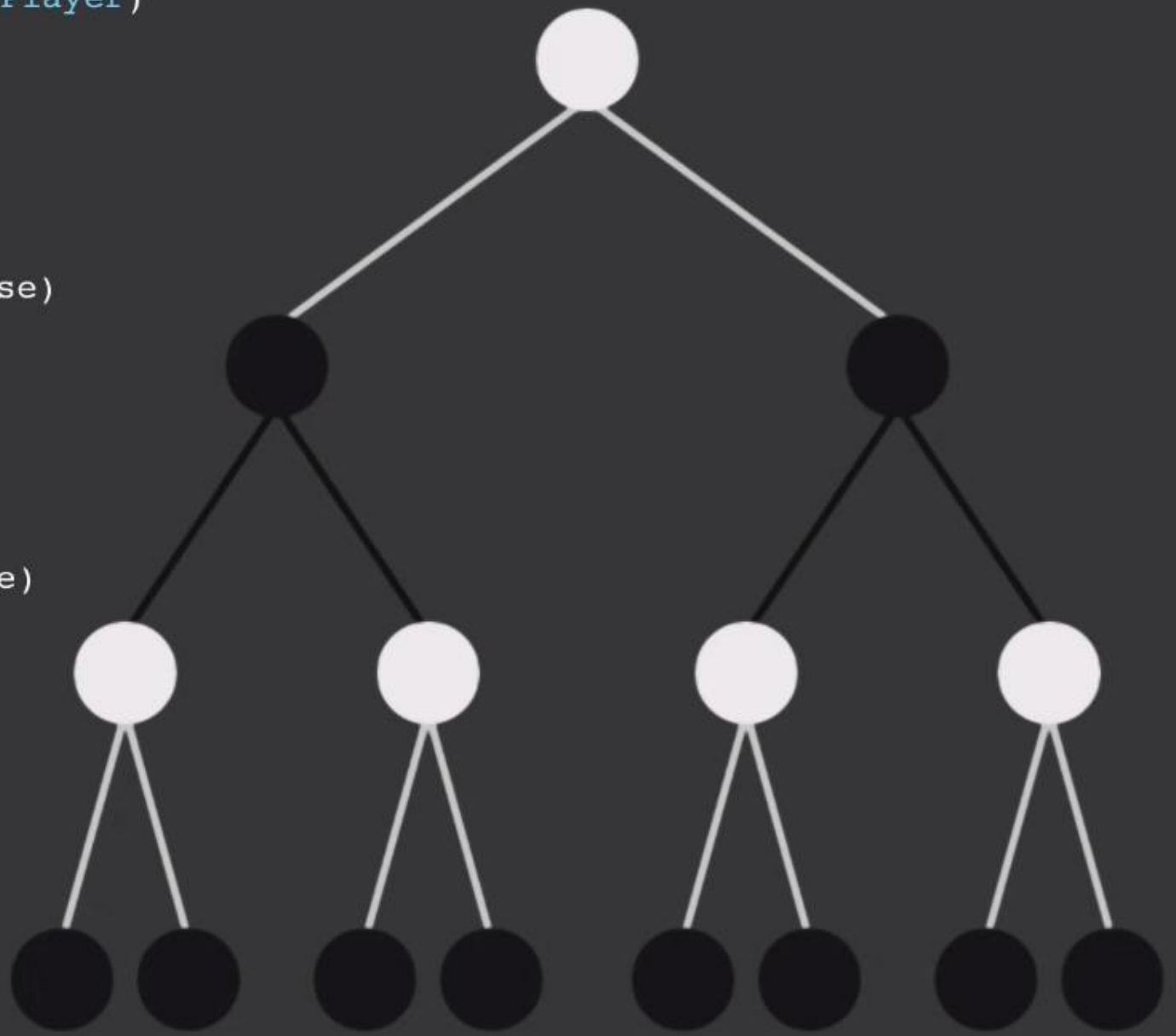
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
```

```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

Minimax Algorithm Example

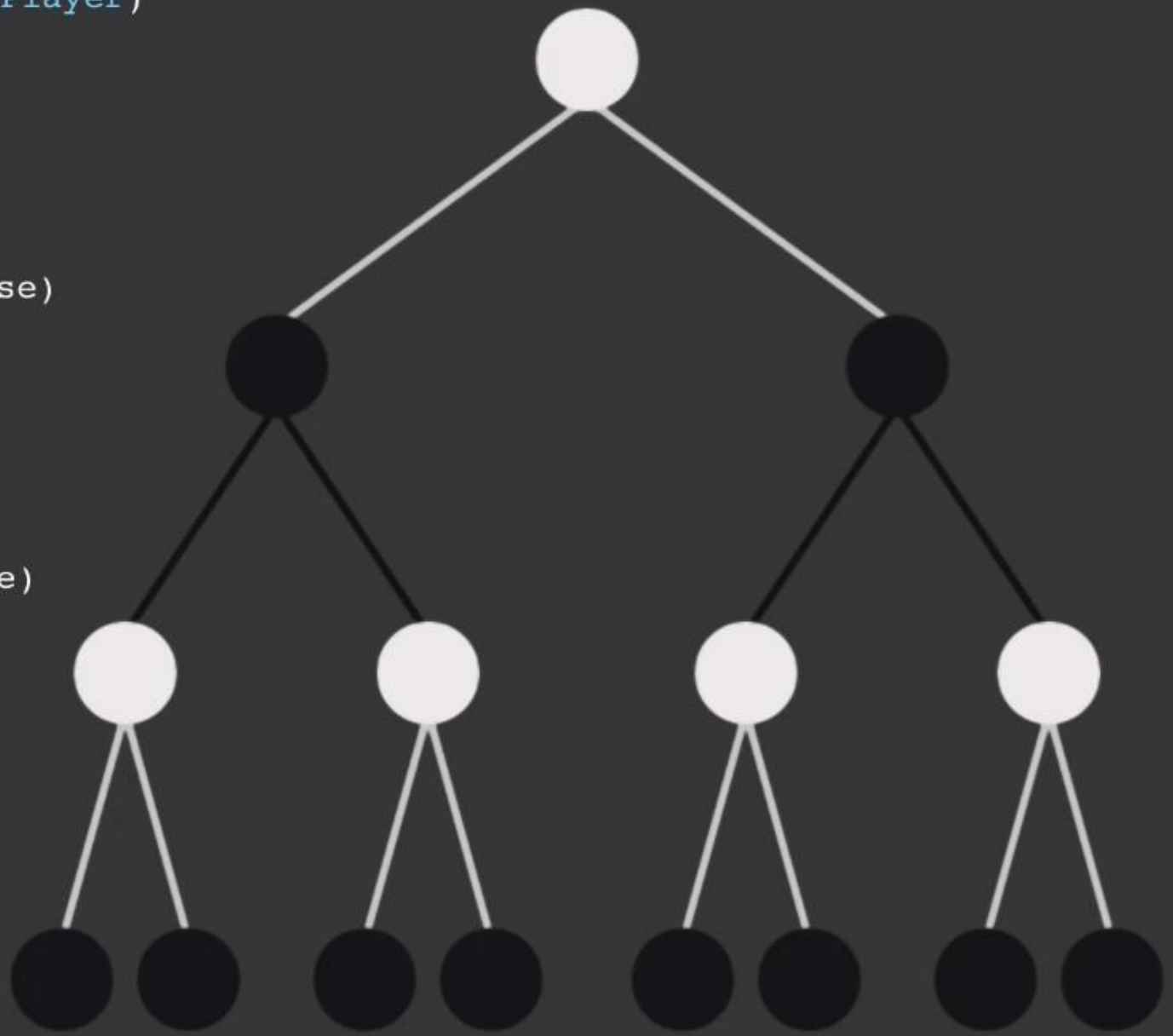


```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```



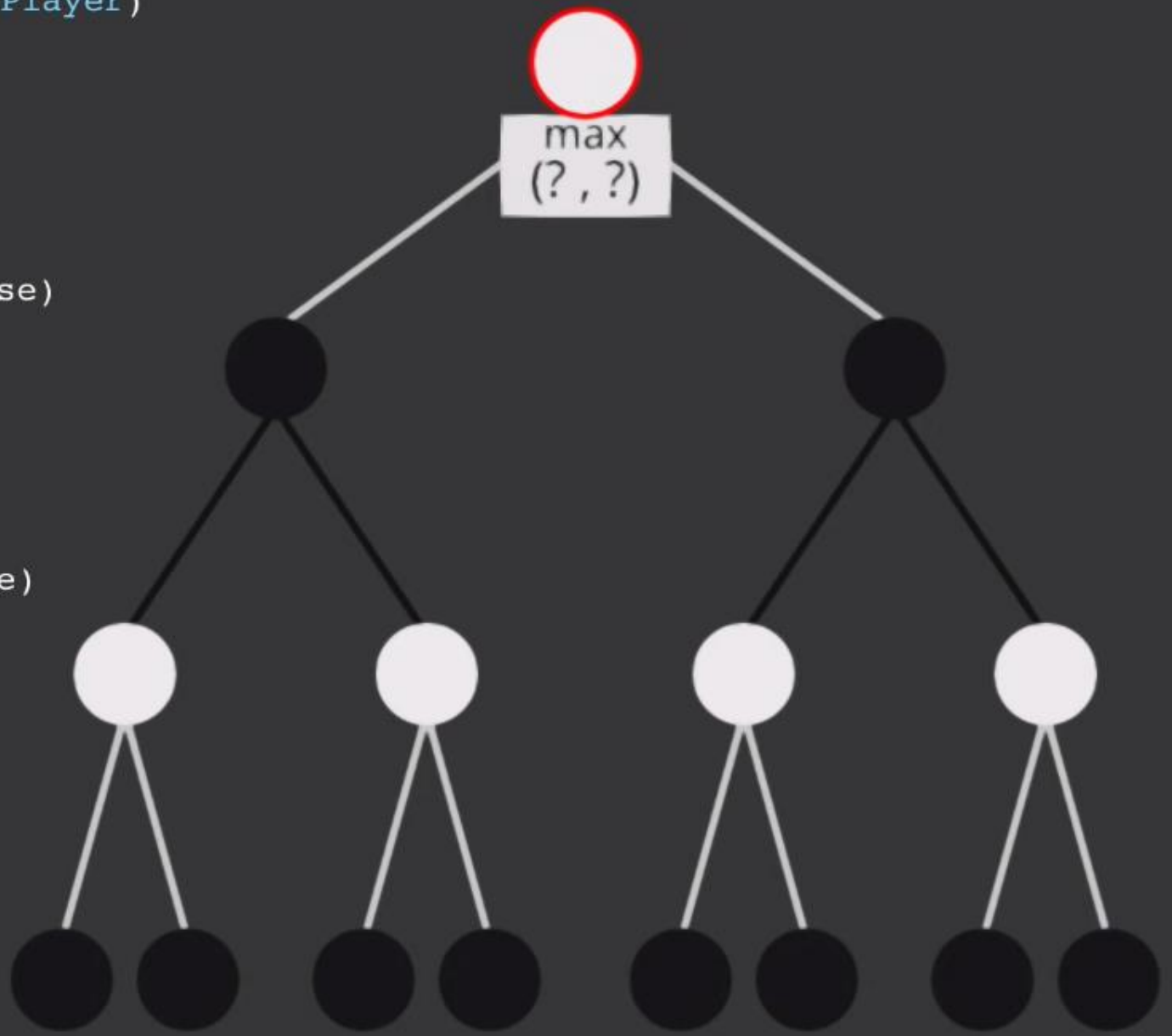
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



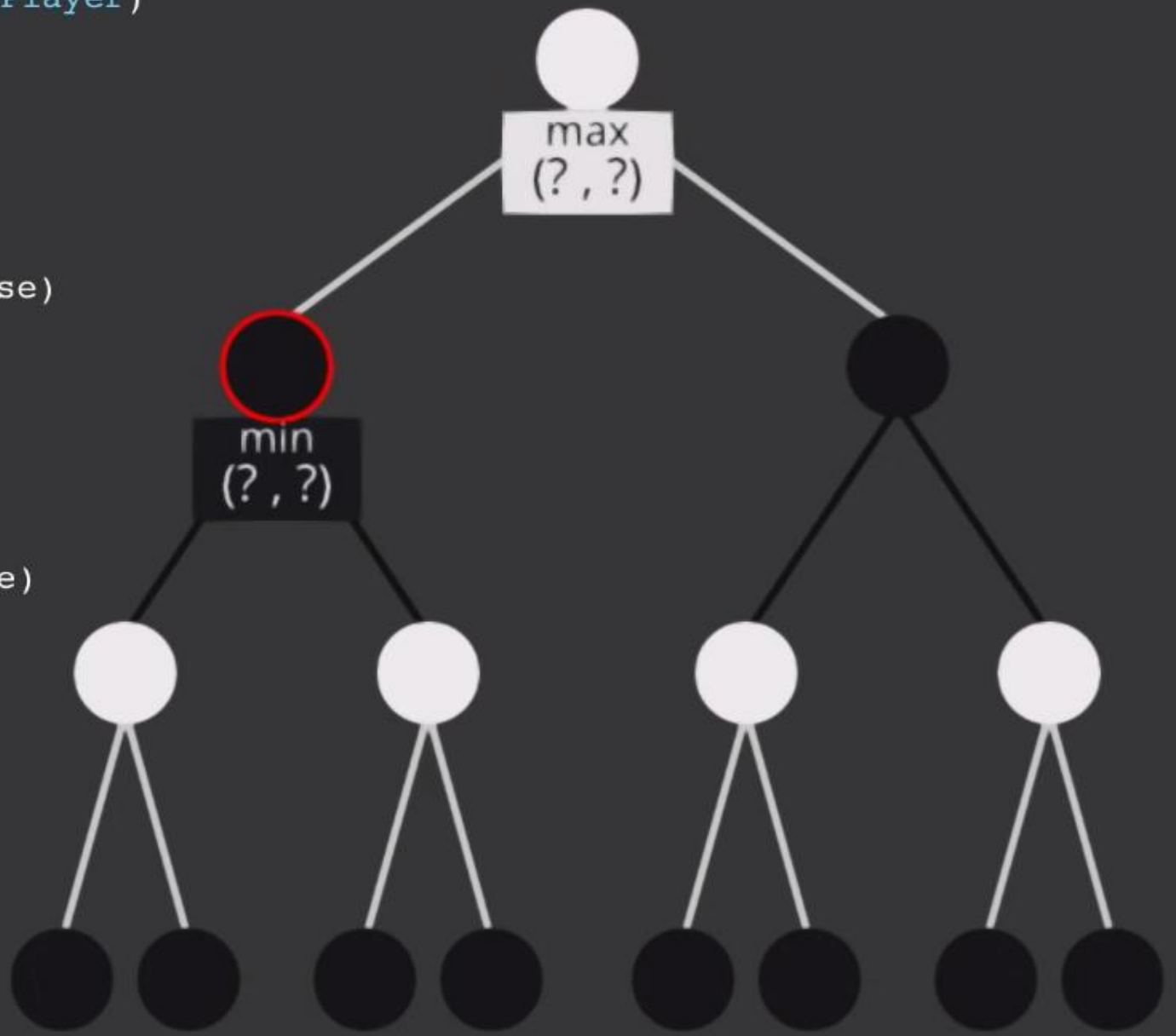
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



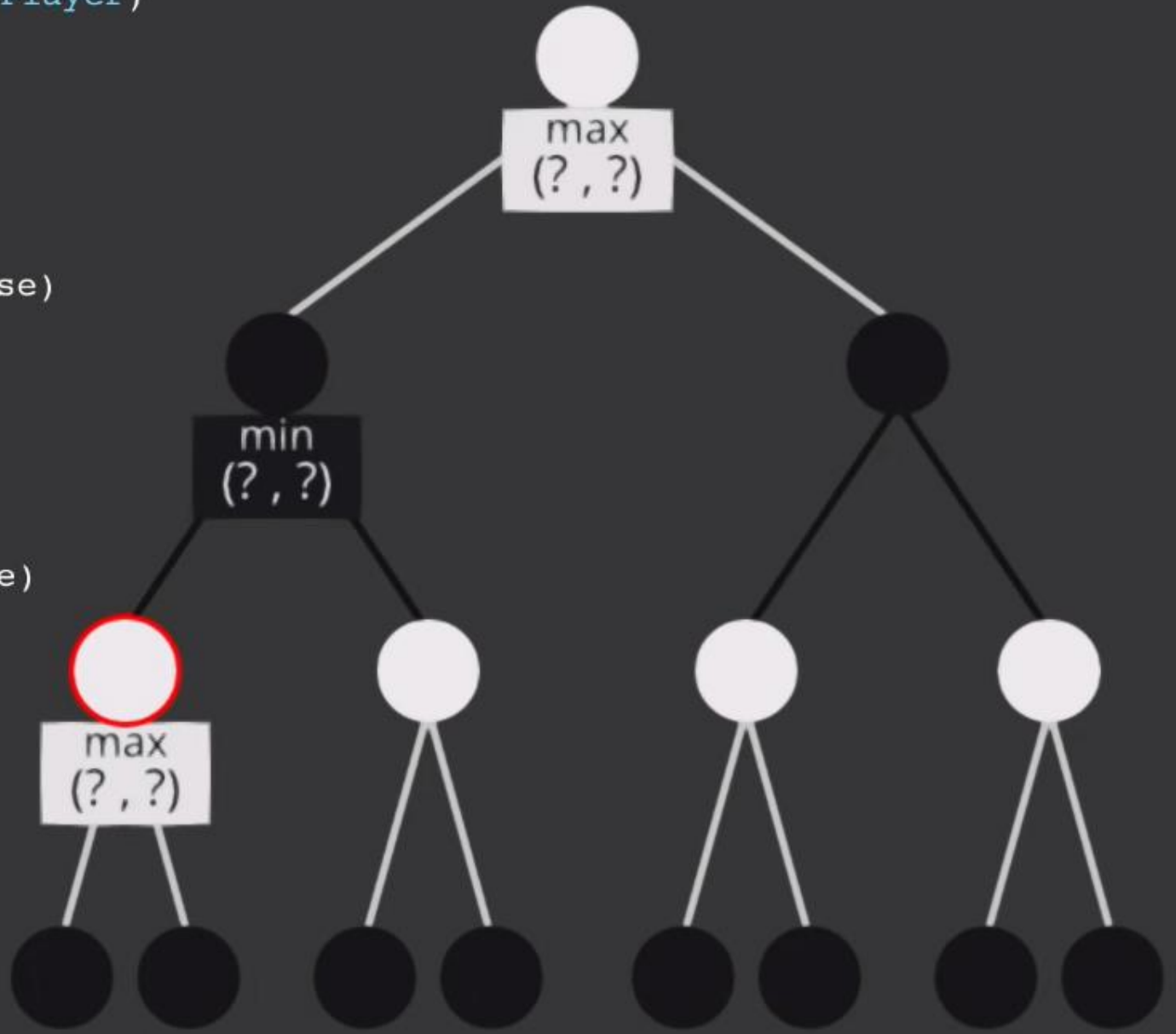
```
function minimax(position, depth, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, false)
         maxEval = max(maxEval, eval)
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, true)
         minEval = min(minEval, eval)
      return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



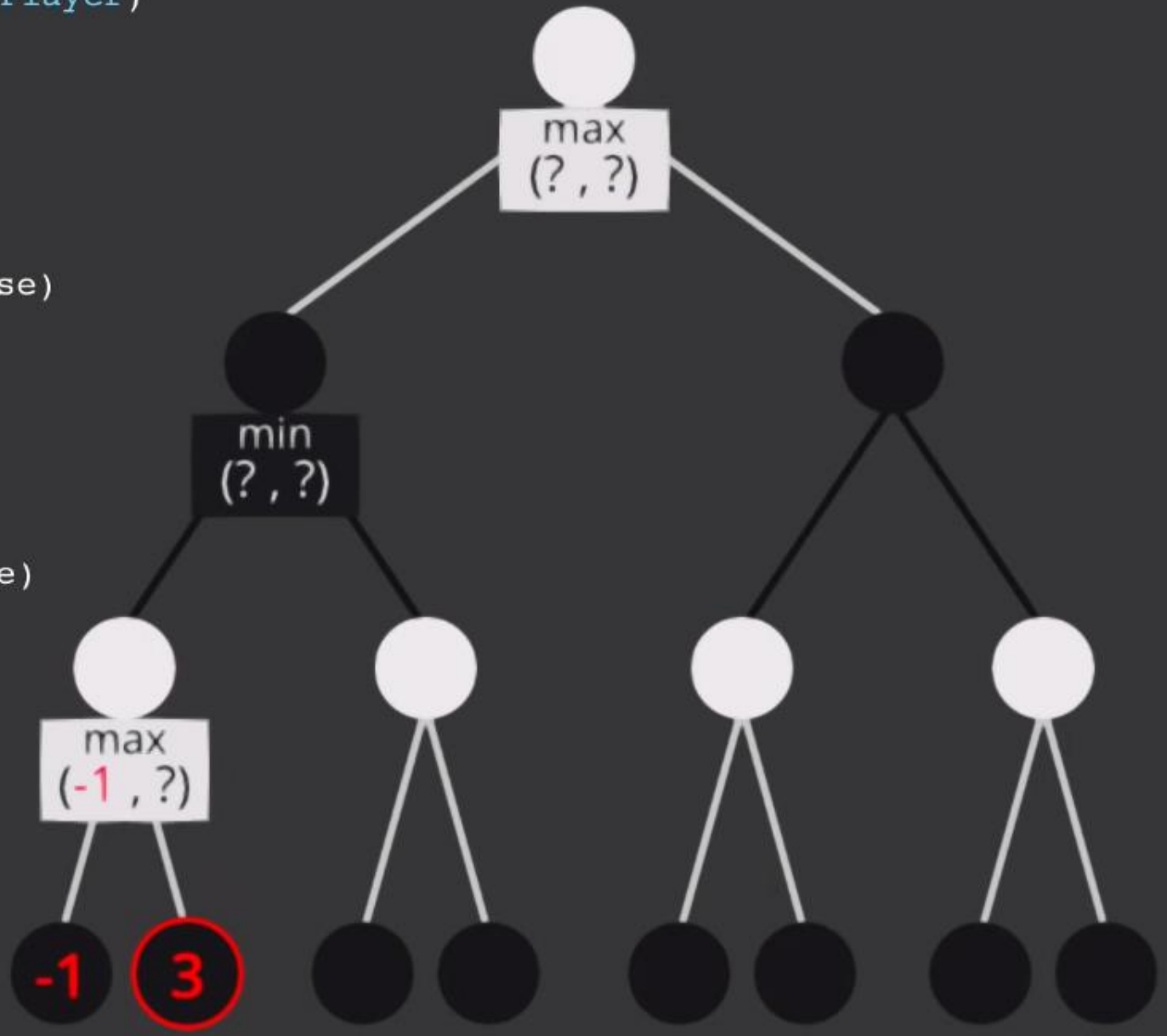
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



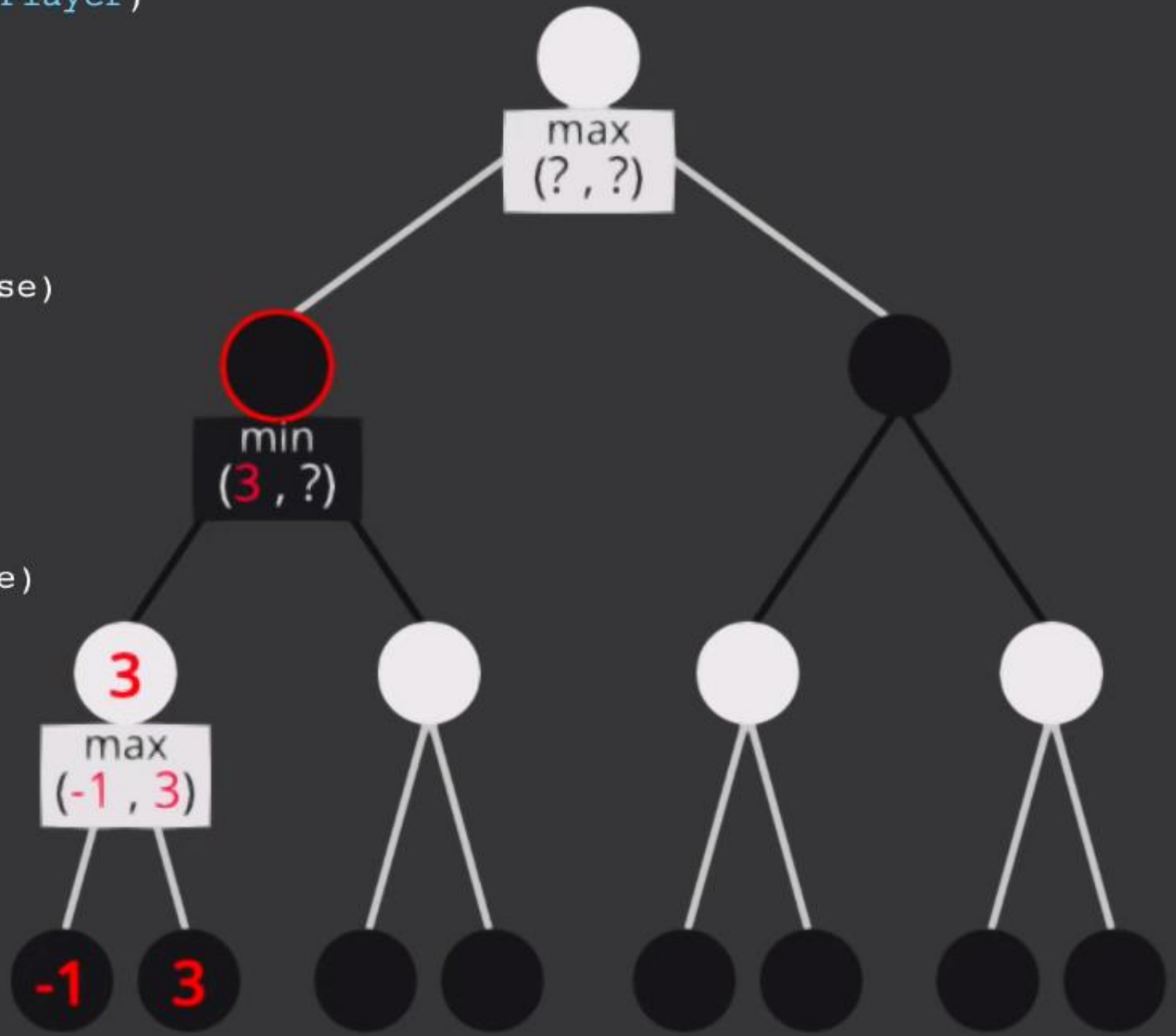
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



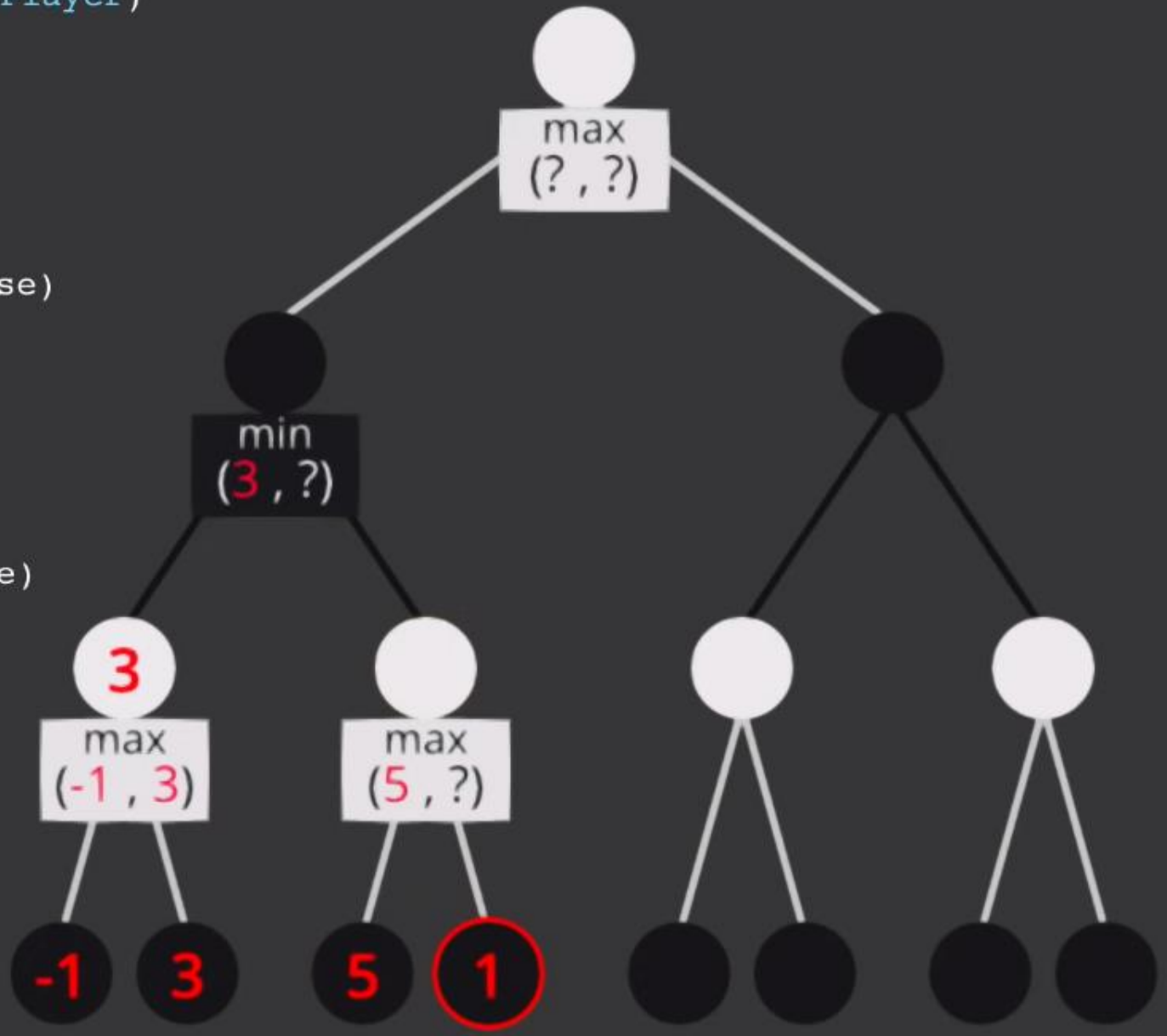
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



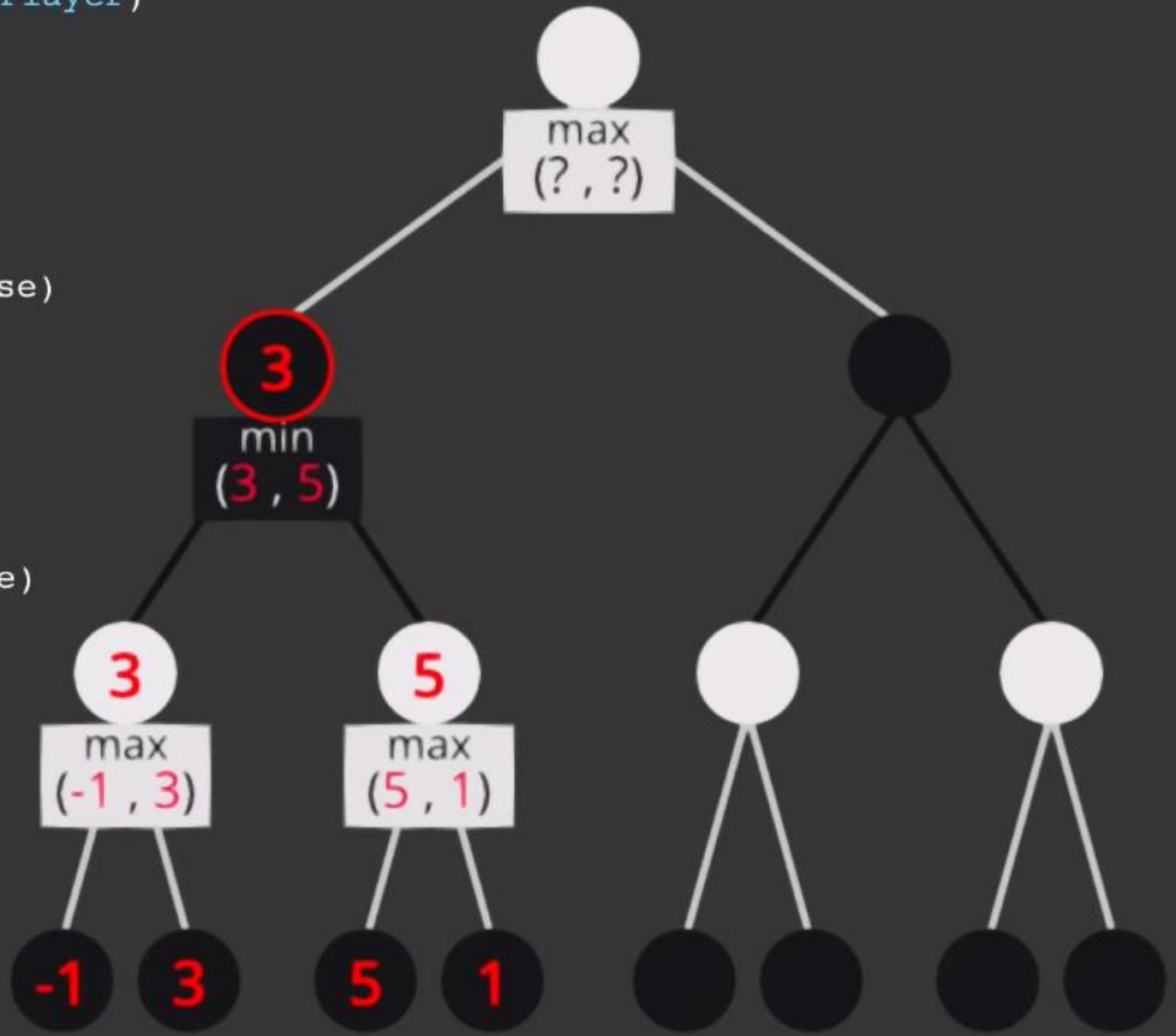
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



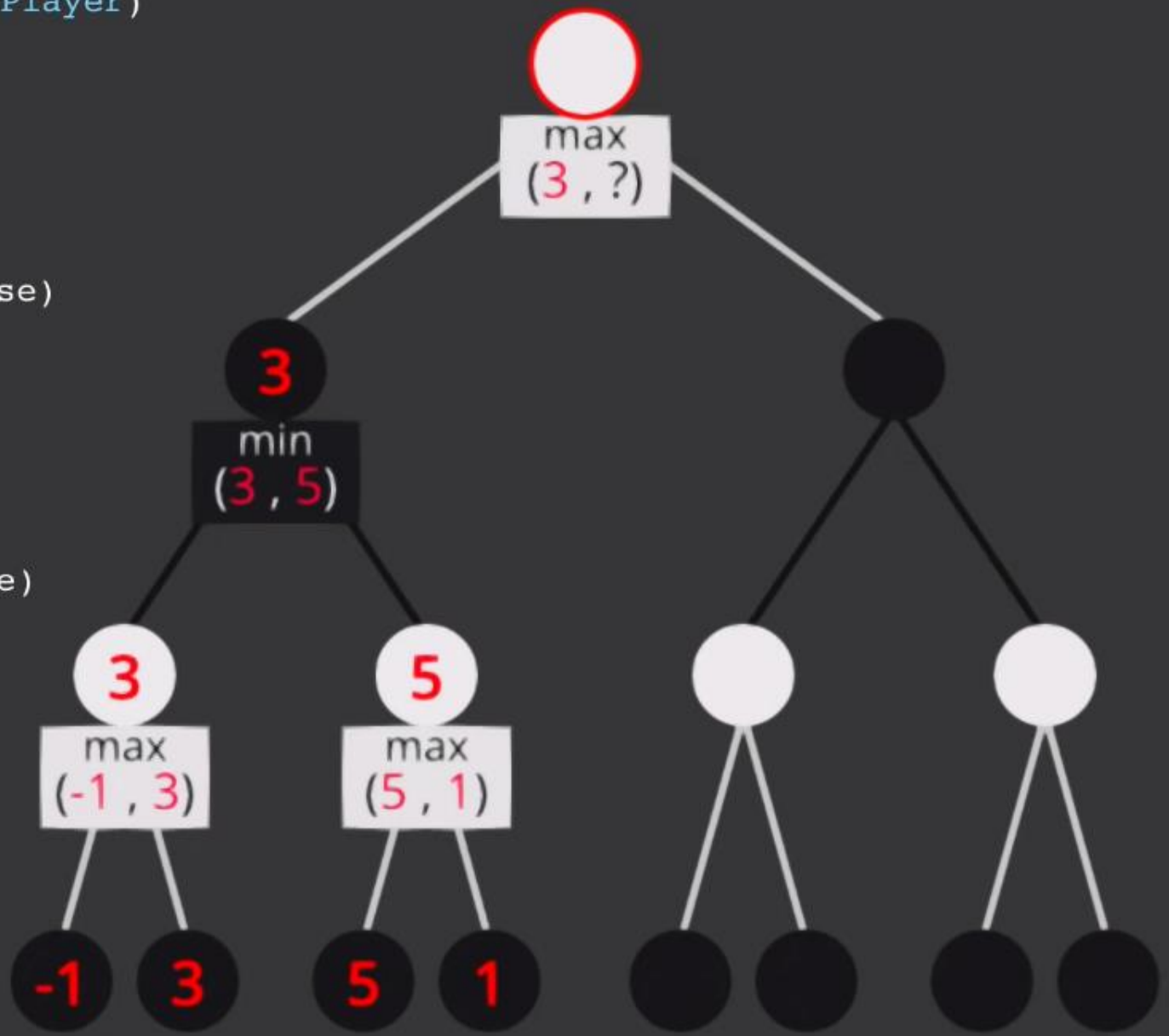
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



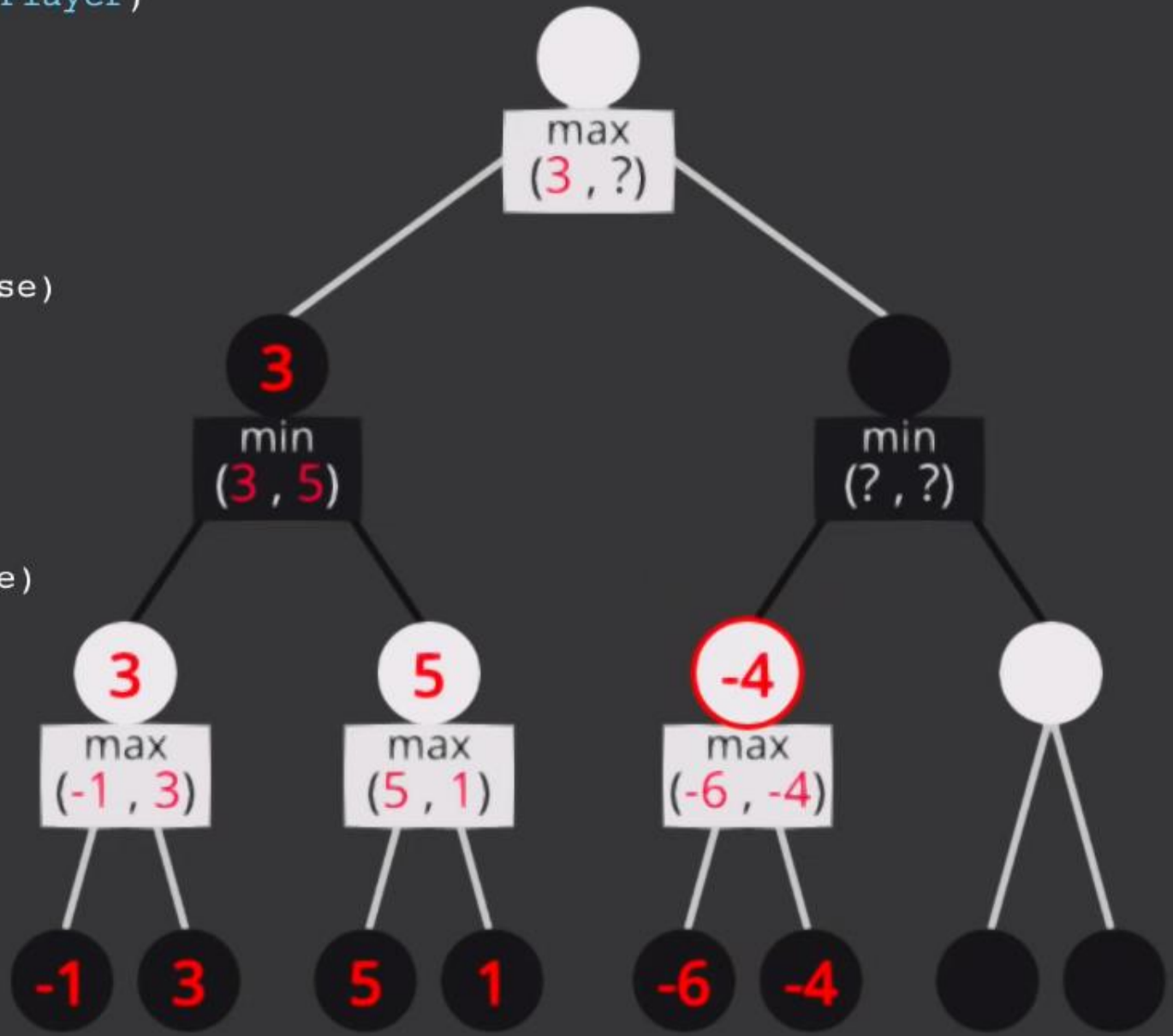
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



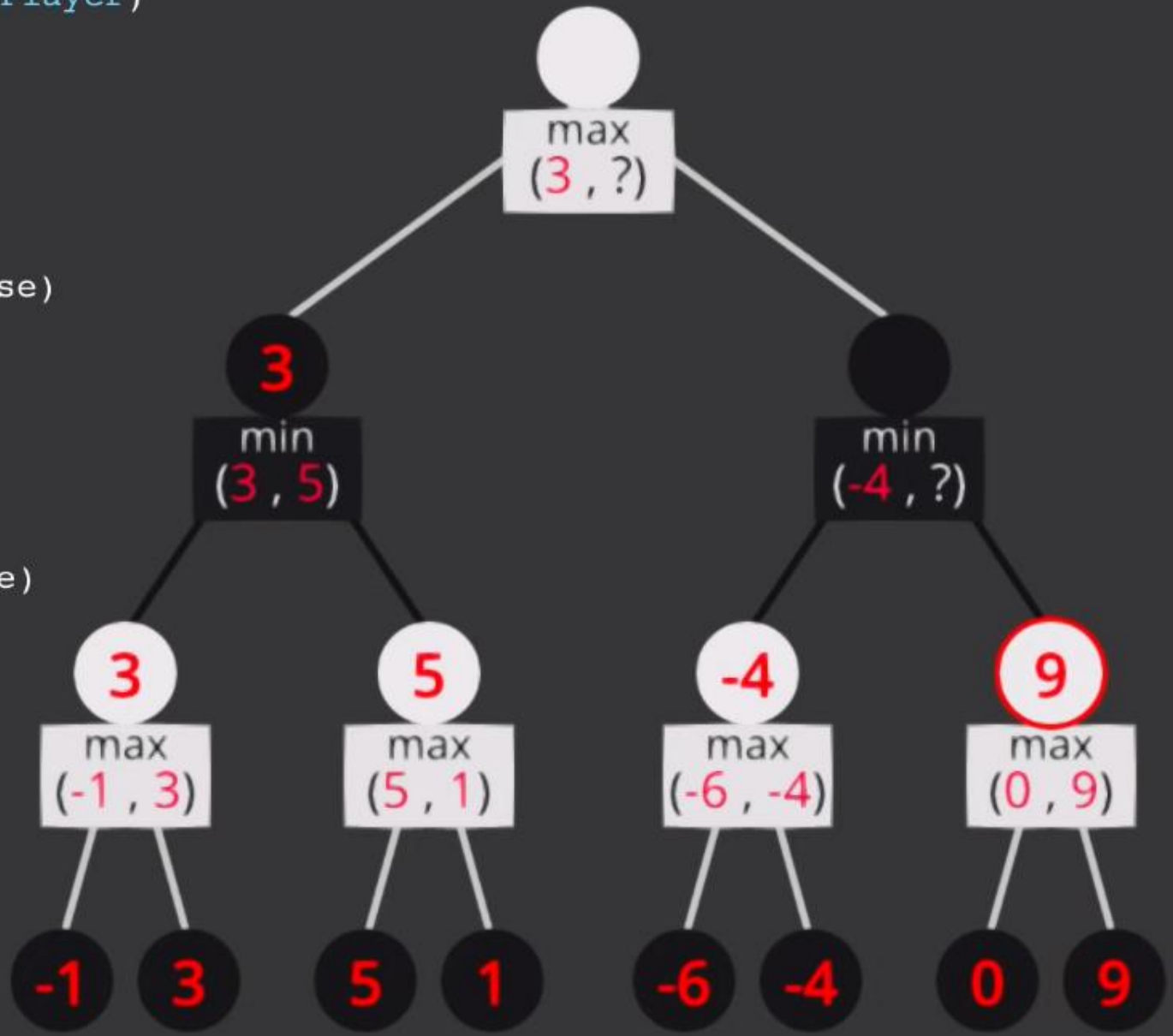
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```



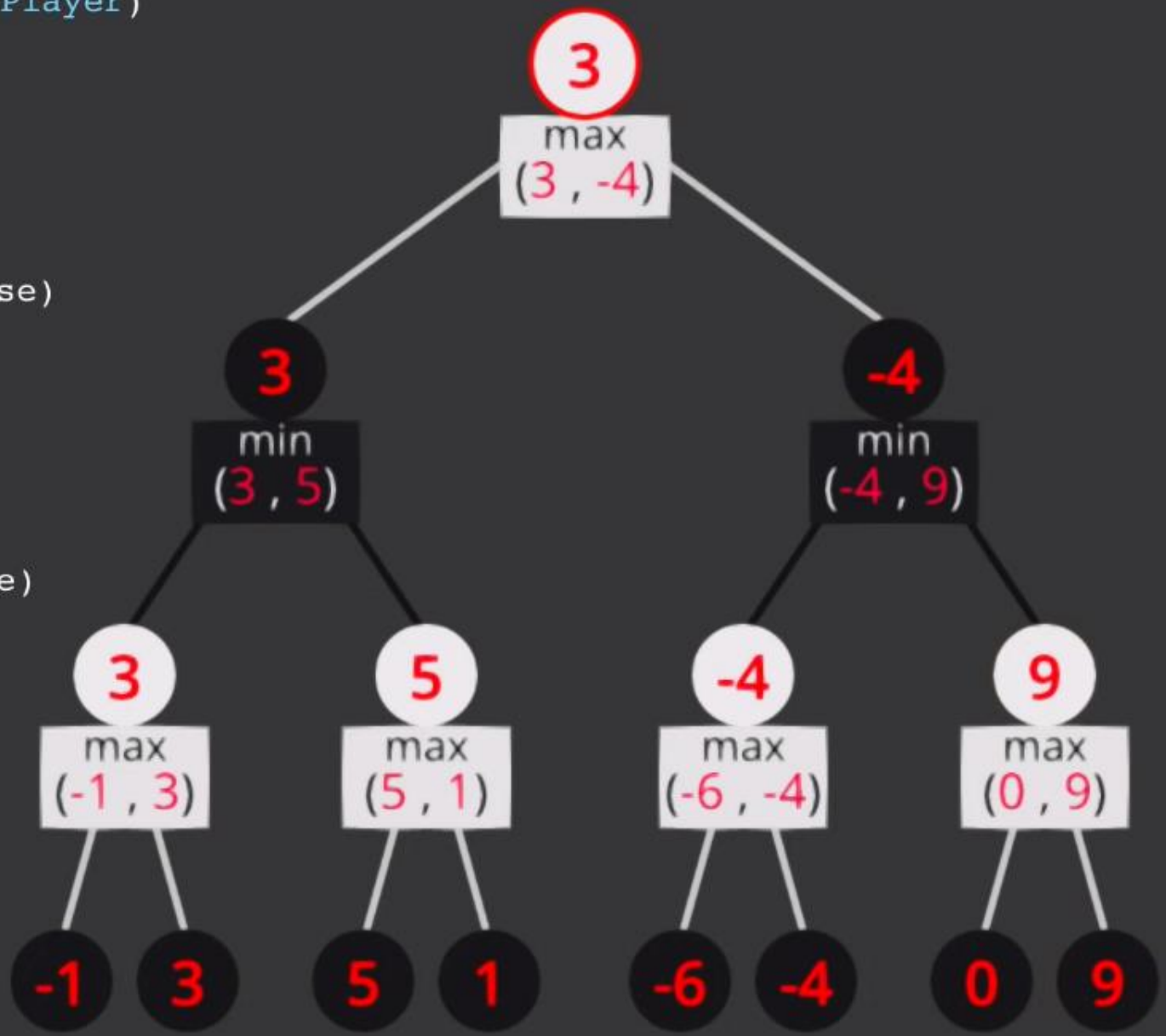
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
// initial call
minimax(currentPosition, 3, true)
```

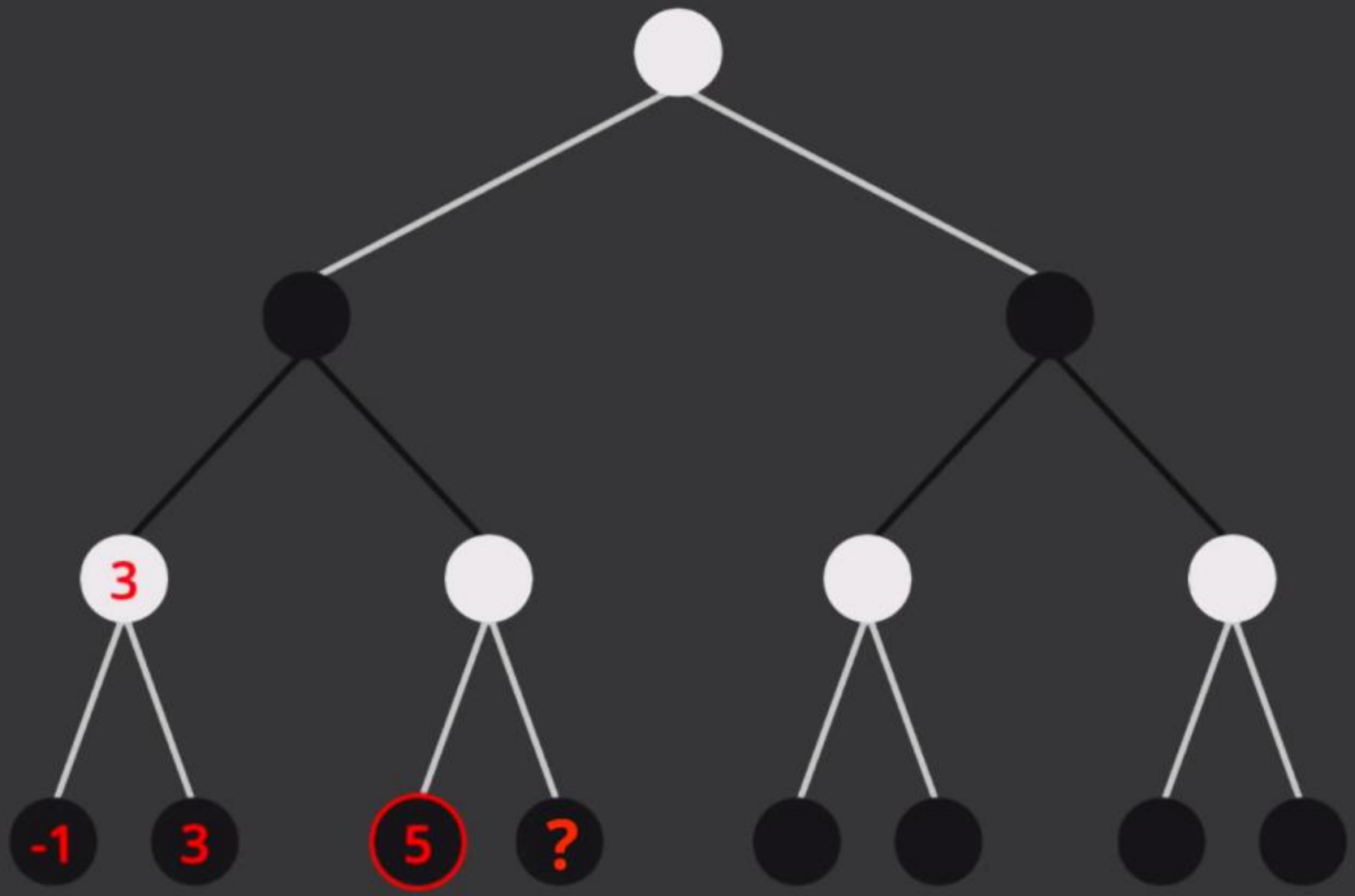


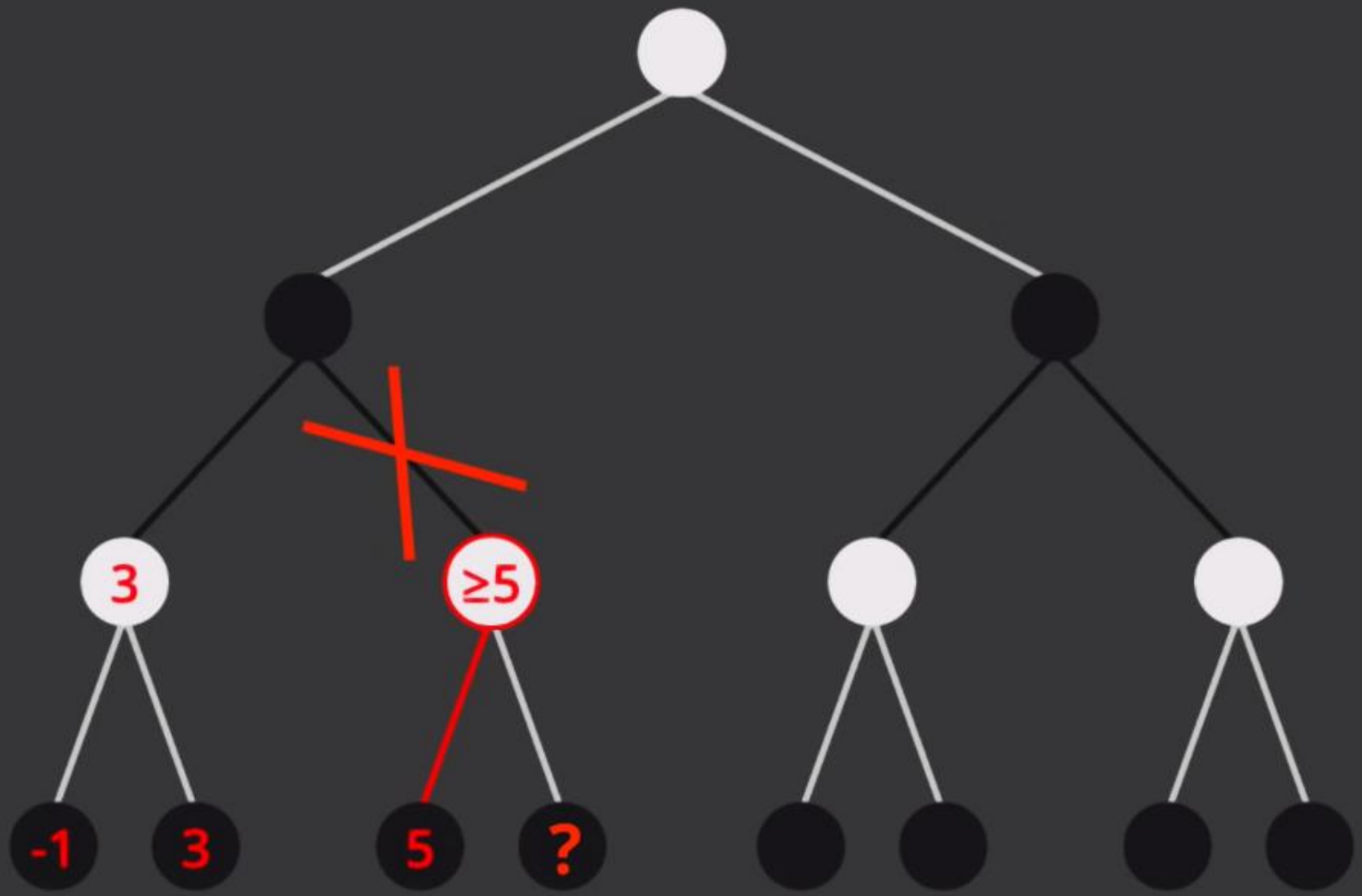
```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

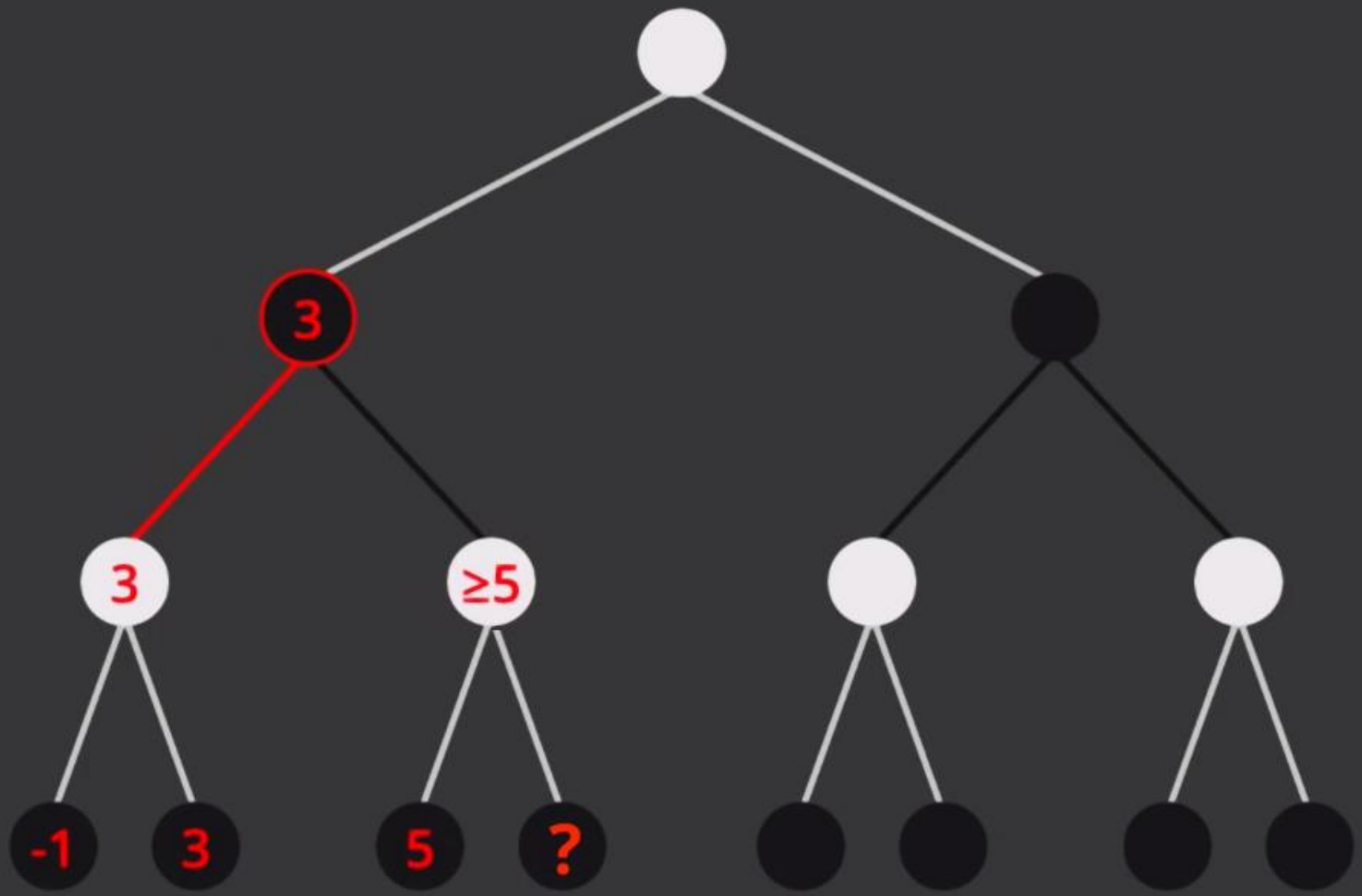
```
// initial call
minimax(currentPosition, 3, true)
```

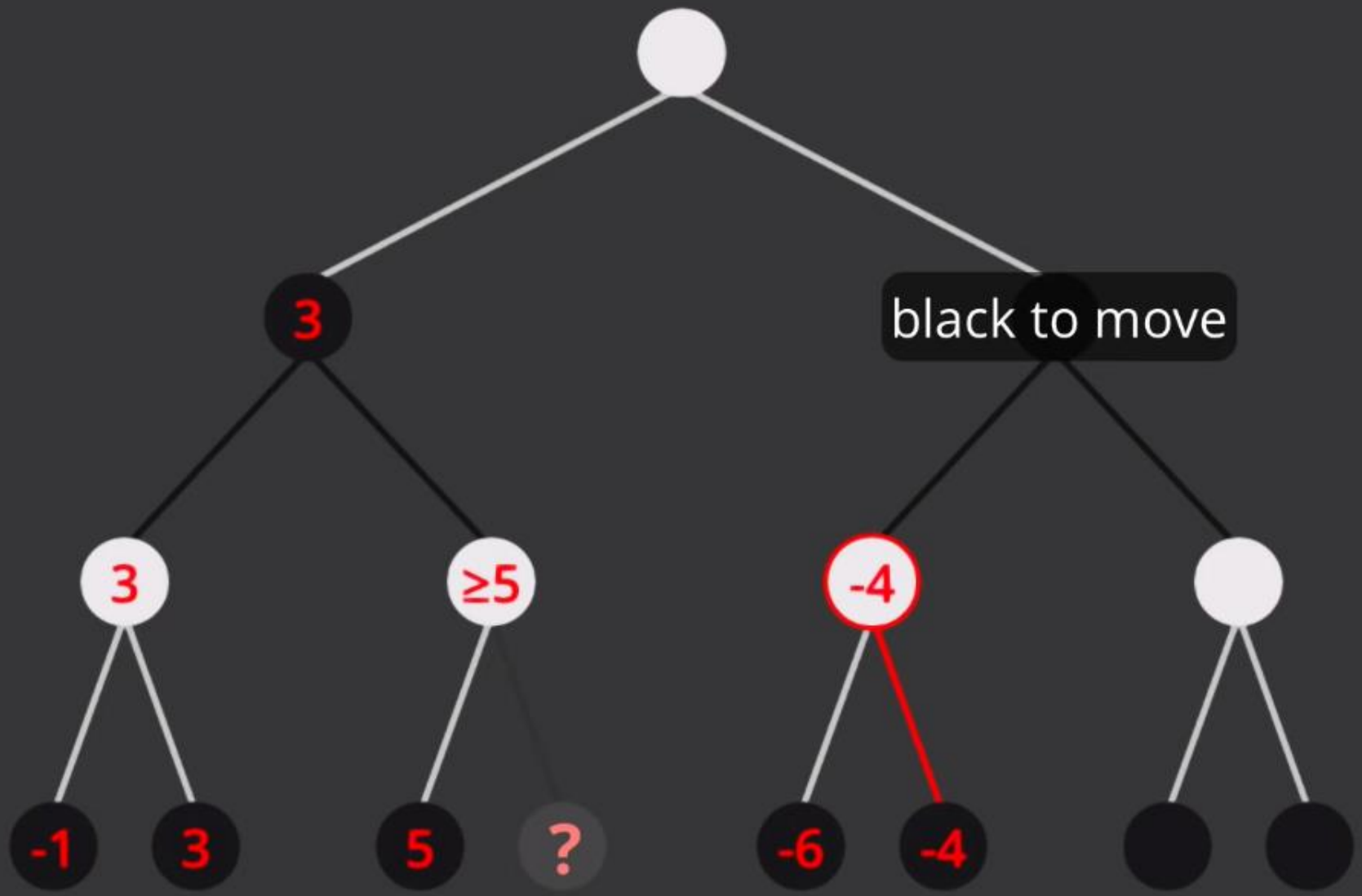


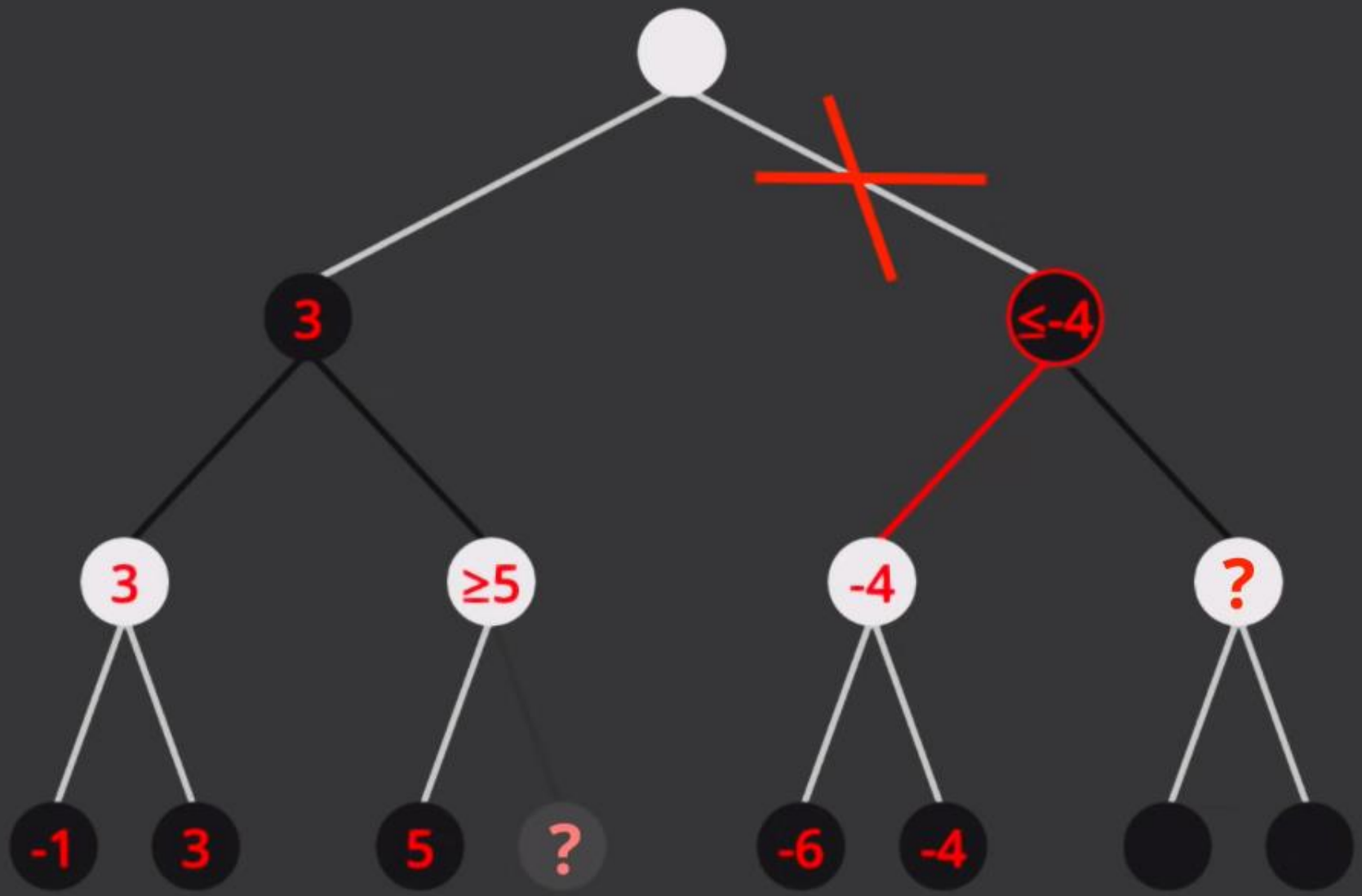
Alpha-Beta Pruning Example

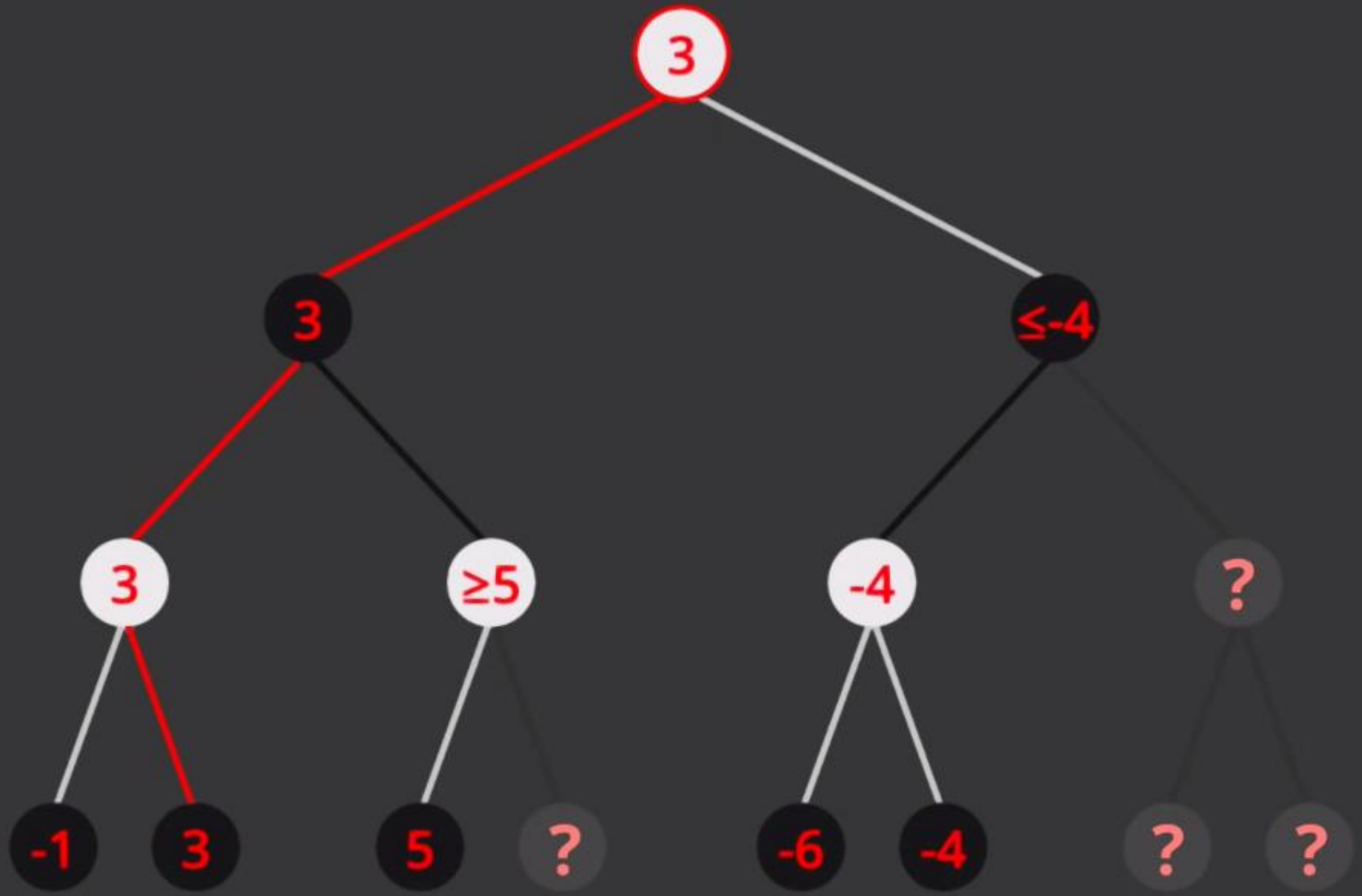

3
-1
3
5
?


3
≥5
-1
3
5
?

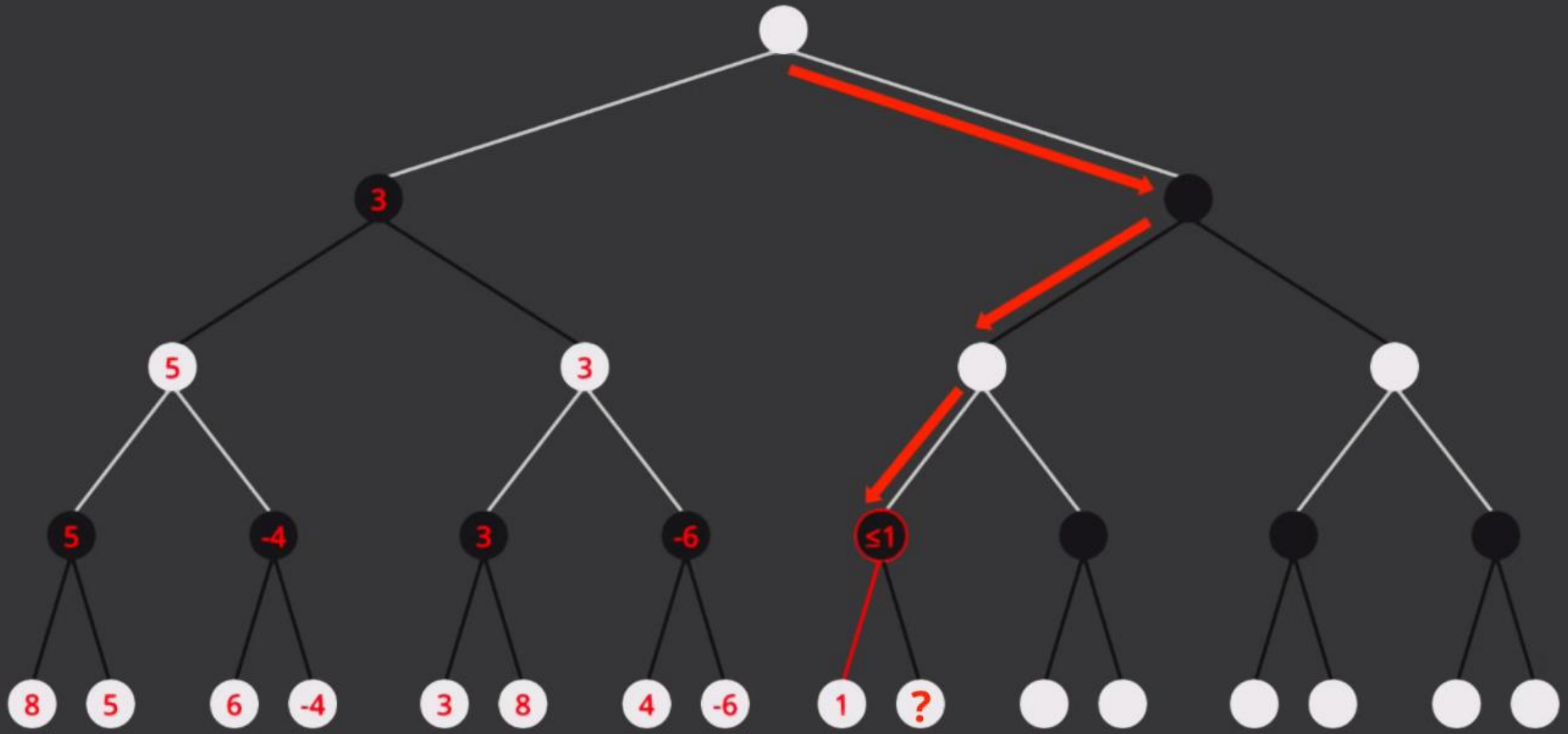

3
3
≥5
-1
3
5
?

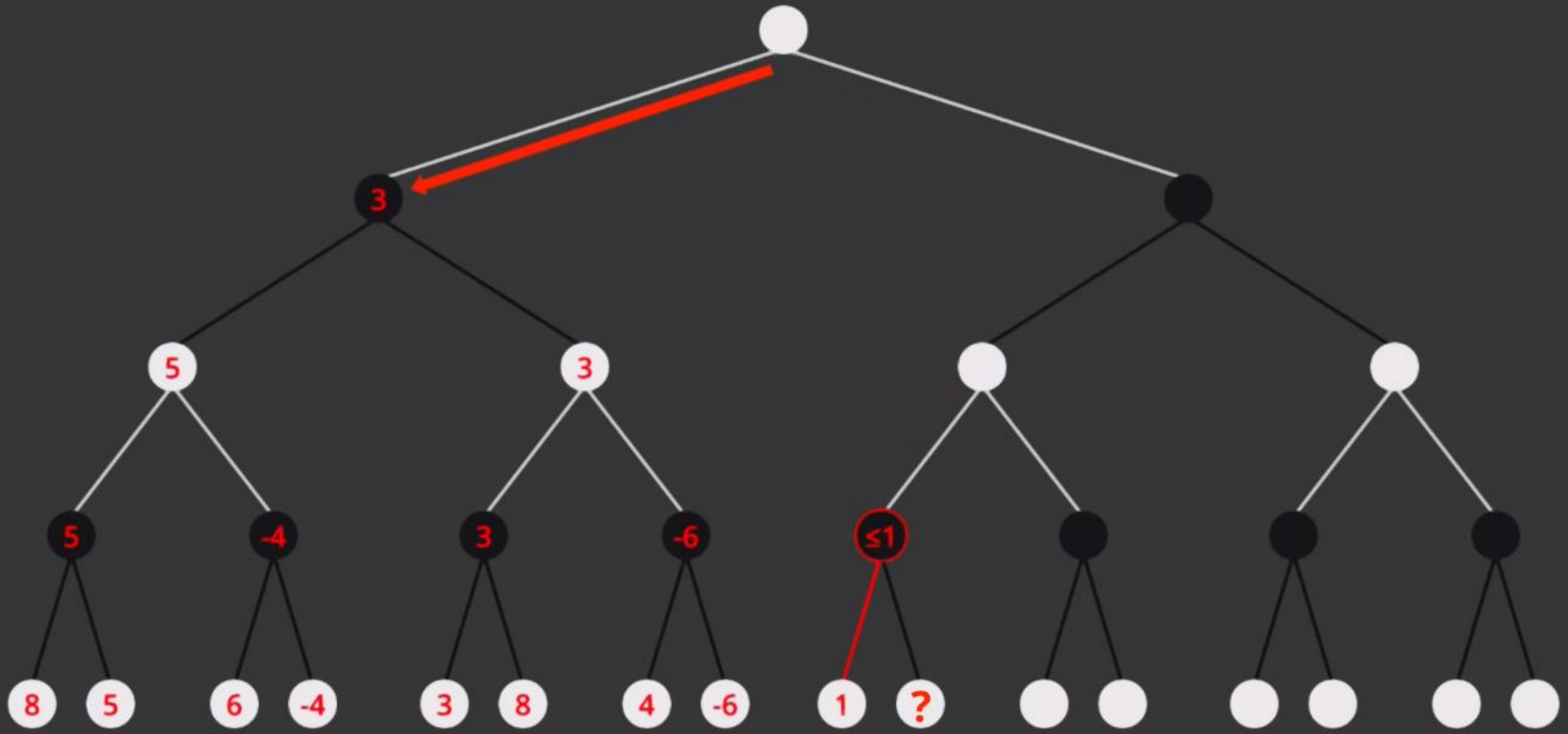

3
black to move
3
≥5
-4
-1
3
5
?
-6
-4

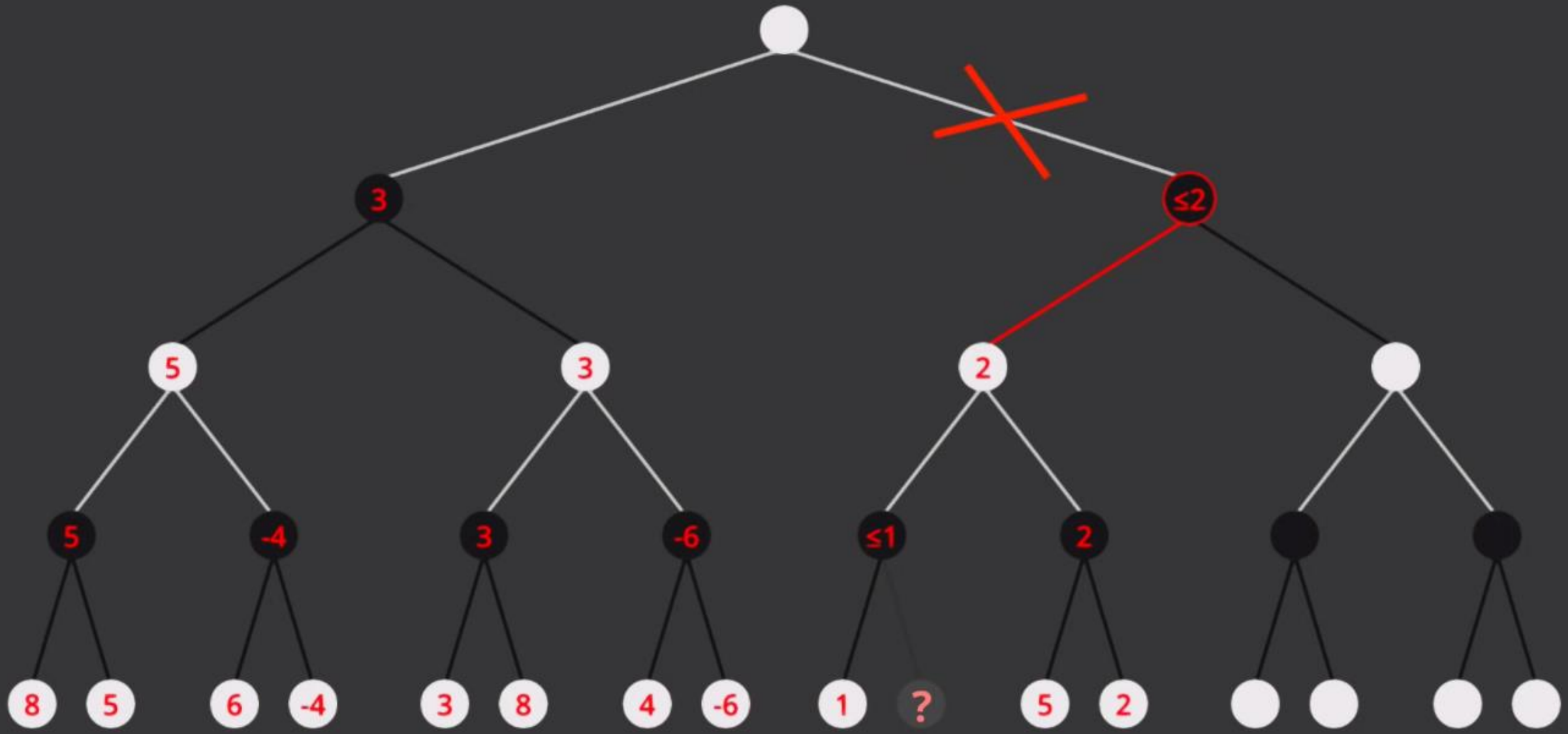

3
≤-4
3
≥5
-4
?
-1
3
5
?
-6
-4

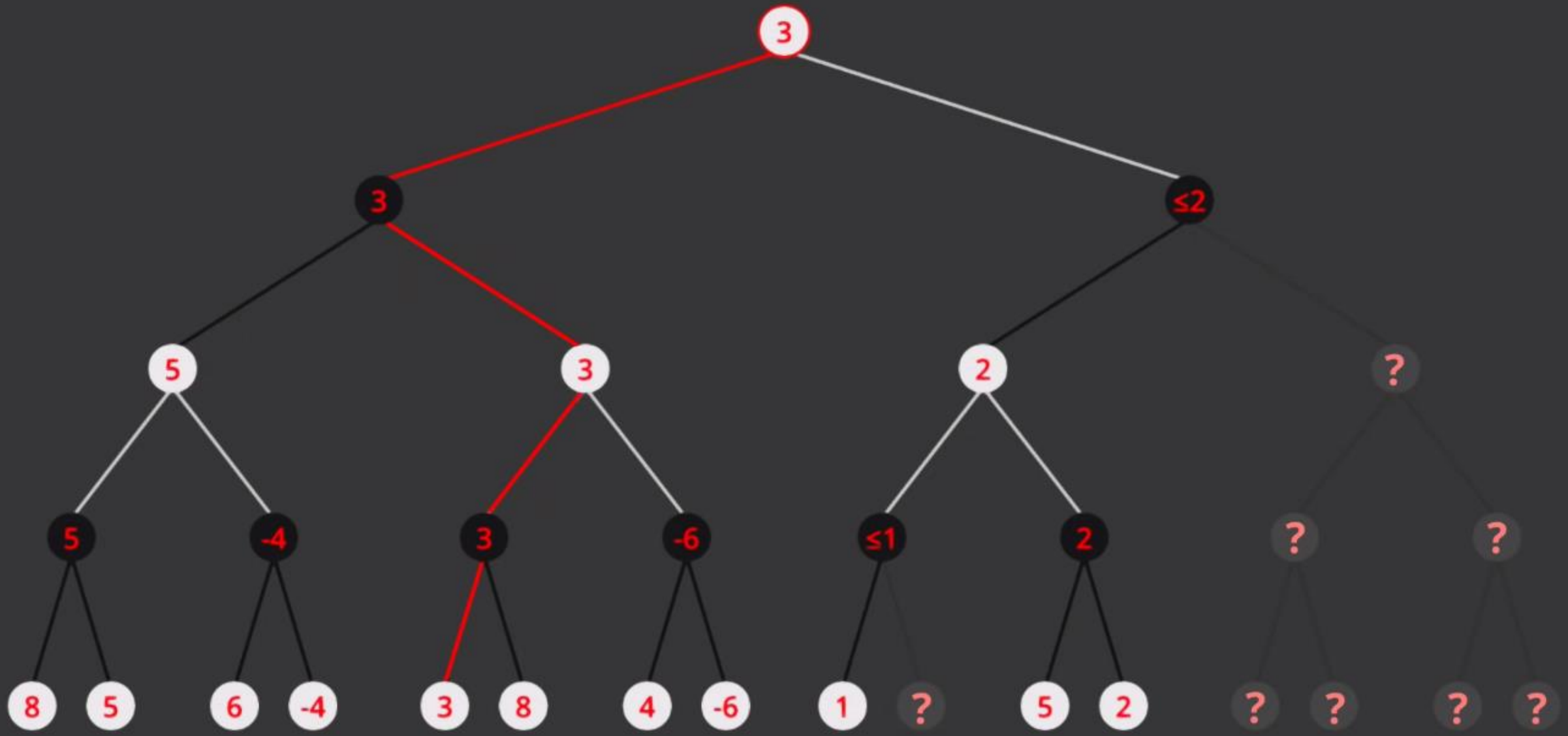

3
3
≤-4
3
≥5
-4
?
-1
3
5
?
-6
-4
?
?

Alpha Beta Pruning Example


3
5
3
5
-4
3
-6
≤1
8
5
6
-4
3
8
4
-6
1
?


3
5
3
5
-4
3
-6
≤1
8
5
6
-4
3
8
4
-6
1
?


3
≤2
5
3
2
5
-4
3
-6
≤1
2
8
5
6
-4
3
8
4
-6
1
?
5
2


3
3
≤2
5
3
2
?
5
-4
3
-6
≤1
2
?
?
8
5
6
-4
3
8
4
-6
1
?
5
2
?
?
?
?

From Minimax to Alpha Beta Pruning Algorithm

Steps to create alpha beta algorithm from Minimax Algorithm

```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
        return minEval
```

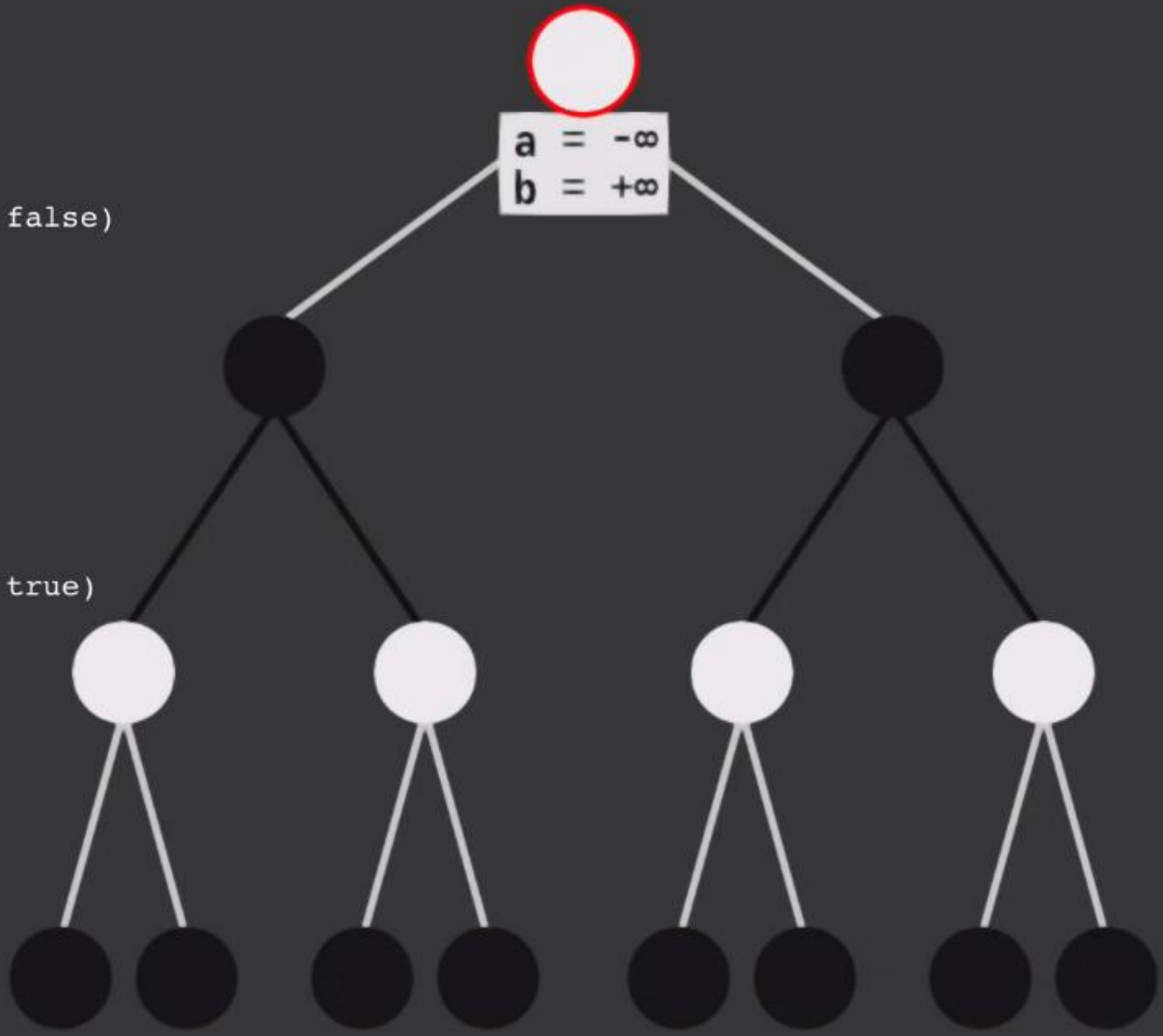
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
        return minEval
```

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha
                break
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
        return minEval
```

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha
                break
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha
                break
        return minEval
```

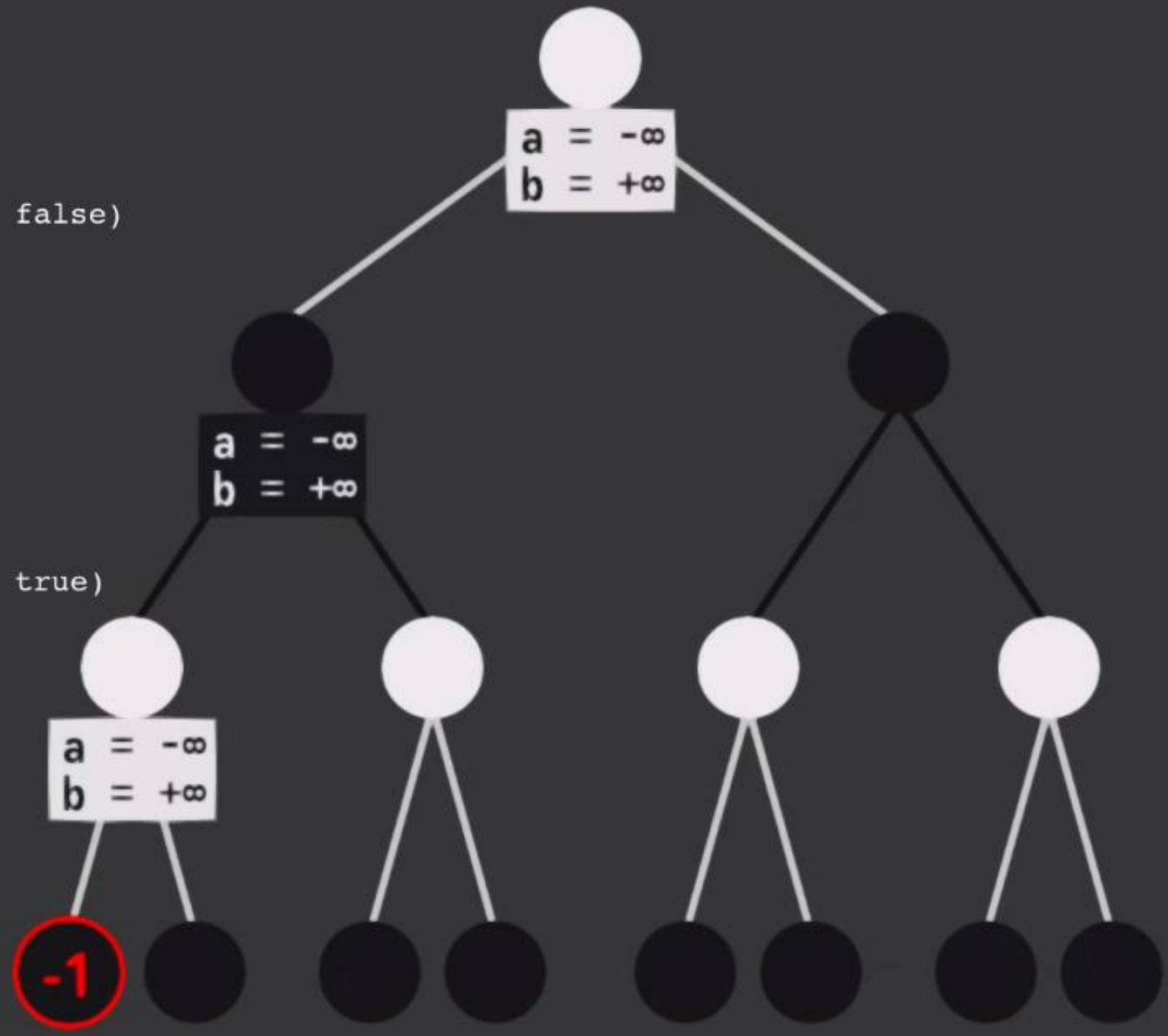
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



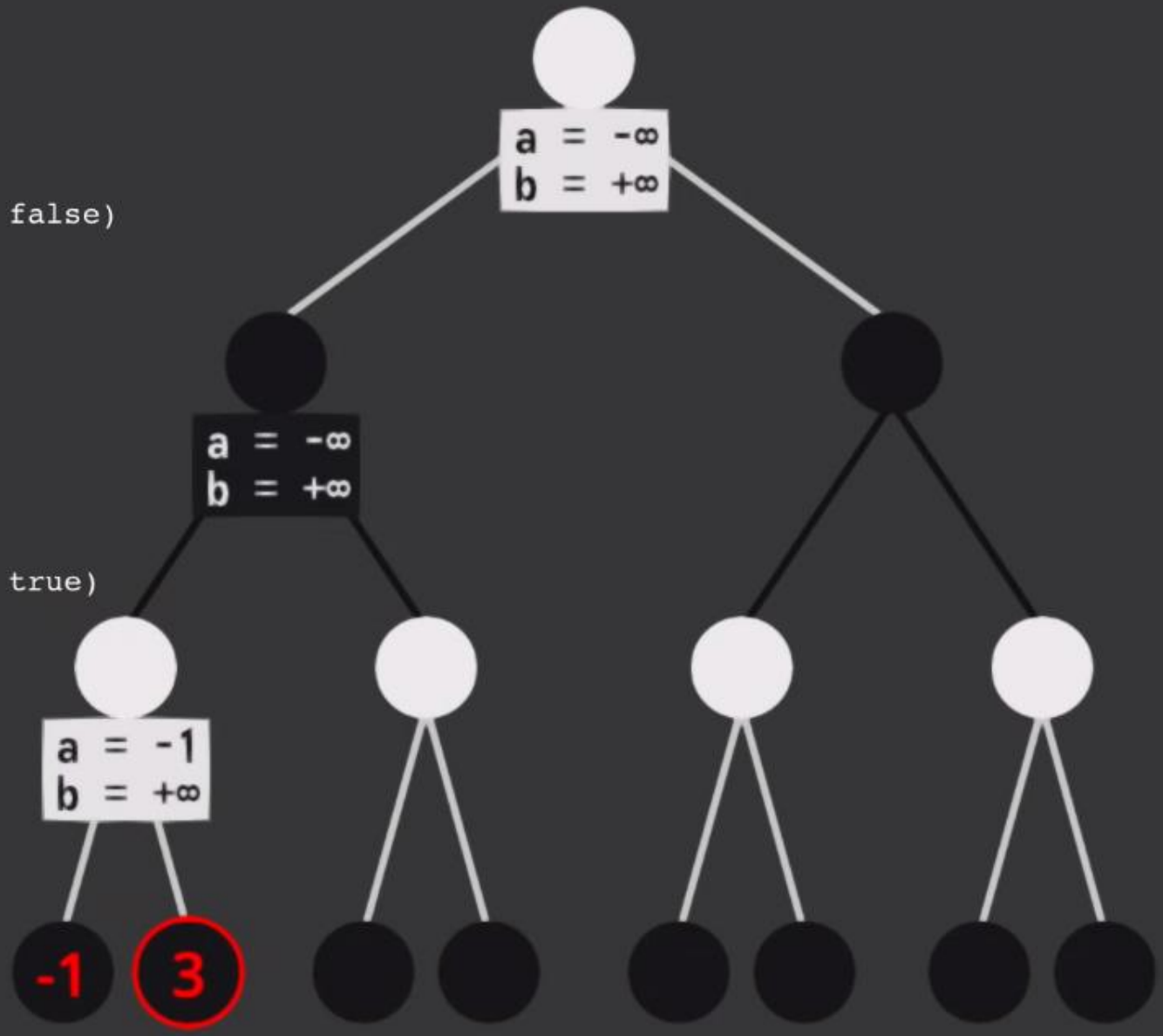
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



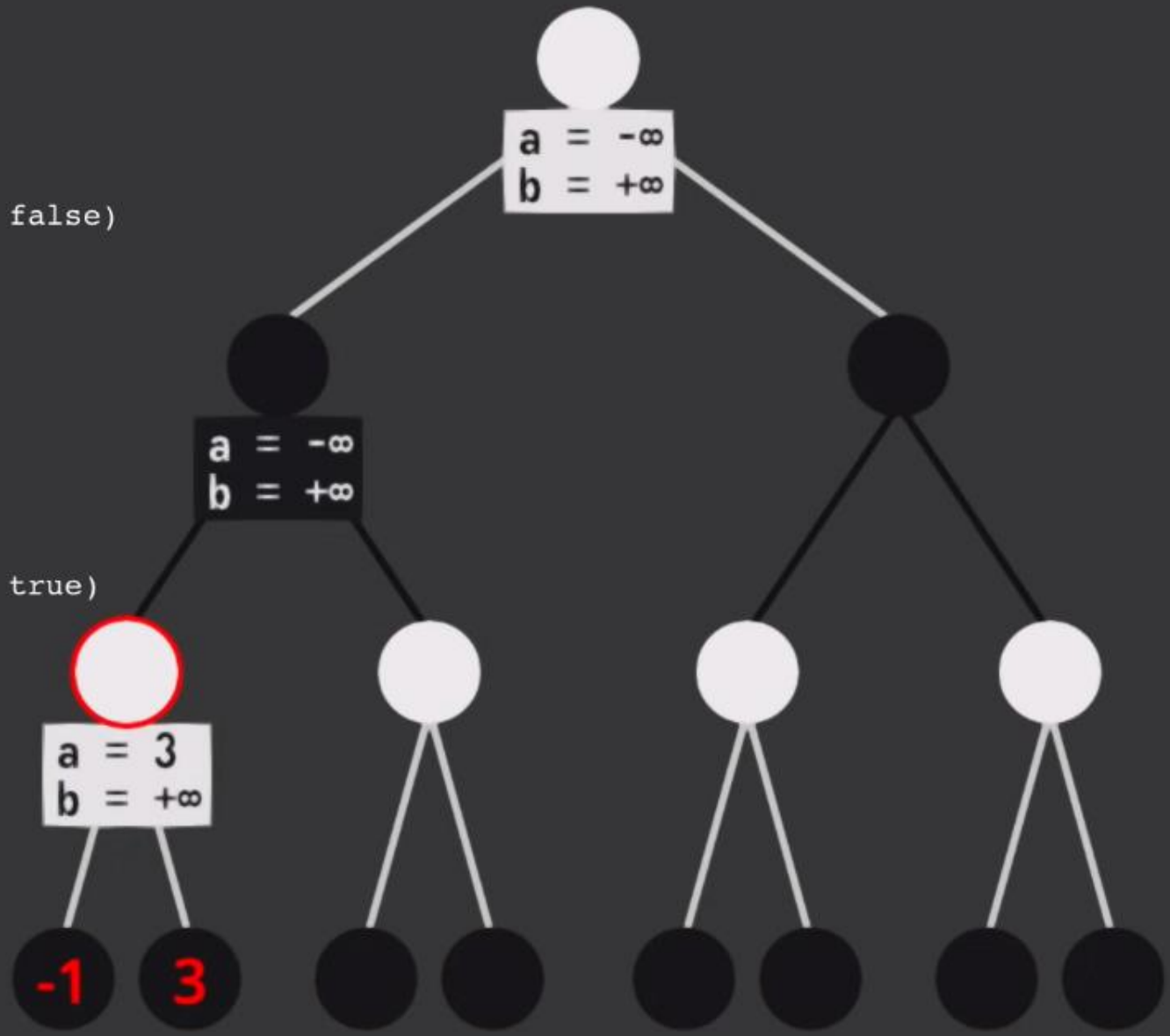
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



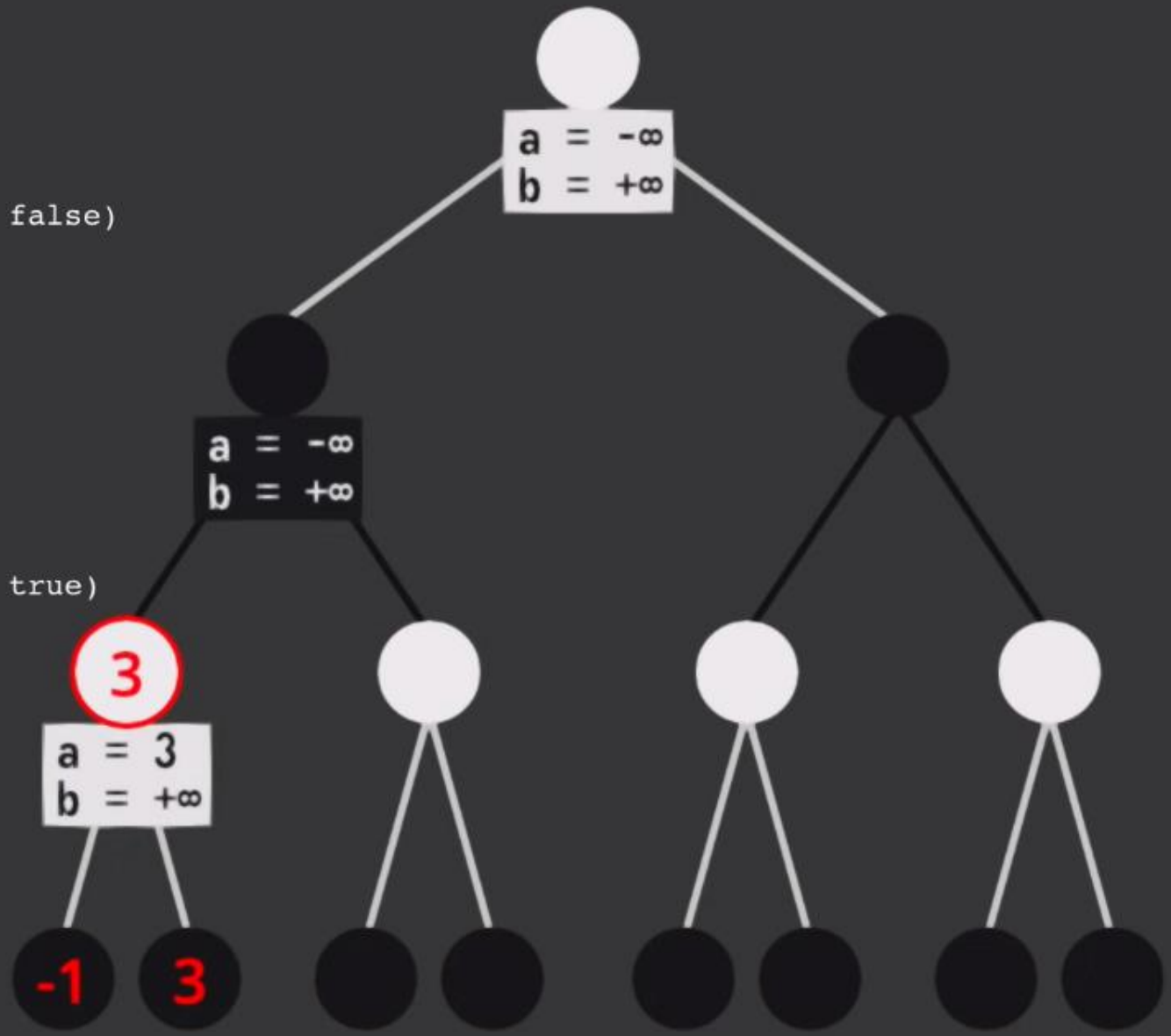
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



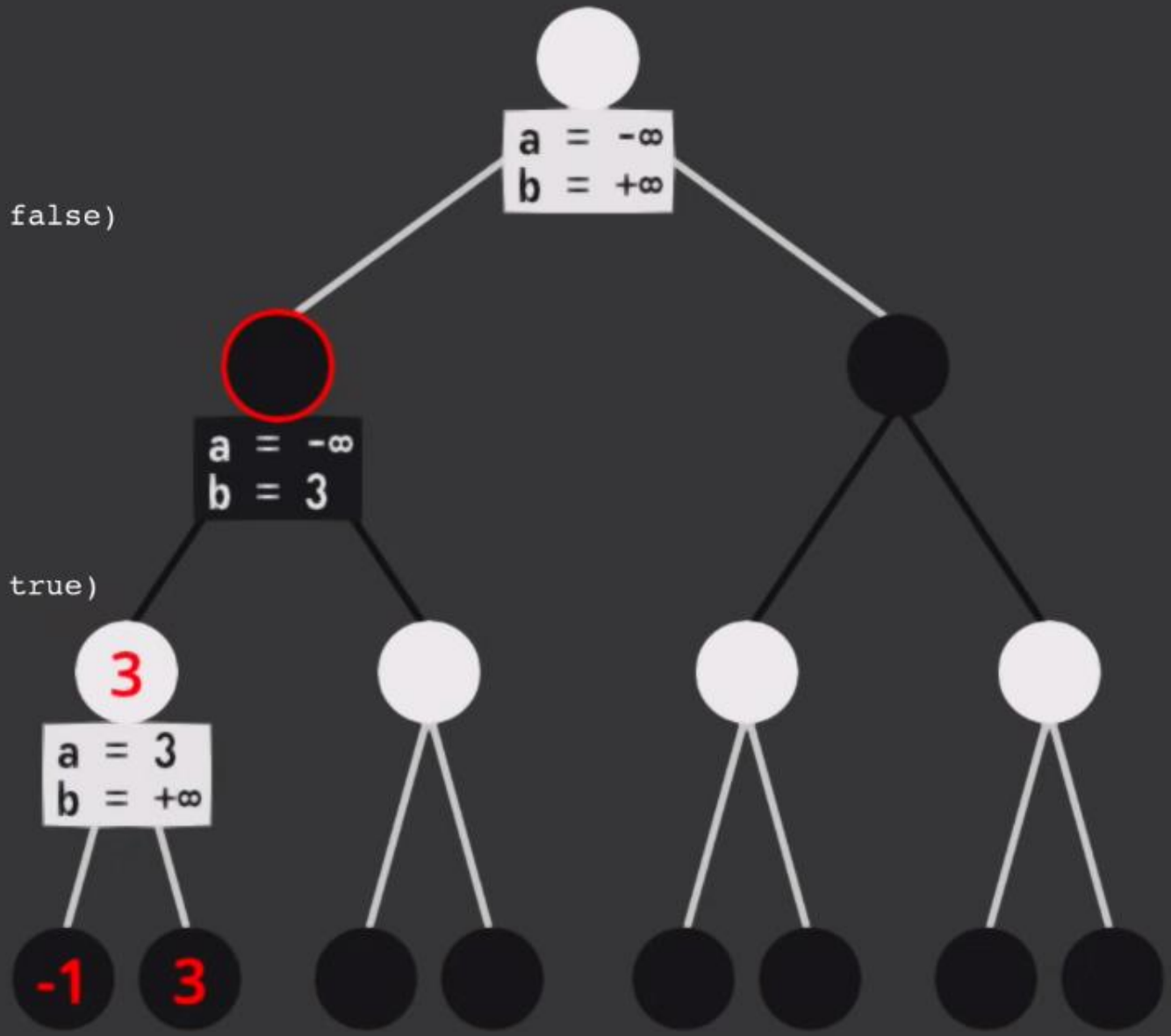
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



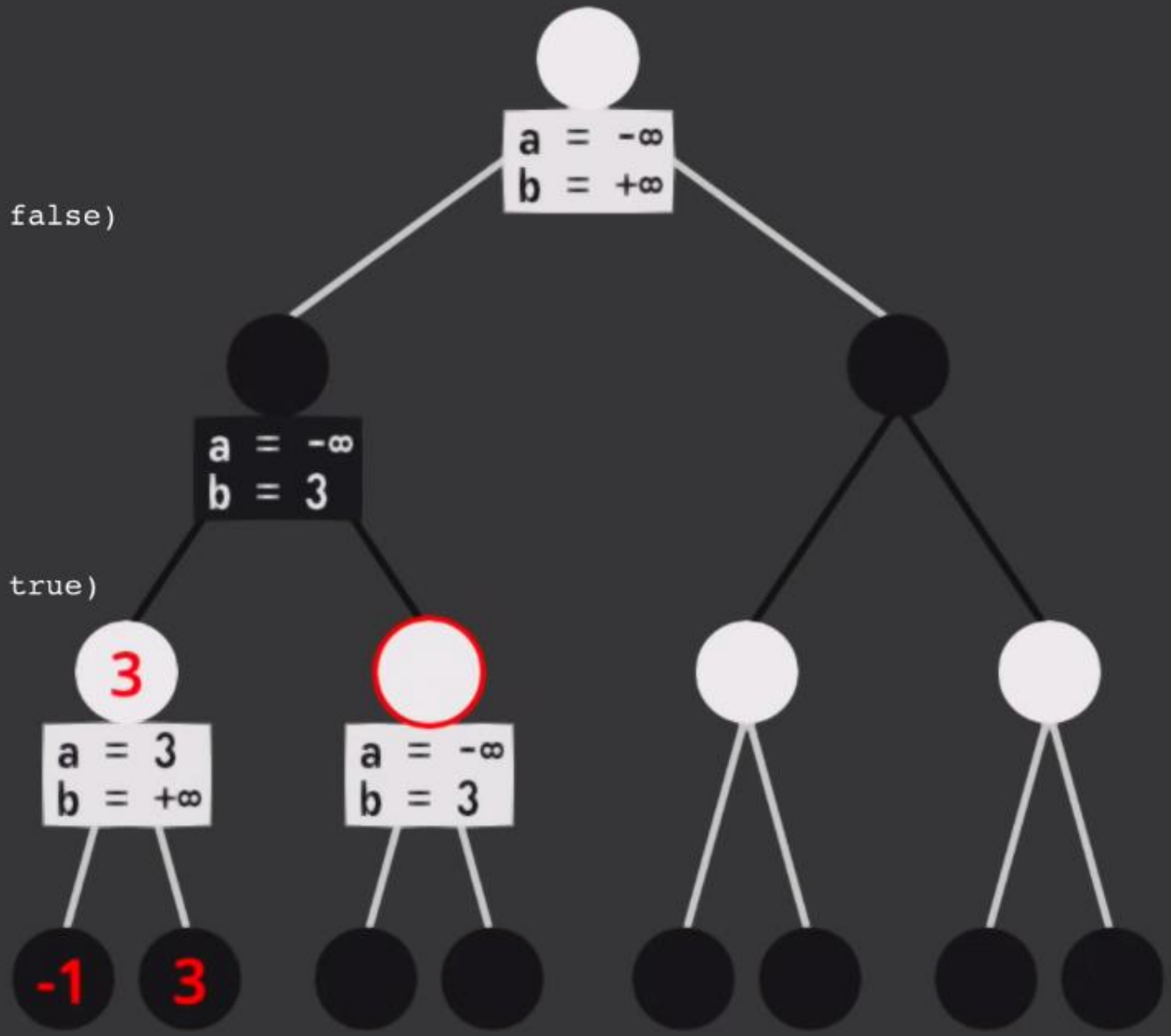
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



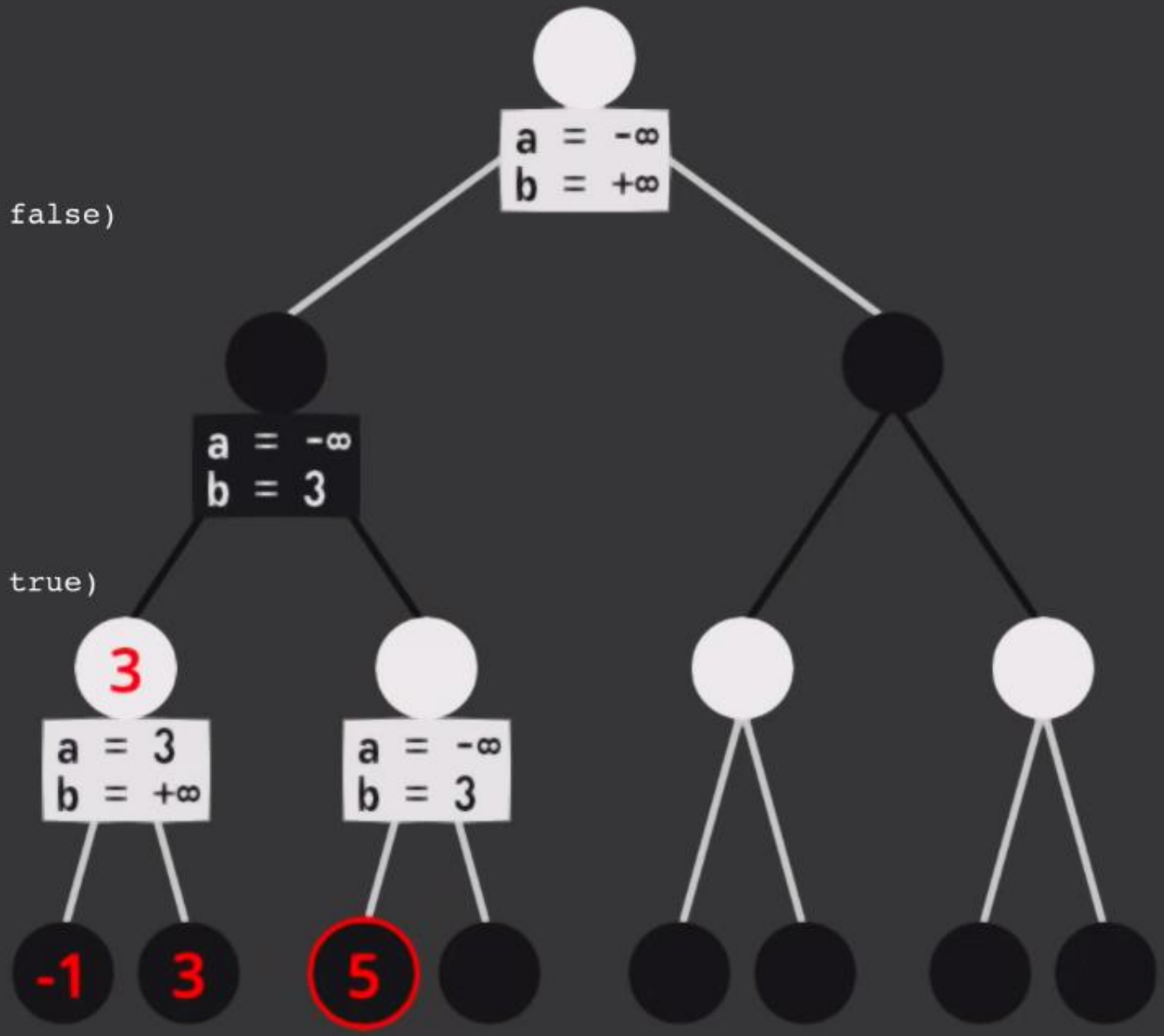
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



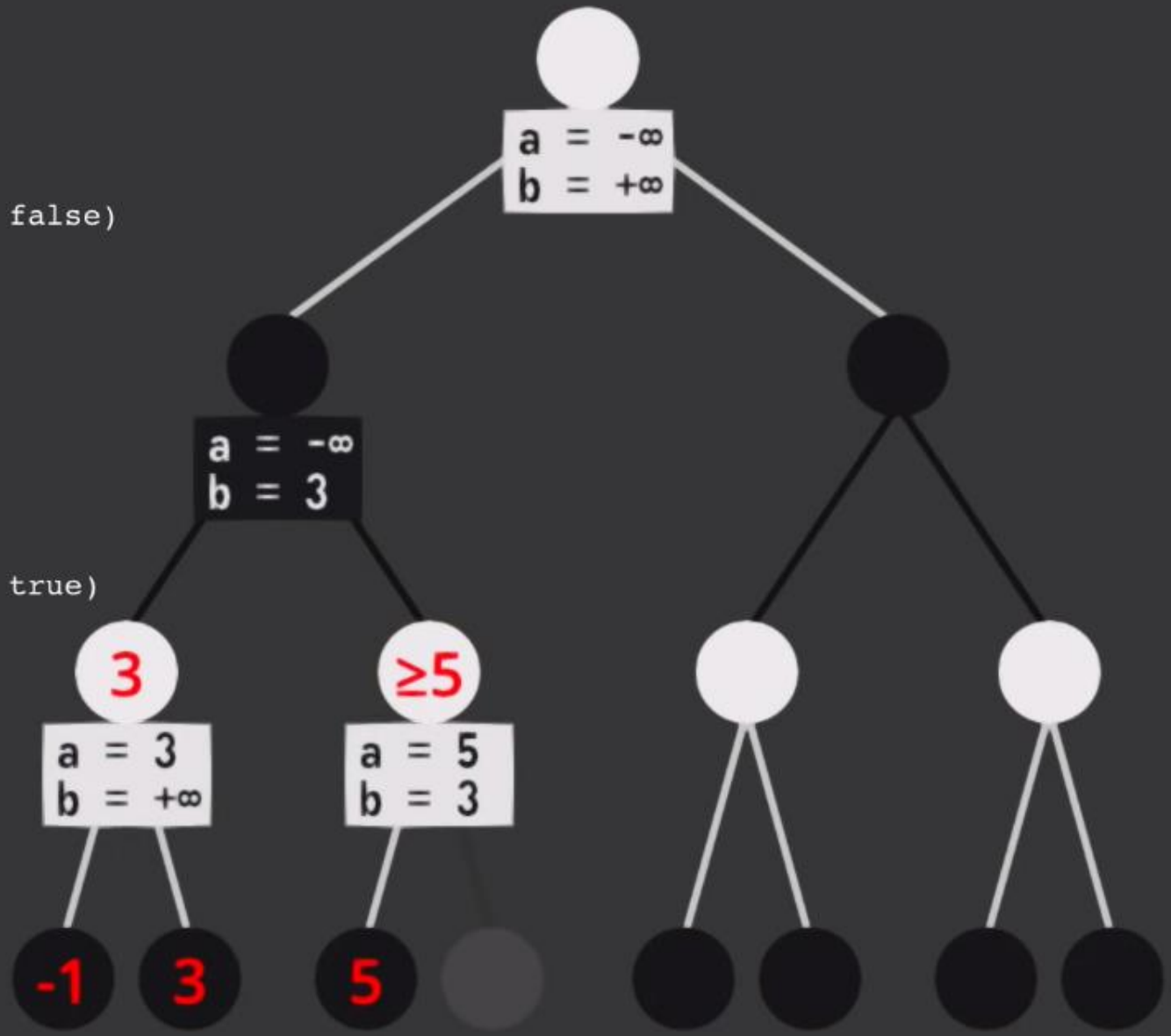
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



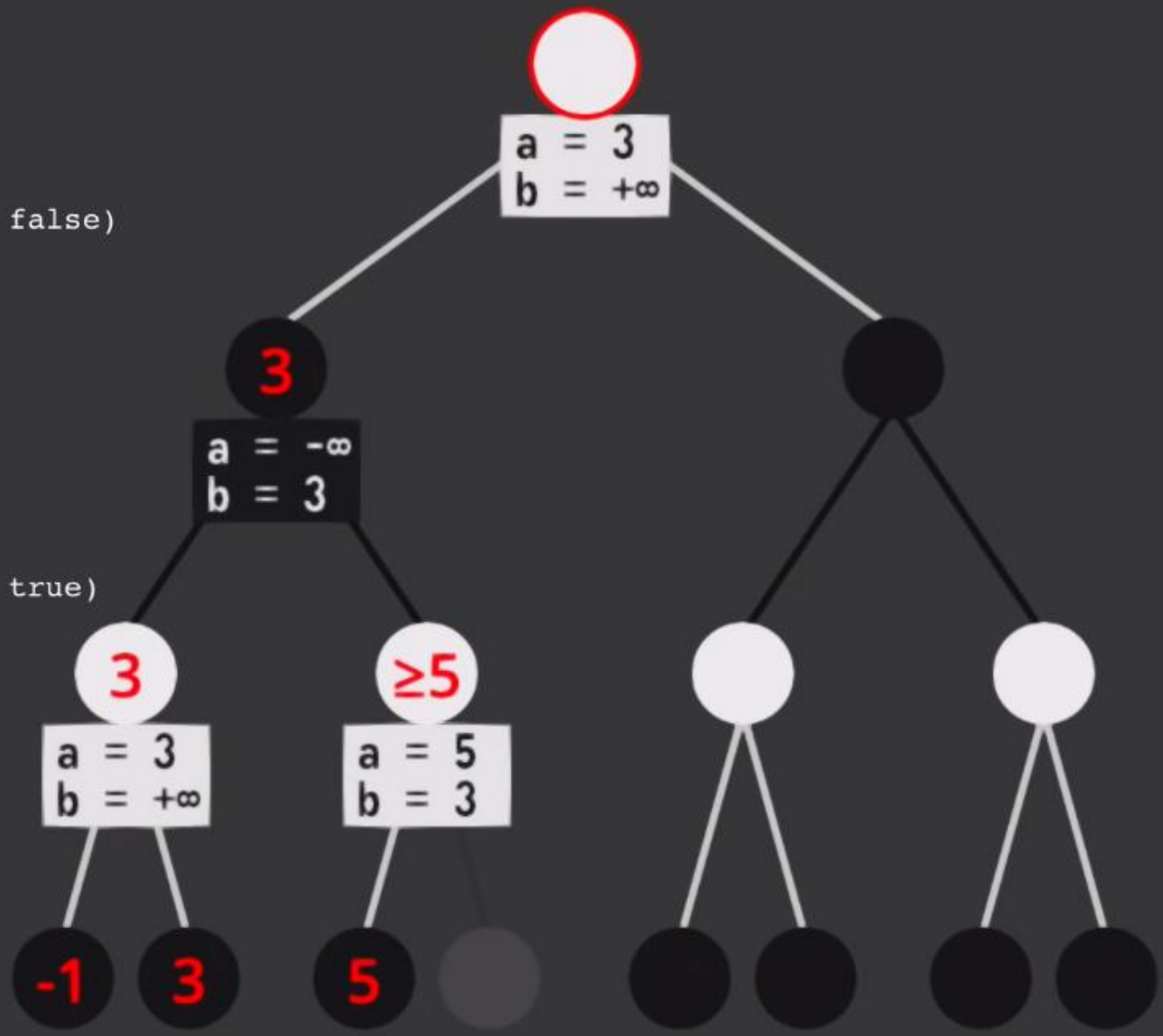
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



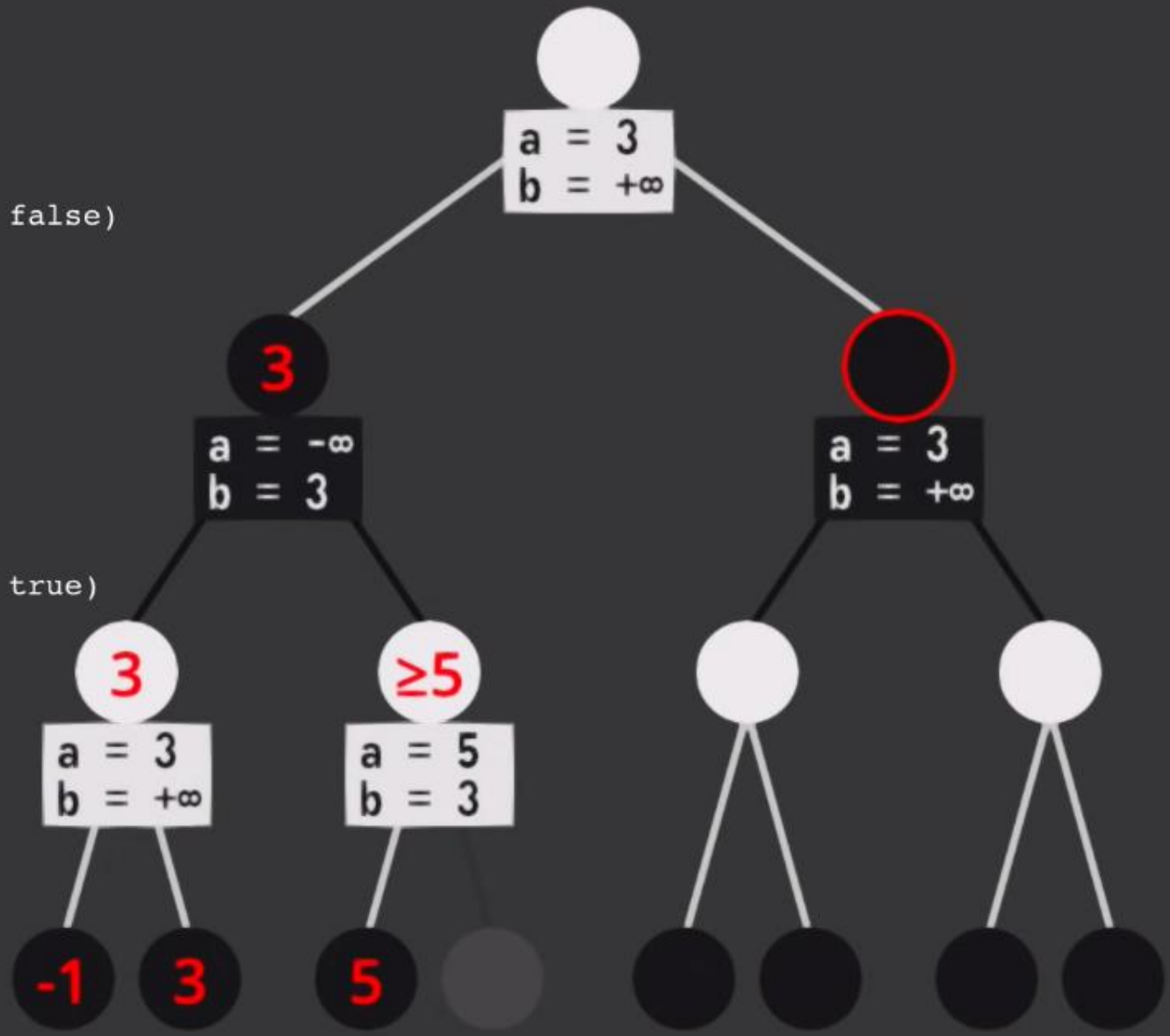
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



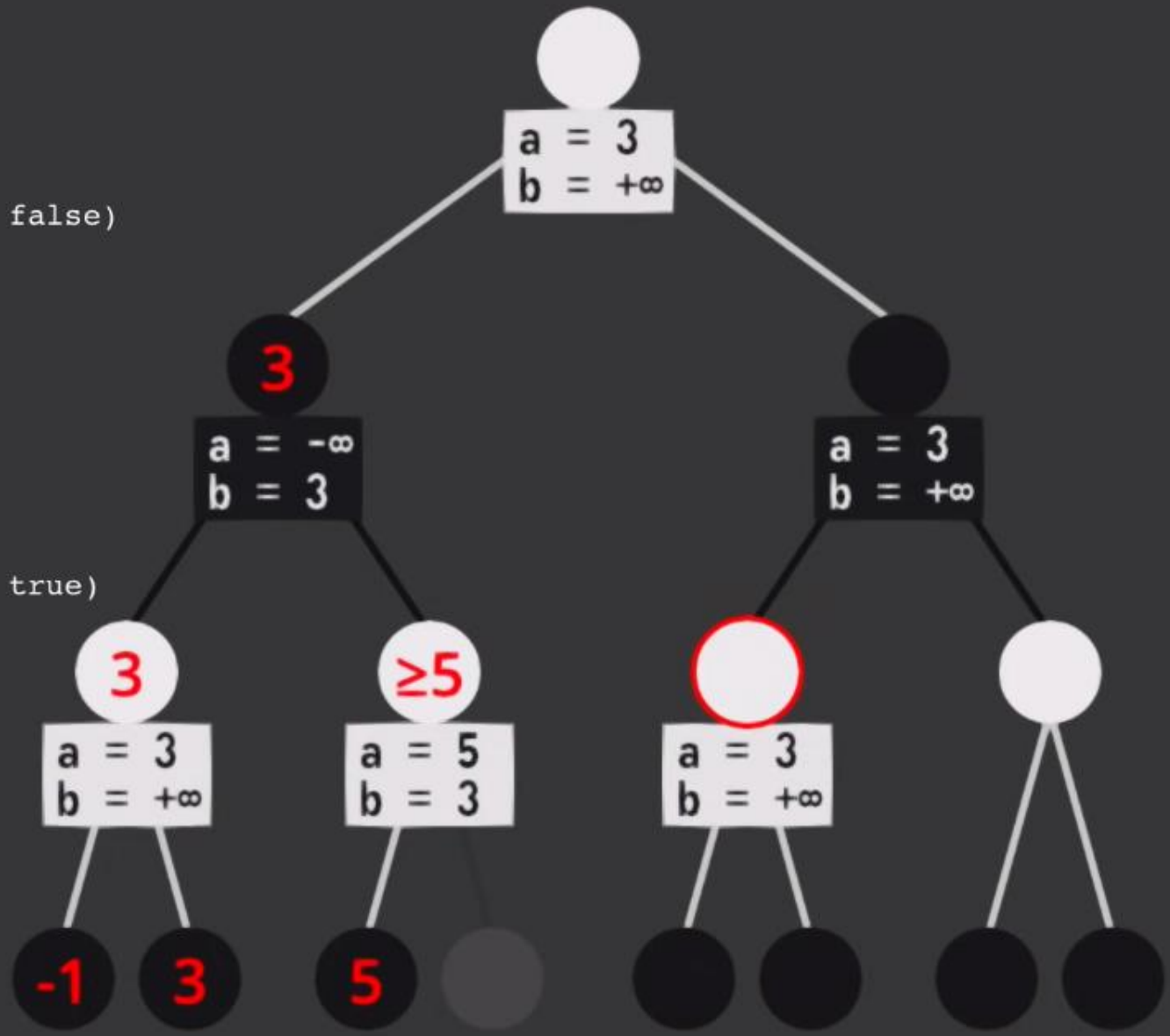
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



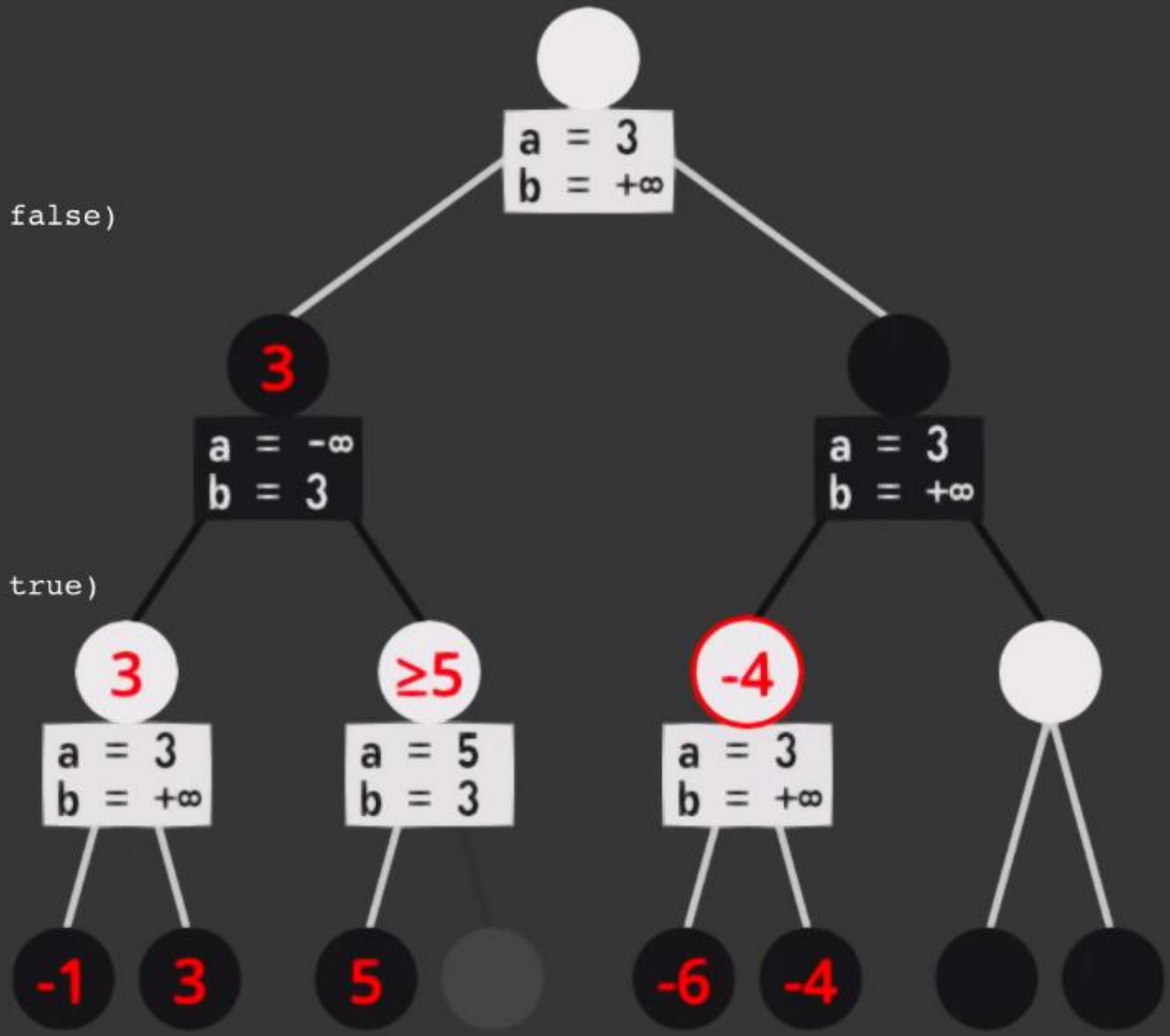
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



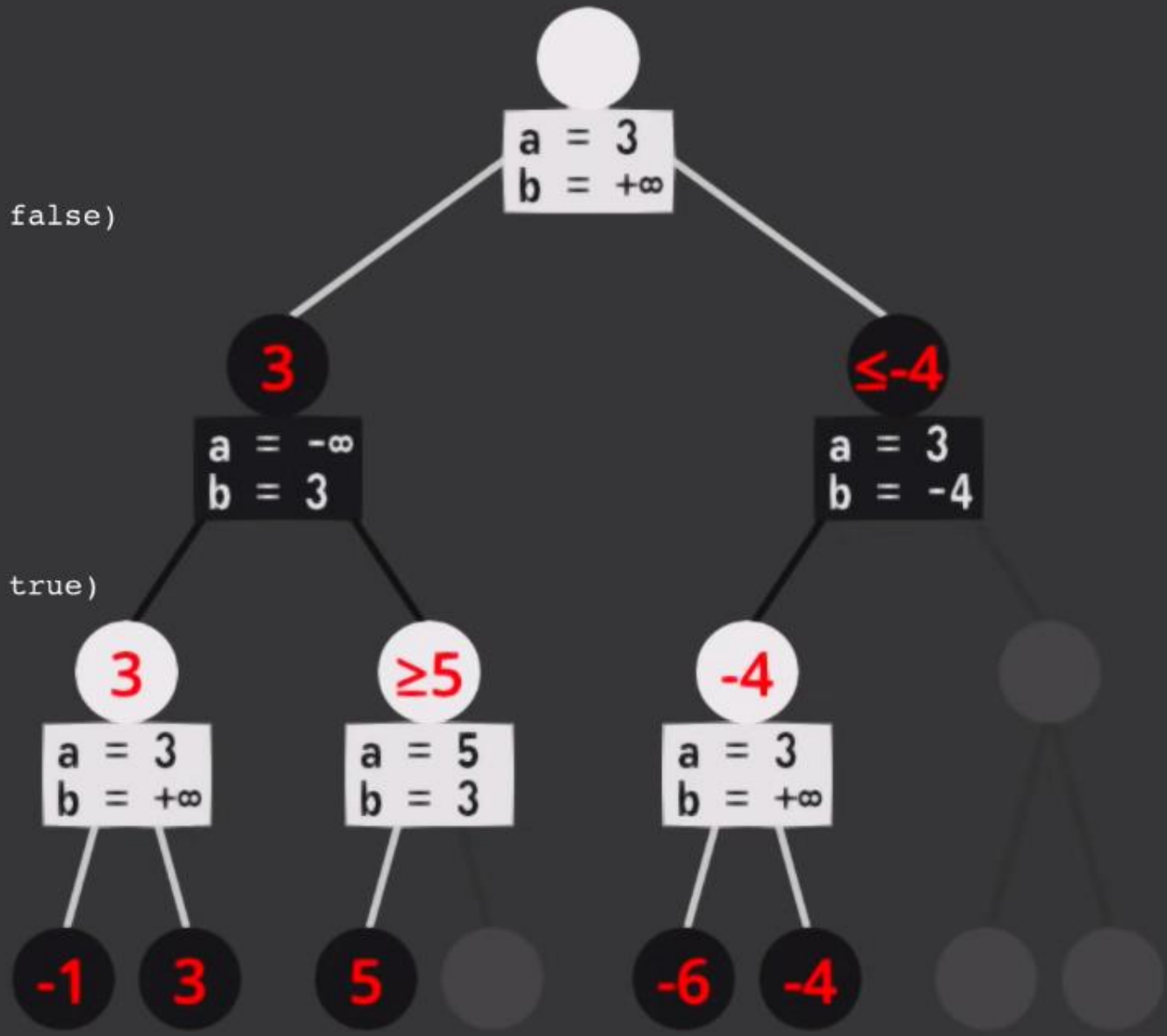
```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```



```
function minimax(position, depth, alpha, beta, maximizingPlayer)
   if depth == 0 or game over in position
      return static evaluation of position

   if maximizingPlayer
      maxEval = -infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

   else
      minEval = +infinity
      for each child of position
         eval = minimax(child, depth - 1, alpha, beta, true)
         minEval = min(minEval, eval)
         beta = min(beta, eval)
         if beta <= alpha
            break
      return minEval
```

```
// initial call
minimax(currentPosition, 3, -∞, +∞, true)
```

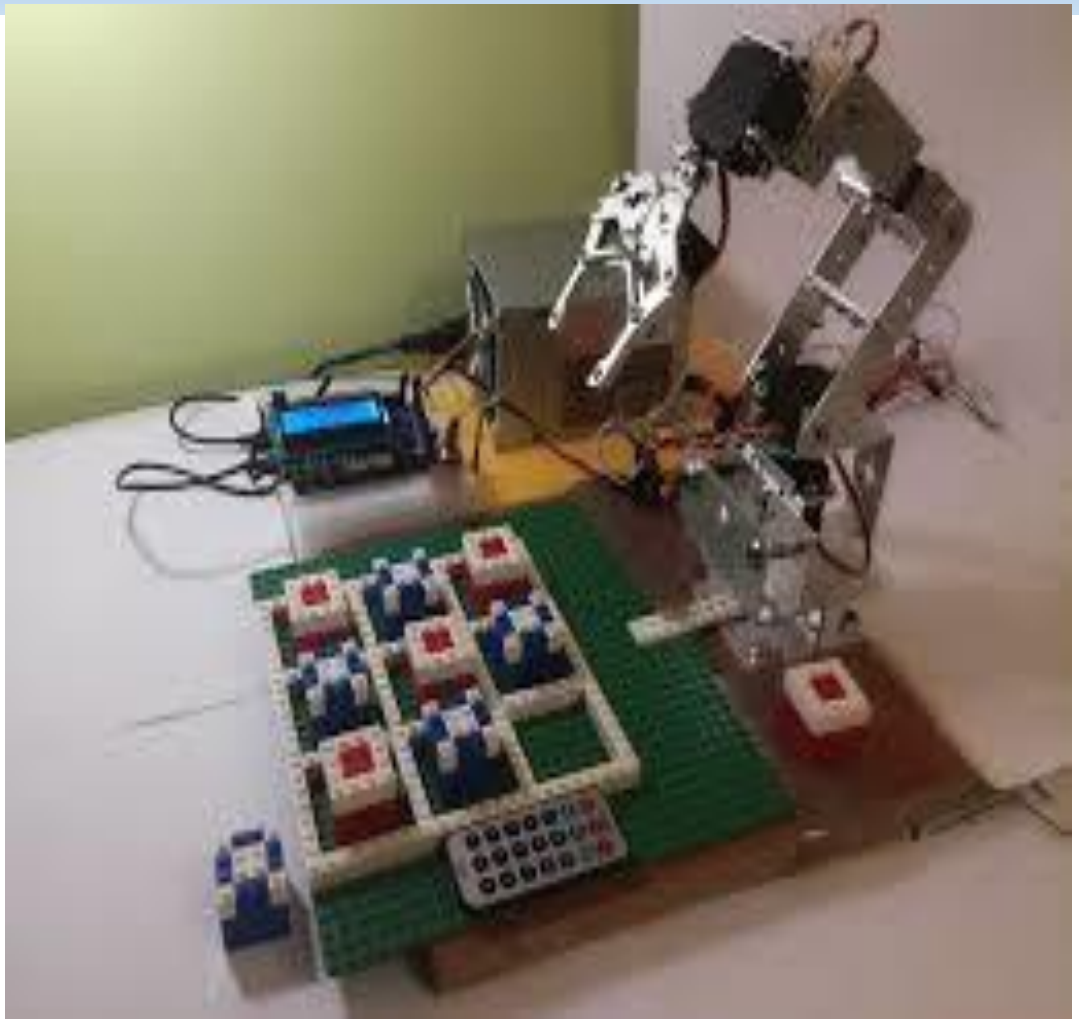


Alpha Beta Algorithm

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha
                break
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha
                break
        return minEval
```

Tic-Tac-toe Game with minmax Algorithm & Alpha Beta Algorithm

Tic-Tac-Toe Game

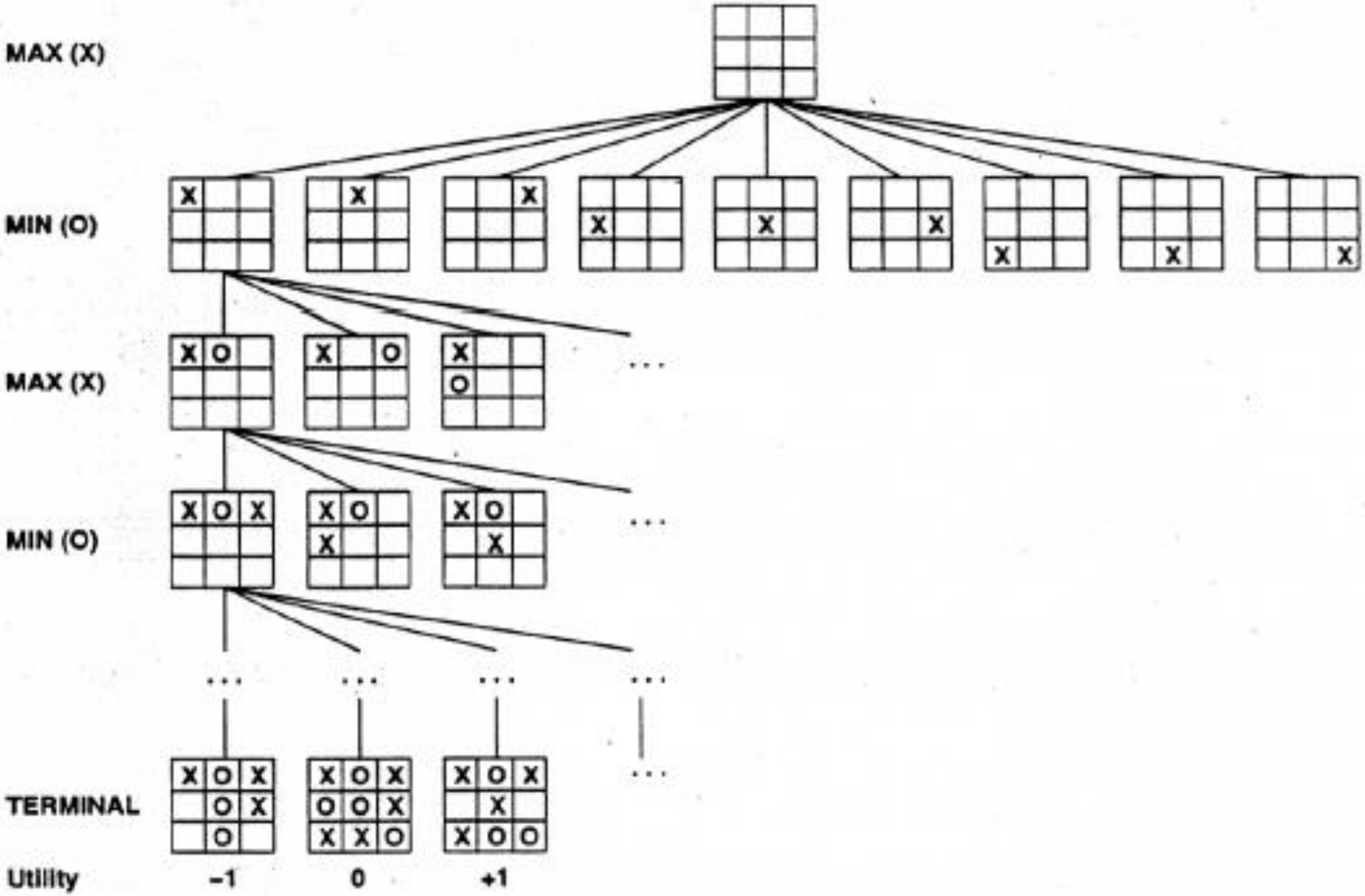


Solving 2-players Games

- **Statement of Game as a Search Problem:**
  - States = board configurations
  - Operators = legal moves. The transition model
  - Initial State = current configuration
  - Goal = winning configuration
  - payoff function (utility)= gives numerical value of outcome of the game

Tic-Tac-Toe Game

- **General principles of game-playing and search**
  - evaluation functions
  - minimax principle
  - alpha-beta-pruning


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
-1
0
+1

Heuristic function: Tic-Tac-Toe Game

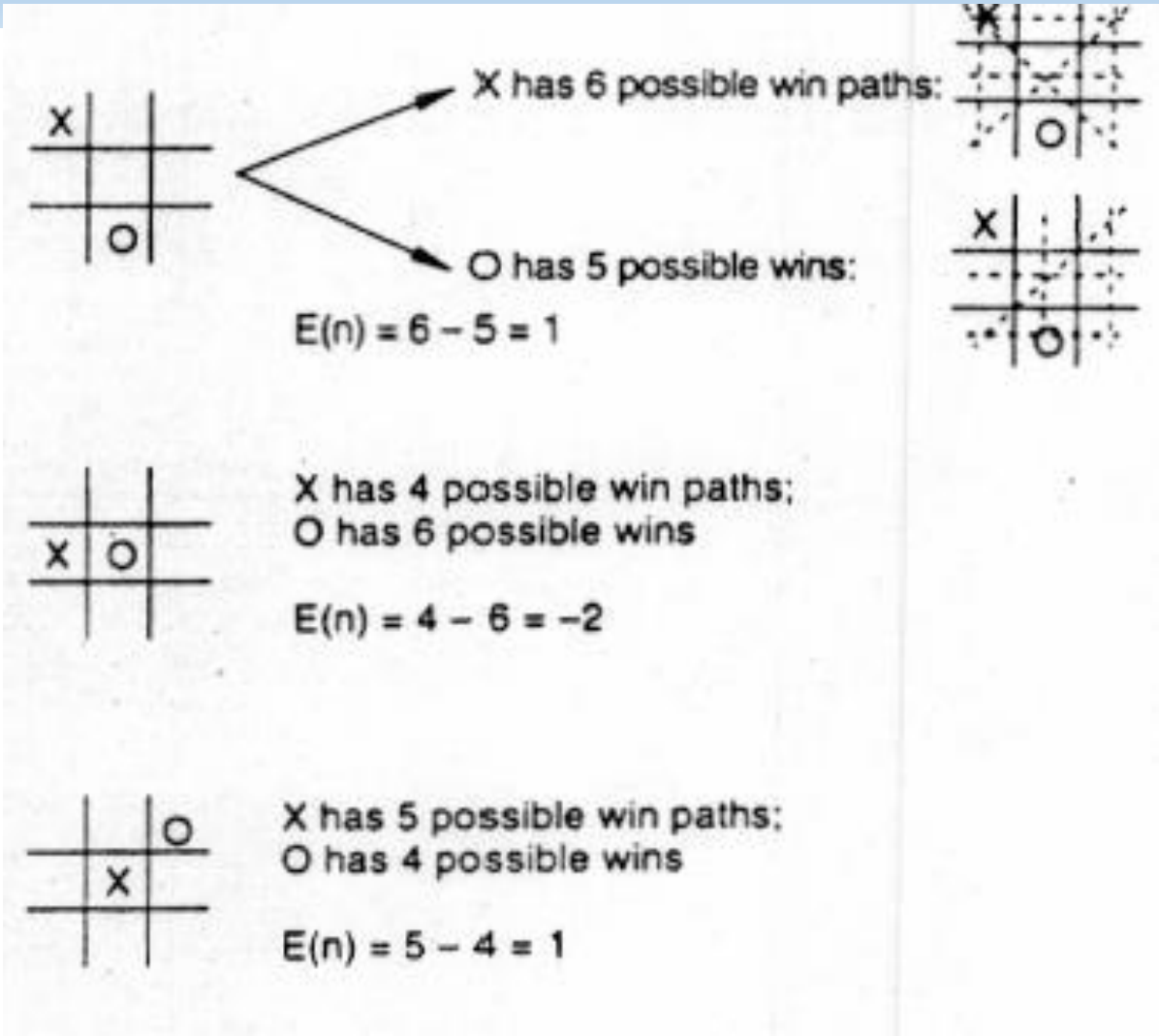
Heuristic is

$$E(n) = M(n) - O(n)$$

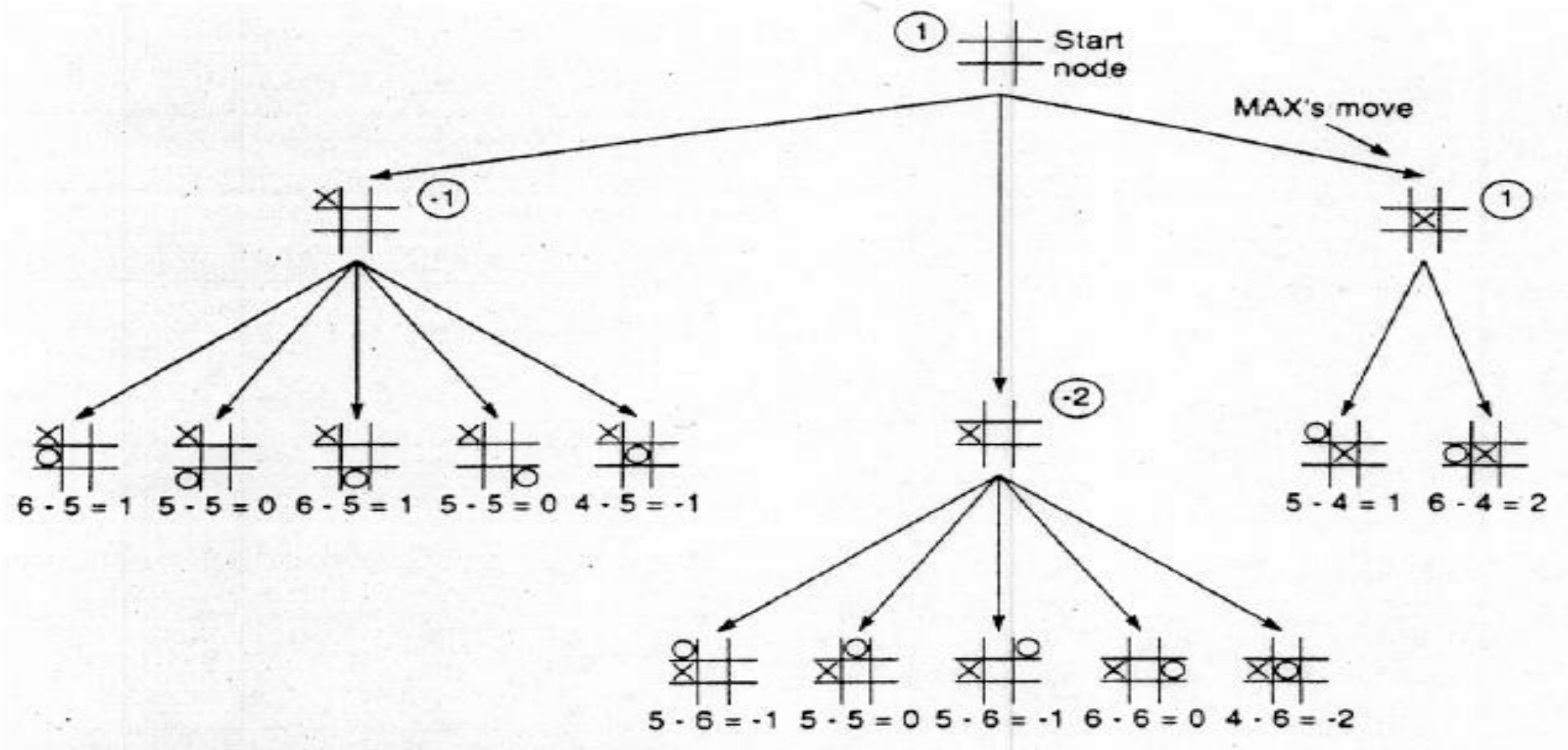
M(n): total no. of my possible winning lines

O(n): total no. of opponent's possible winning lines

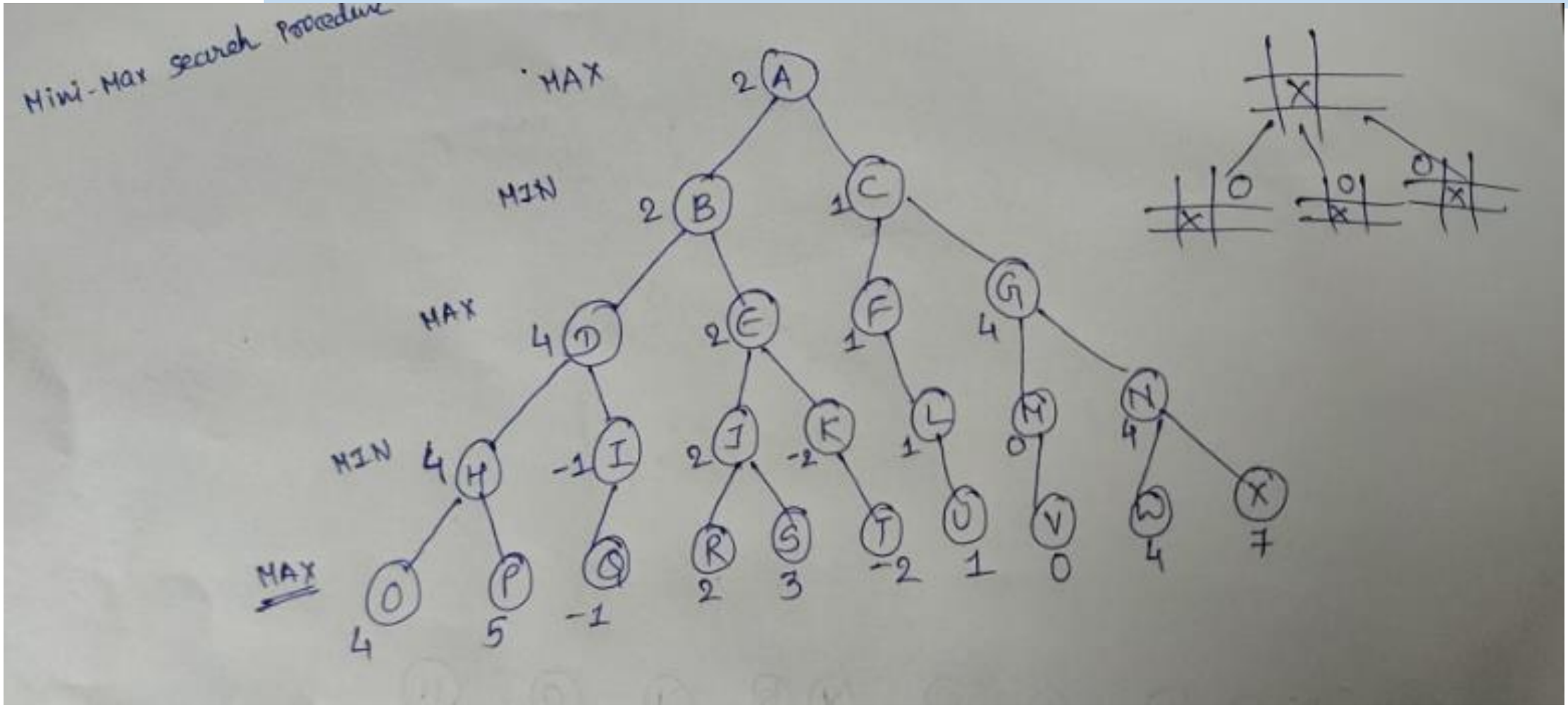
E(n): total Evaluation for state n



Back Values: Tic-Tac-Toe Game



Mini-Max Search Procedure: Tic-Tac-Toe Game



ALPHA BETA PRUNING

- Minimax procedure is a depth-first process
- One path is explored as far as time allows, the static evolution function is applied to the game positions at the last step of the path.
- The efficiency of the depth-first search can improve by branch and bound technique in which partial solutions that clearly worse than known solutions can abandon early.
- It is necessary to modify the branch and bound strategy to include two bounds, one for each of the players.
- This modified strategy called alpha-beta pruning.
-

KEY POINTS IN ALPHA BETA PRUNING

´Alpha: It is the best choice or the highest value that we have found at any instance along the path of Maximizer. The initial value for alpha is –∞.

´Beta: It is the best choice or the lowest value that we have found at any instance along the path of Minimizer. The initial value for alpha is + ∞.

´The condition for Alpha-beta Pruning is that α >= β.

´Each node has to keep track of its alpha and beta values. Alpha can be updated only when it's MAX's turn and, similarly, beta can be updated only when it's MIN's chance.

´MAX will update only alpha values and MIN player will update only beta values.

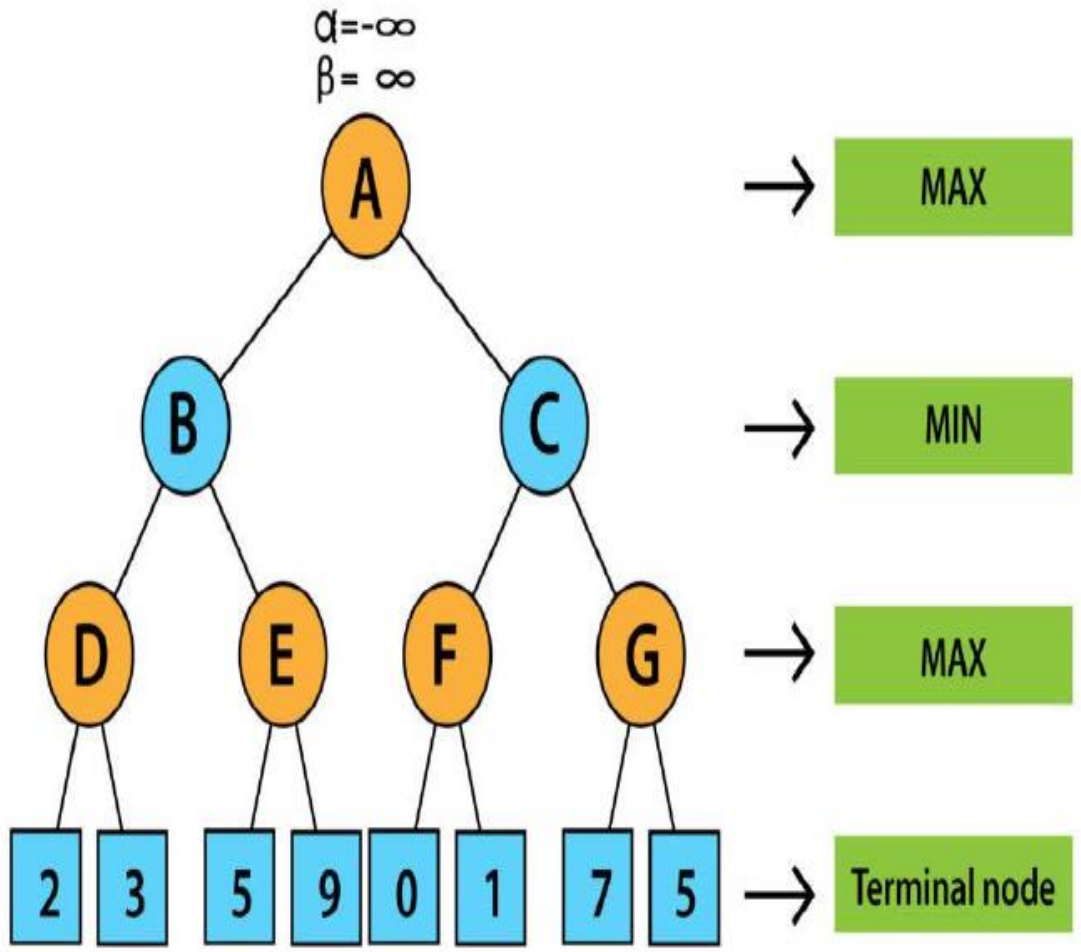
´The node values will be passed to upper nodes instead of values of alpha and beta during go into reverse of tree.

´Alpha and Beta values only be passed to child nodes.

´

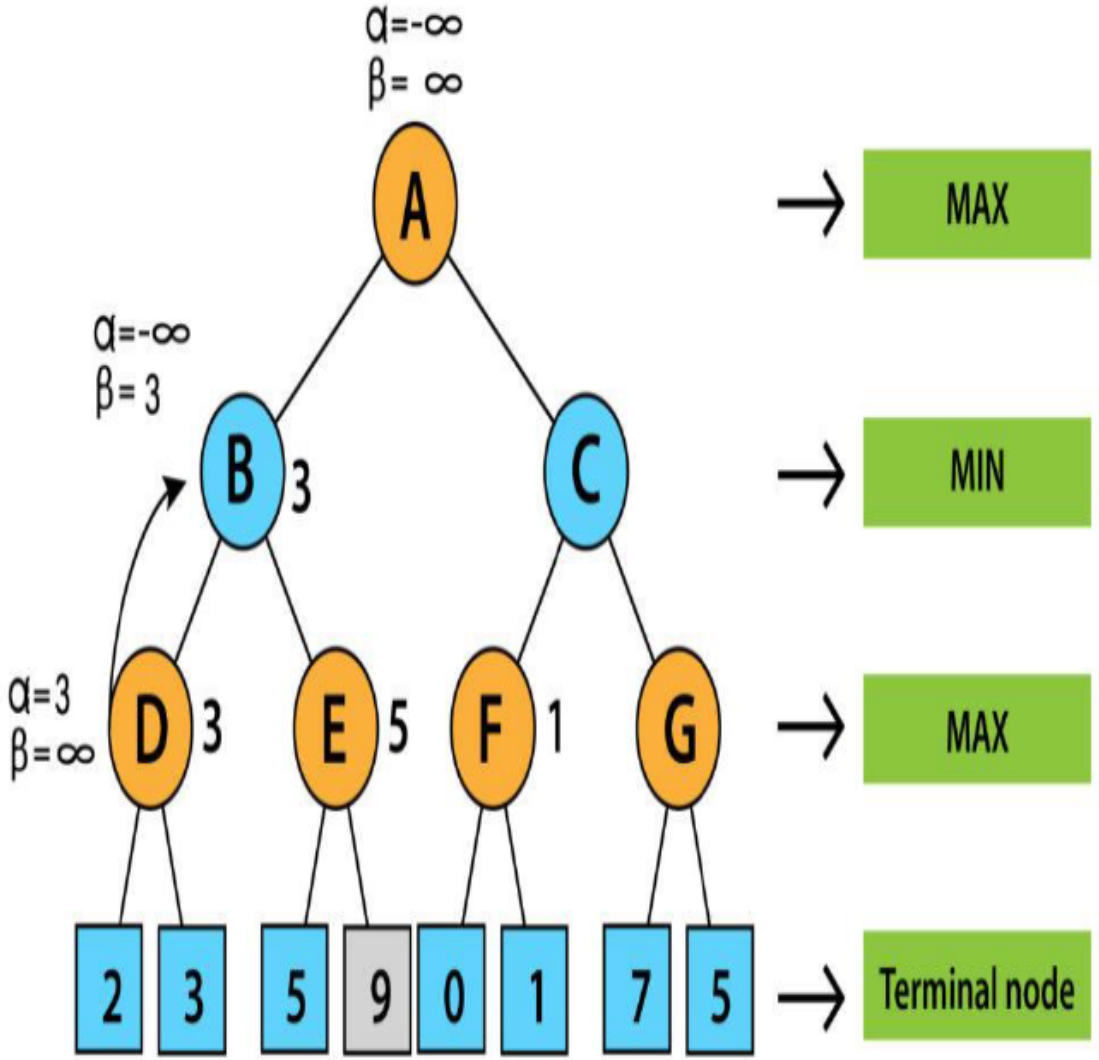
Working of ALPHA BETA PRUNING

We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. $\alpha = -\infty$ and $\beta = +\infty$. We will prune the node only when alpha becomes greater than or equal to beta.



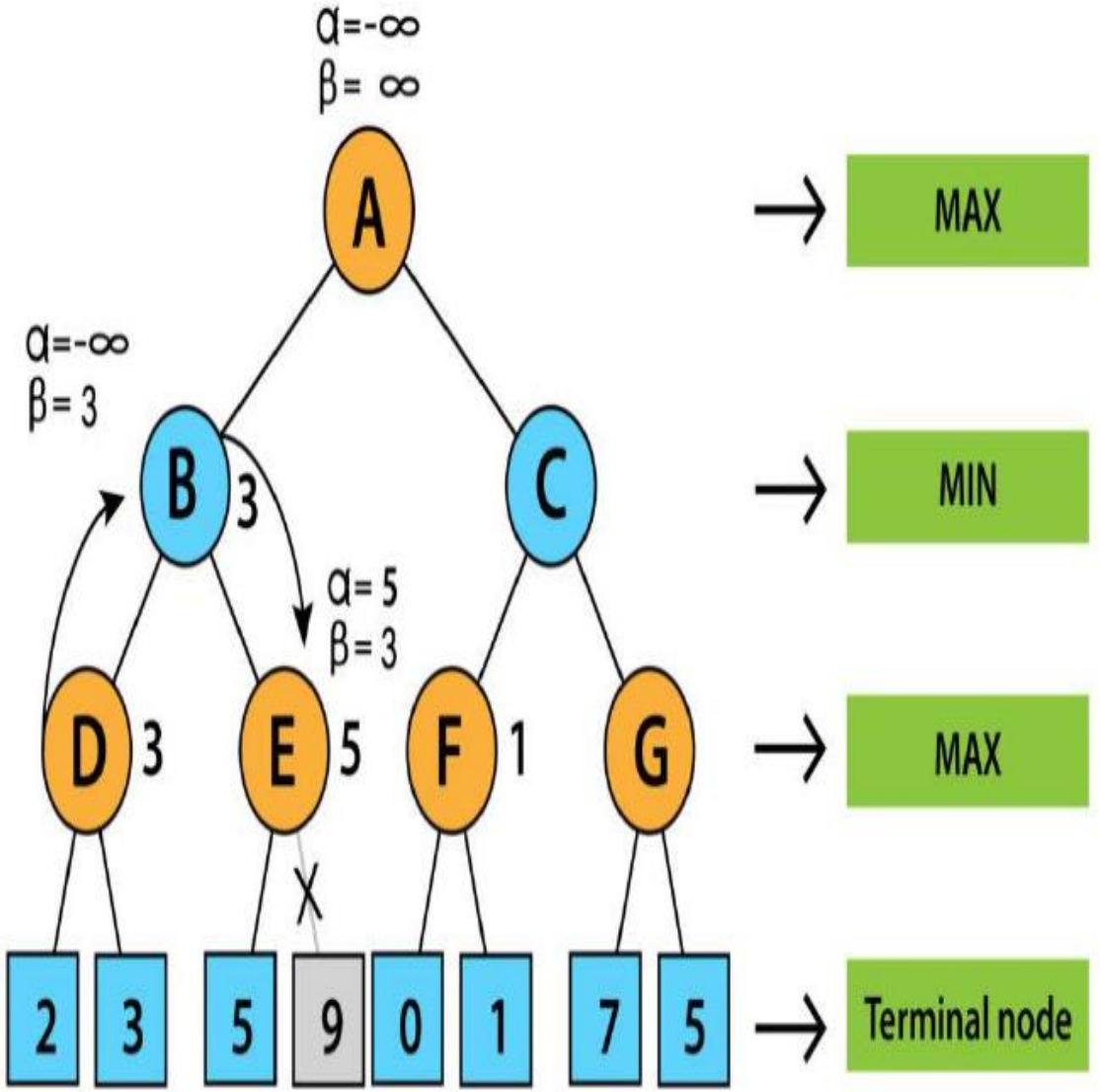
Working of ALPHA BETA PRUNING

- Since the initial value of alpha is less than beta so we didn't prune it. Now it's turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.
- Now the next move will be on node B and its turn for MIN now. So, at node B, the value of alpha beta will be min (3, ∞). So, at node B values will be alpha= – ∞ and beta will be 3.
- In the next step, algorithms traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.



Working of ALPHA BETA PRUNING

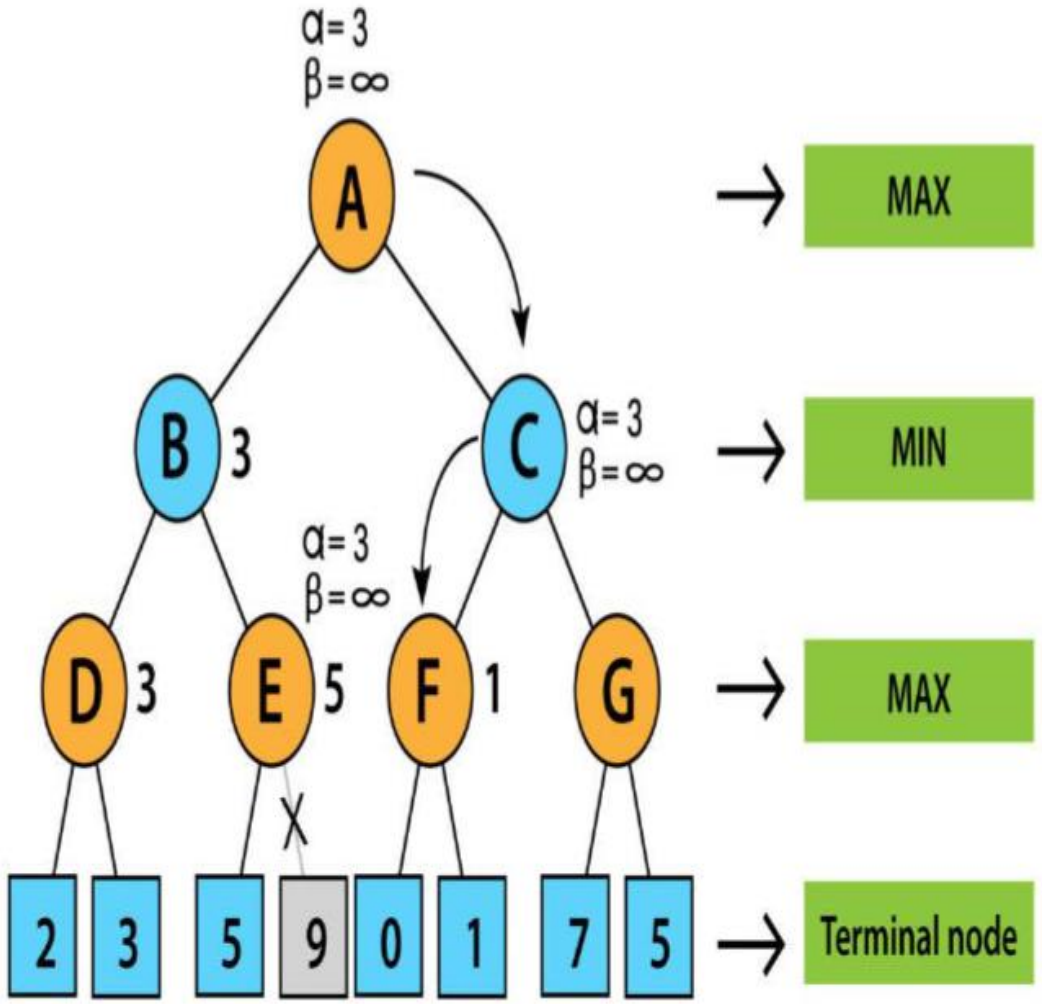
- Now it's turn for MAX. So, at node E we will look for MAX. The current value of alpha at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5. So, at node E, alpha = 5, Beta = 5. Now as we can see that alpha is greater than beta which is satisfying the pruning condition so we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.



Working of ALPHA BETA PRUNING

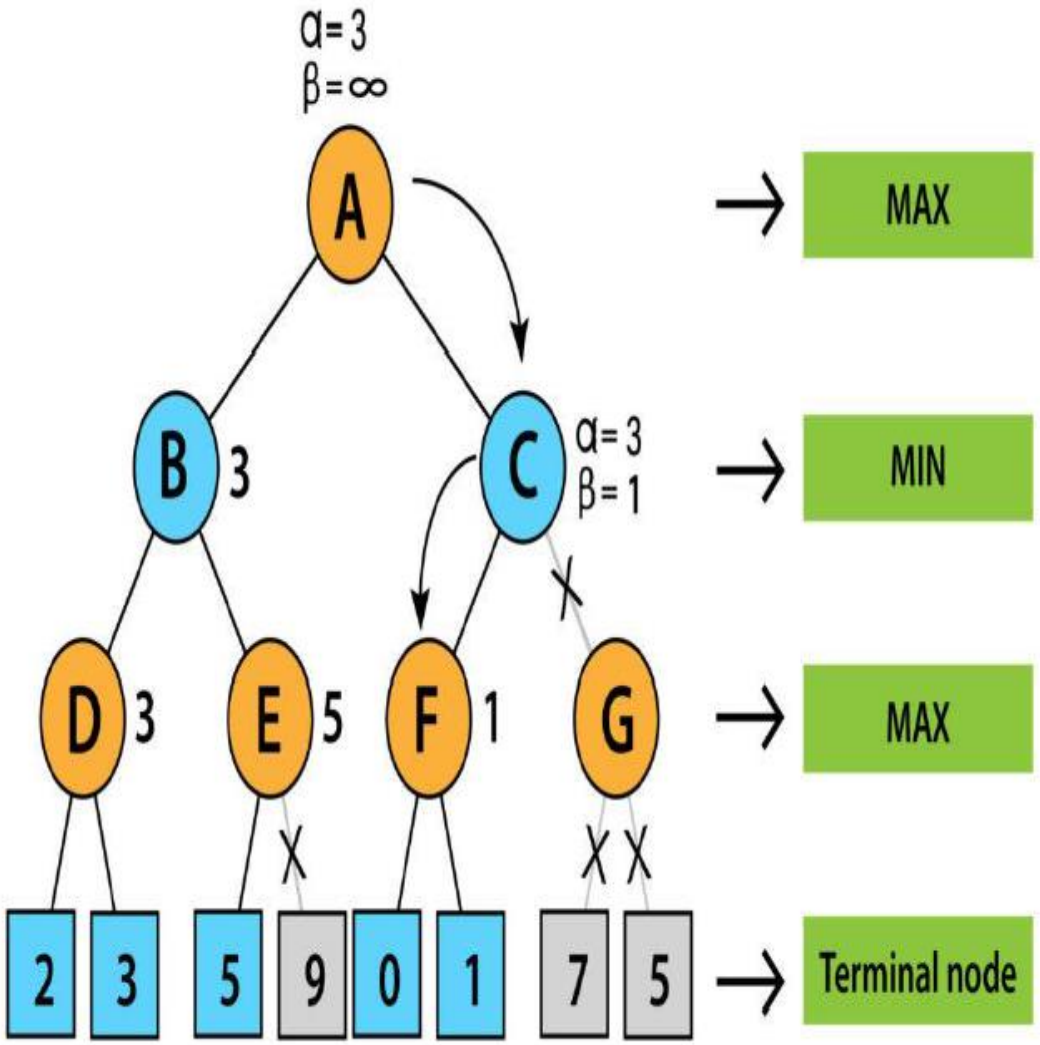
- In the next step the algorithm again comes to node A from node B. At node A alpha will be changed to maximum value as MAX (- ∞, 3). So now the value of alpha and beta at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.

- At node F the value of alpha will be compared to the left branch which is 0. So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3 still α remains 3, but the node value of F will become 1.



Working of ALPHA BETA PRUNING

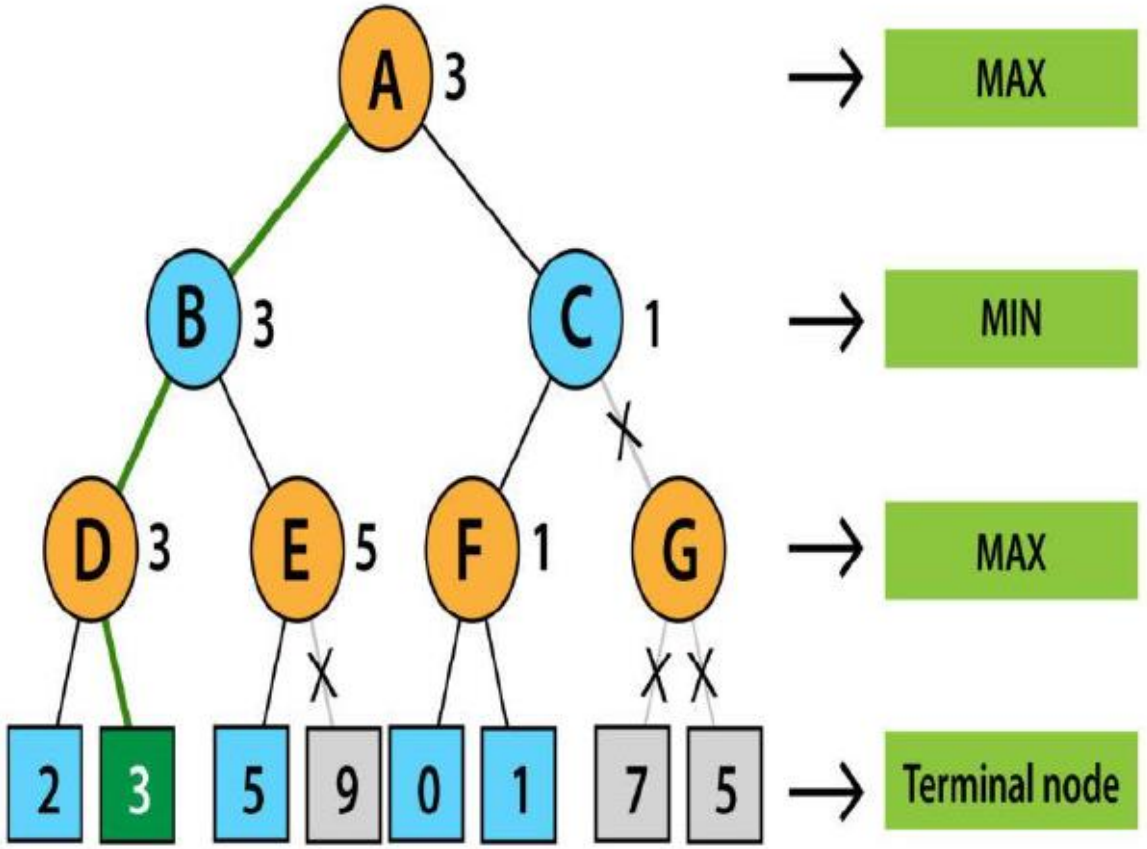
- Now node F will return the node value 1 to C and will compare to beta value at C. Now its turn for MIN. So, MIN (+ ∞, 1) will be 1. Now at node C, α= 3, and β= 1 and alpha is greater than beta which again satisfies the pruning condition. So, the next successor of node C i.e. G will be pruned and the algorithm didn't compute the entire subtree G.



Working of ALPHA BETA PRUNING

- Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3

- The represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.



Working of ALPHA BETA PRUNING

- **Move Ordering in Pruning**
- The effectiveness of alpha – beta pruning is based on the order in which node is examined. Move ordering plays an important role in alpha beta pruning.
- There are two types of move ordering in Alpha beta pruning:

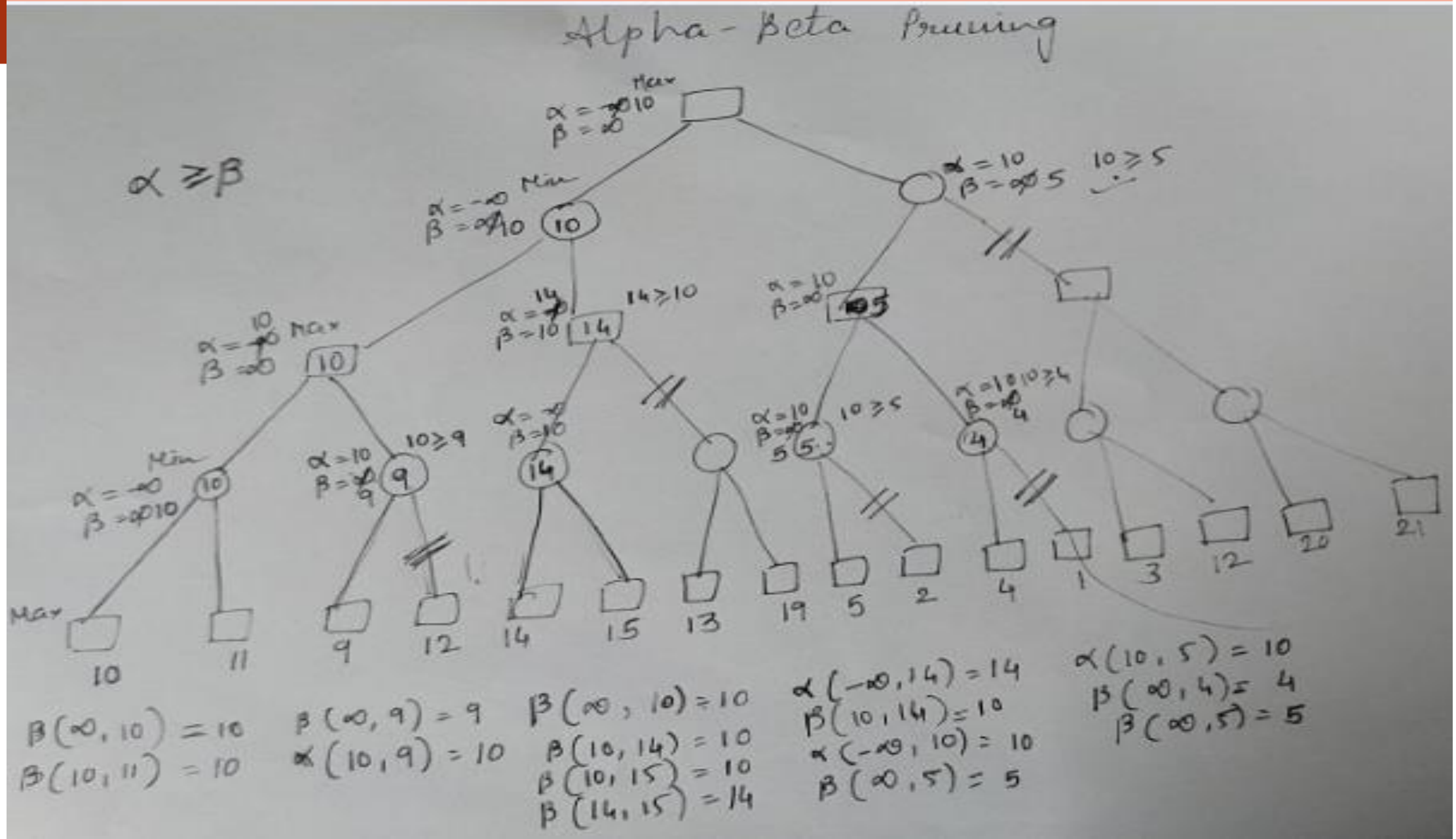
1. **Worst Ordering**: In some cases of alpha beta pruning none of the node pruned by the algorithm and works like standard minimax algorithm. This consumes a lot of time as because of alpha and beta factors and also not gives any effective results. This is called Worst ordering in pruning. In this case, the best move occurs on the right side of the tree.
2. **Ideal Ordering**: In some cases of alpha beta pruning lot of the nodes pruned by the algorithm. This is called Ideal ordering in pruning. In this case, the best move occurs on the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time.

- **Rules to find Good ordering**
- The best move happens from the lowest node
- Use domain knowledge while finding the best move
- Order of nodes should be in such a way that the best nodes will be computed first

Algorithm: ALPHA BETA PRUNING

```
Minimaxab(board, player, alpha, beta) :
    if board is a leaf node :
        return the value of board
    if player = 1 :
        best = -∞
        for each child of board :
            val = minimaxab(child, -1, alpha, beta)
            best = max(val, best)
            alpha = max(best, alpha)
            if alpha ≥ beta :
                break
        return best
    else :
        best = ∞
        for each child of board :
            val = minimaxab(child, -1)
            best = min(val, best)
            beta = min(best, beta)
            if alpha ≥ beta :
                break
        return best
```

Alpha-Beta Pruning: Tic-Tac-Toe Game



Thanks