

# Introduction to Artificial Intelligence

IFT3335 Lecture 5: Adversarial Search and Games

*Bang Liu, Jian-Yun Nie*

## 2 Certain Slides Adapted From or Referred To...

- ◎ Slides from **UC Berkeley CS188, Dan Klein and Pieter Abbeel**
  - Game Trees I & II: <https://inst.eecs.berkeley.edu/~cs188/su21/>
- ◎ Slides from **UPenn CIS391, Mitch Marcus**
  - 2-Player Games: Adversarial Search: <https://www.seas.upenn.edu/~cis391/#LECTURES>
- ◎ <https://www.javatpoint.com/ai-alpha-beta-pruning>
- ◎ <https://int8.io/monte-carlo-tree-search-beginners-guide/>

## 3 Plan

- Game AI
- The MiniMax Rule
- Alpha-Beta Pruning
- Monte-Carlo Tree Search
- Search with Uncertainty

# Game AI



# AI for Checkers

- 1950: First computer player.
- 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame.
- 2007: Checkers was weakly solved in 2007 by a team of Canadian computer scientists led by Jonathan Schaeffer. From the standard starting position, perfect play by each side would result in a draw!



## AI for Chess

- 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- <https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/>



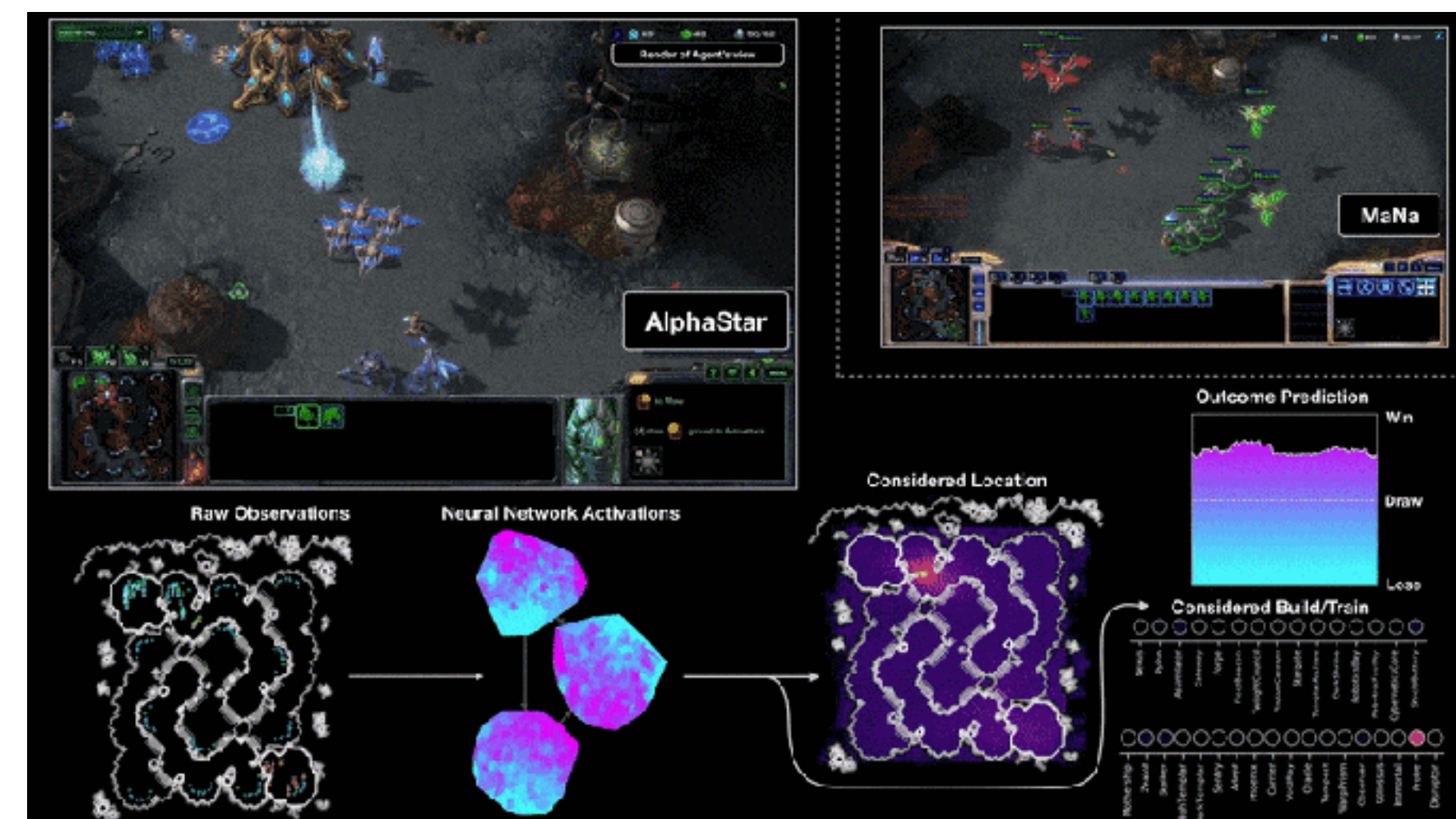
## 7 AI for Go

- Go originated in China over 3,000 years ago. Winning this board game requires multiple layers of strategic thinking.
- Two players, using either white or black stones, take turns placing their stones on a board. The goal is to surround and capture their opponent's stones or strategically create spaces of territory. Once all possible moves have been played, both the stones on the board and the empty points are tallied. The highest number wins.
- As simple as the rules may seem, Go is profoundly complex. There are an astonishing 10 to the power of **170 possible board configurations - more than the number of atoms in the known universe**. This makes the game of Go a googol times more complex than chess.
- 2016: AlphaGO (created by DeepMind) defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function. (More details later.)



## 8 More Games...

- Poker AI: Libratus (CMU, 2017), Pluribus (CMU, 2019), DeepStack (University of Alberta)...
- StarCraft AI: AlphaStar (DeepMind, 2019)
- DotA 2 AI: OpenAI Five (OpenAI, 2018)





## 9 Types of Games

- Many different kinds of games!
- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?
- Want algorithms for calculating a **strategy (policy)** which recommends a move from each state

	Deterministic	Chance
Perfect Information	chess, checkers, go, othello	backgammon, monopoly
Imperfect Information	battleship	bridge, poker, scrabble, nuclear war

# 10 Deterministic Games

## ⦿ Deterministic v.s. nondeterministic

- Whether the next state of the environment is completely determined by the current state and the action executed by the agent(s)

## ⦿ Many possible formalizations, one is:

- States:  $S$  (start at  $s_0$ )
- Players:  $P = \{1 \dots N\}$  (usually take turns)
- Actions:  $A$  (may depend on player / state)
- Transition Function:  $S \times A \rightarrow S$
- Terminal Test:  $S \rightarrow \{t, f\}$
- Terminal Utilities:  $S \times P \rightarrow R$

## ⦿ Solution for a player is a policy: $S \rightarrow A$



# Zero-Sum Games

## Zero-Sum Games

- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

## General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible

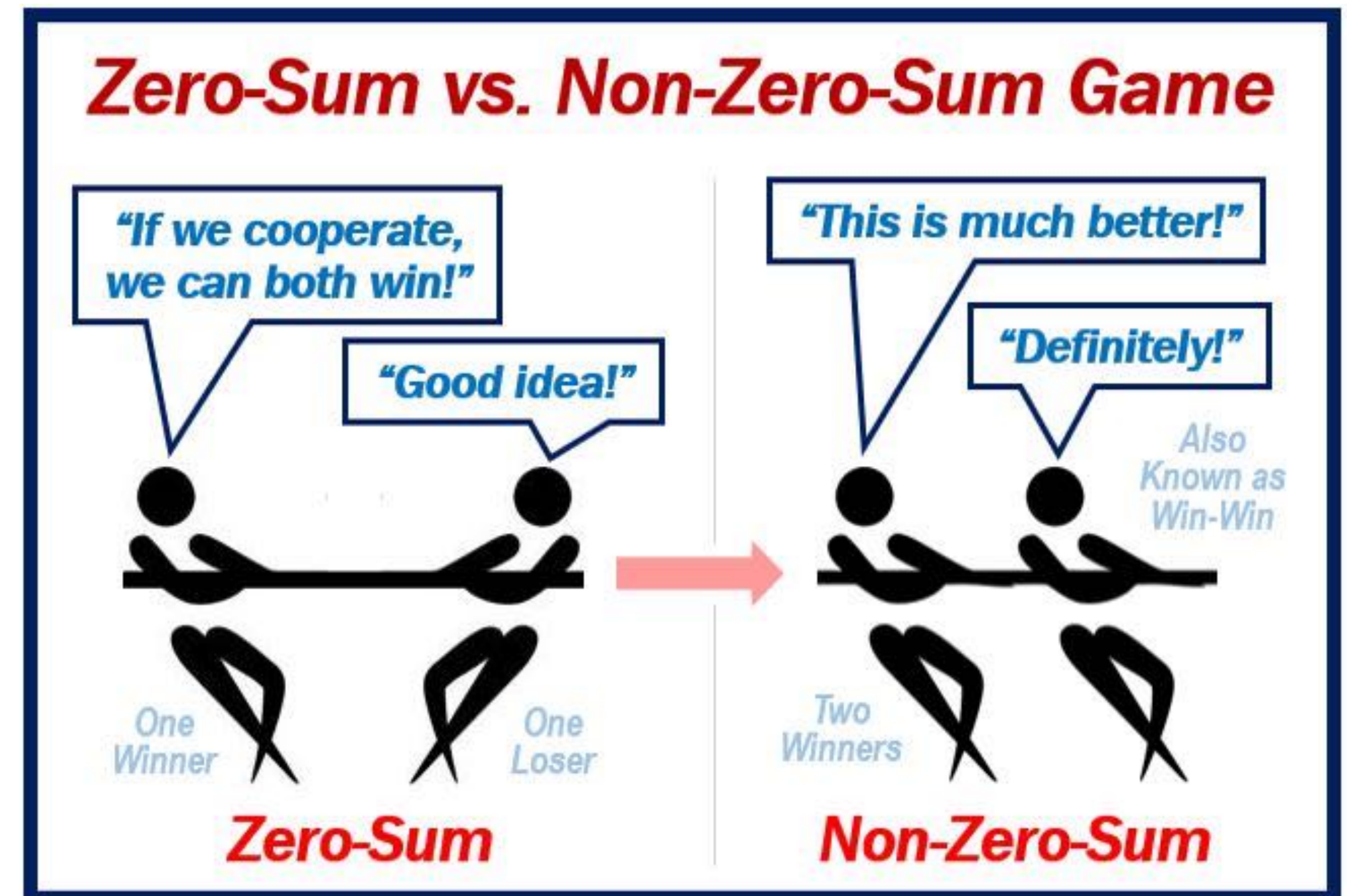
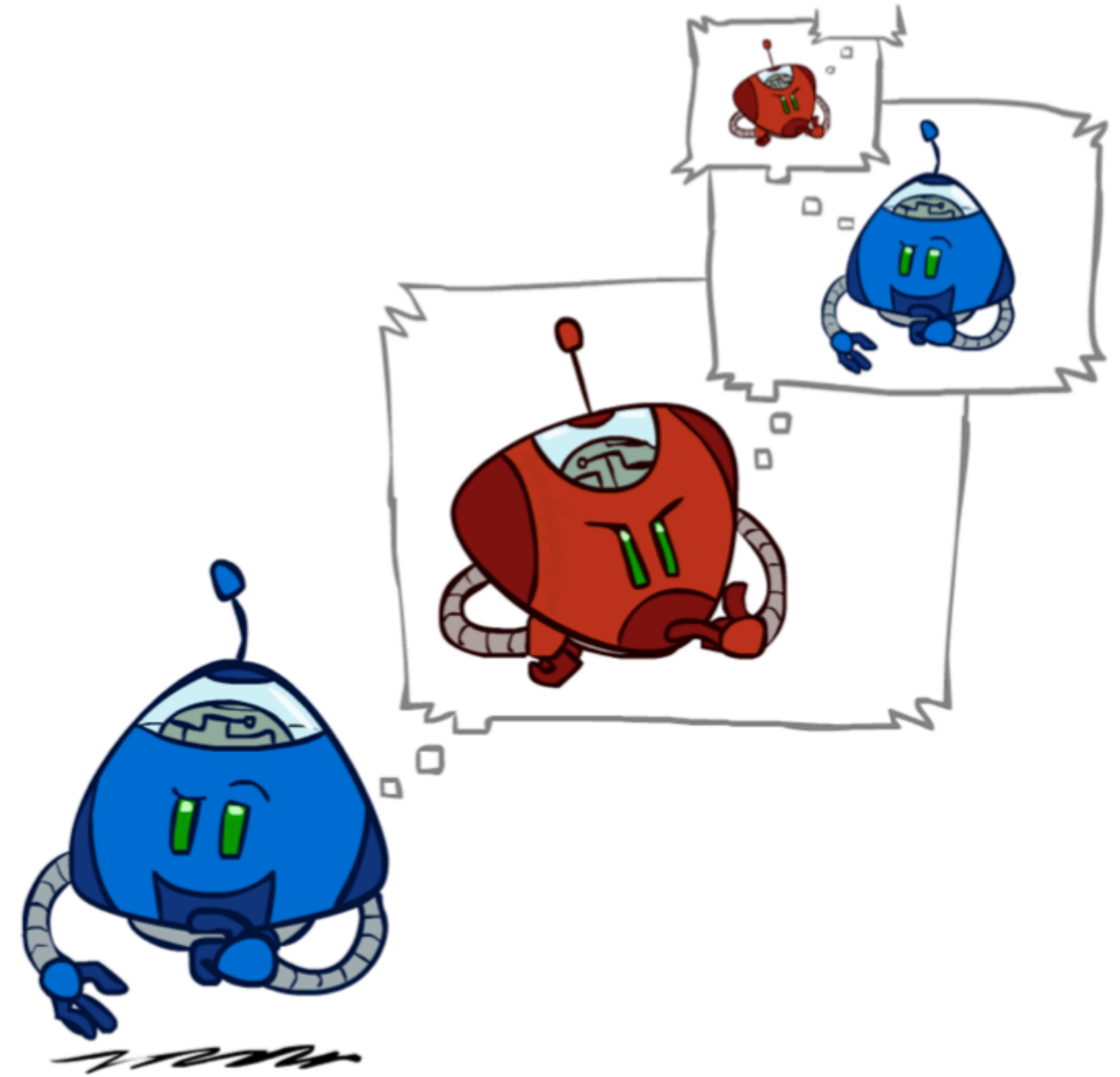


Image created by Market Business News.

# Adversarial Search

- Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.
- The environment with more than one agent is termed as **multi-agent environment**. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called **adversarial searches**, often known as **Games**



# Game Tree

A game tree is a tree where nodes of the tree are the game **states** and Edges of the tree are the **moves** by players.

The right figure is showing part of the game-tree for **tic-tac-toe** game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.



MAX (X)

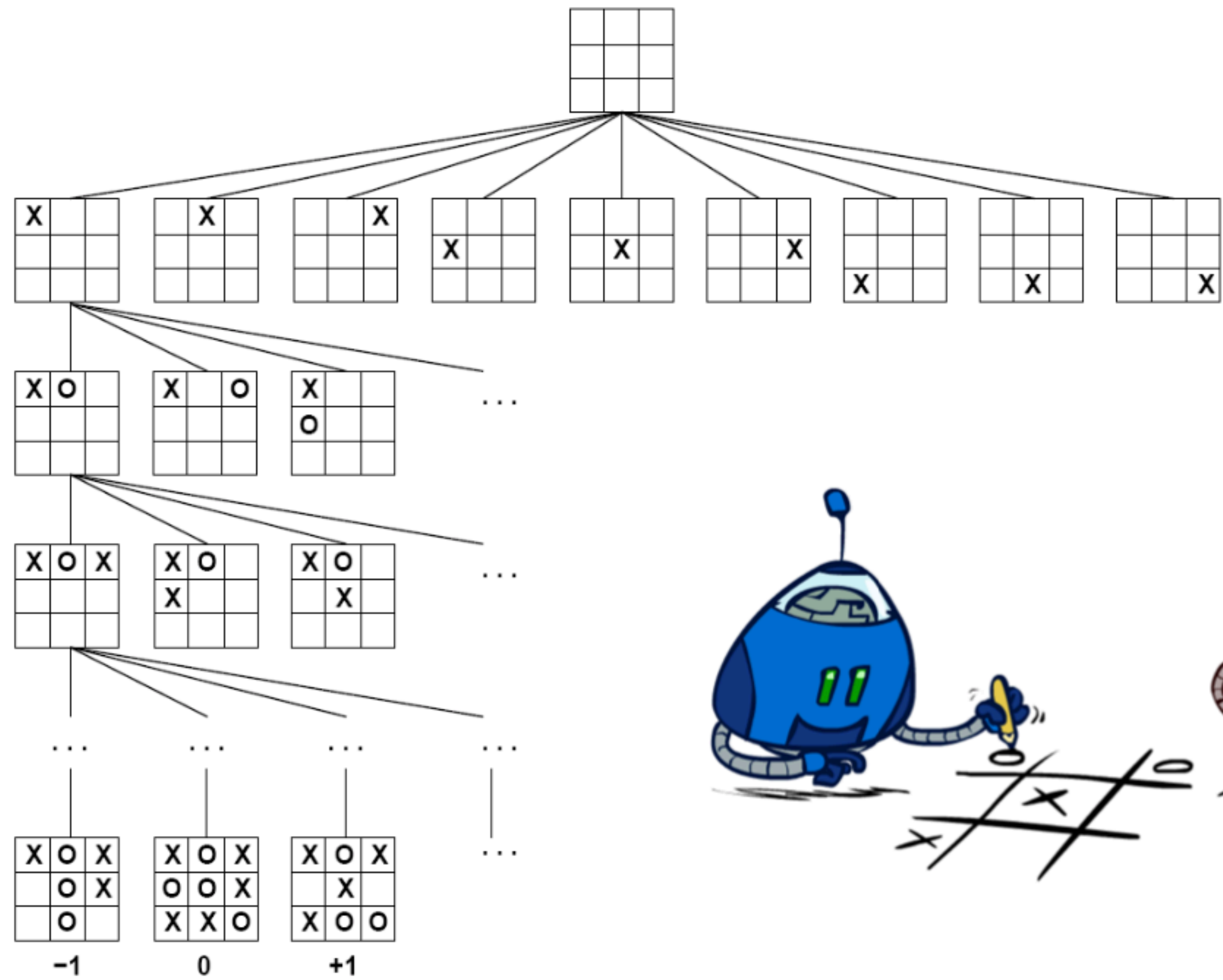
MIN (O)

MAX (X)

MIN (O)

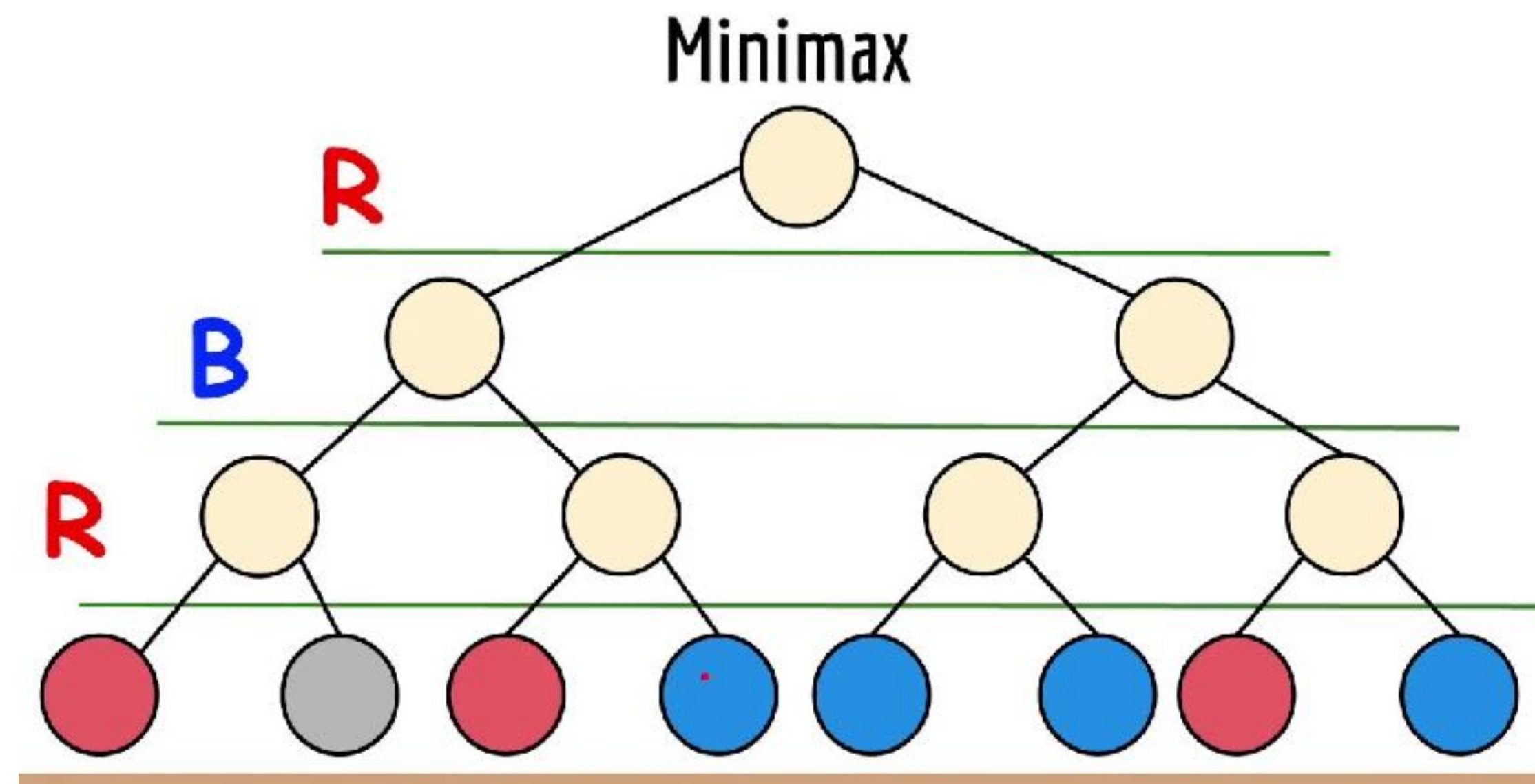
TERMINAL

Utility



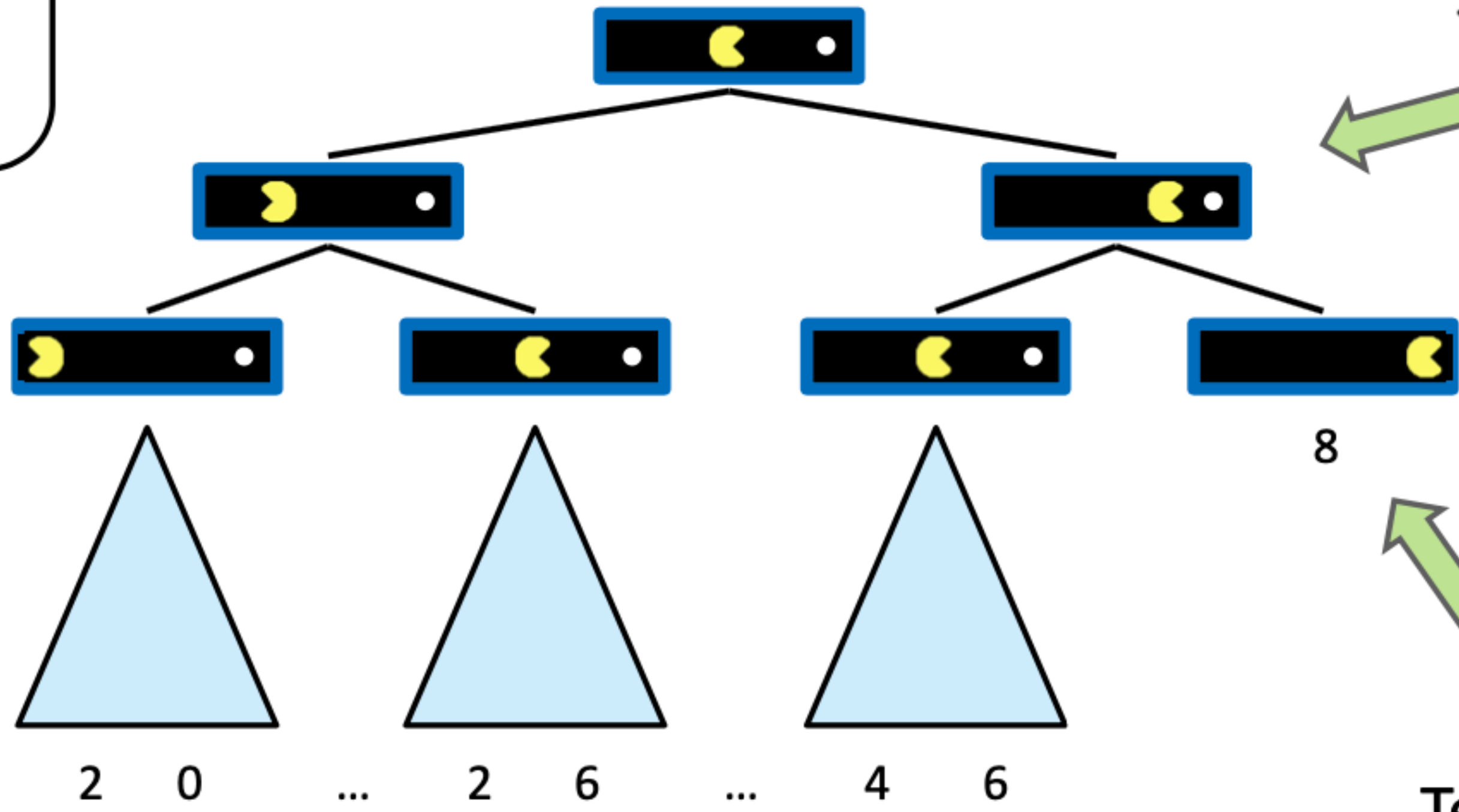
**Tic-Tac-Toe Game Tree**

# The MiniMax Rule



# 15 Value of a State

Value of a state:  
The best achievable  
outcome (utility)  
from that state



Non-Terminal States:  
 $V(s) = \max_{s' \in \text{children}(s)} V(s')$

Terminal States:  
 $V(s) = \text{known}$

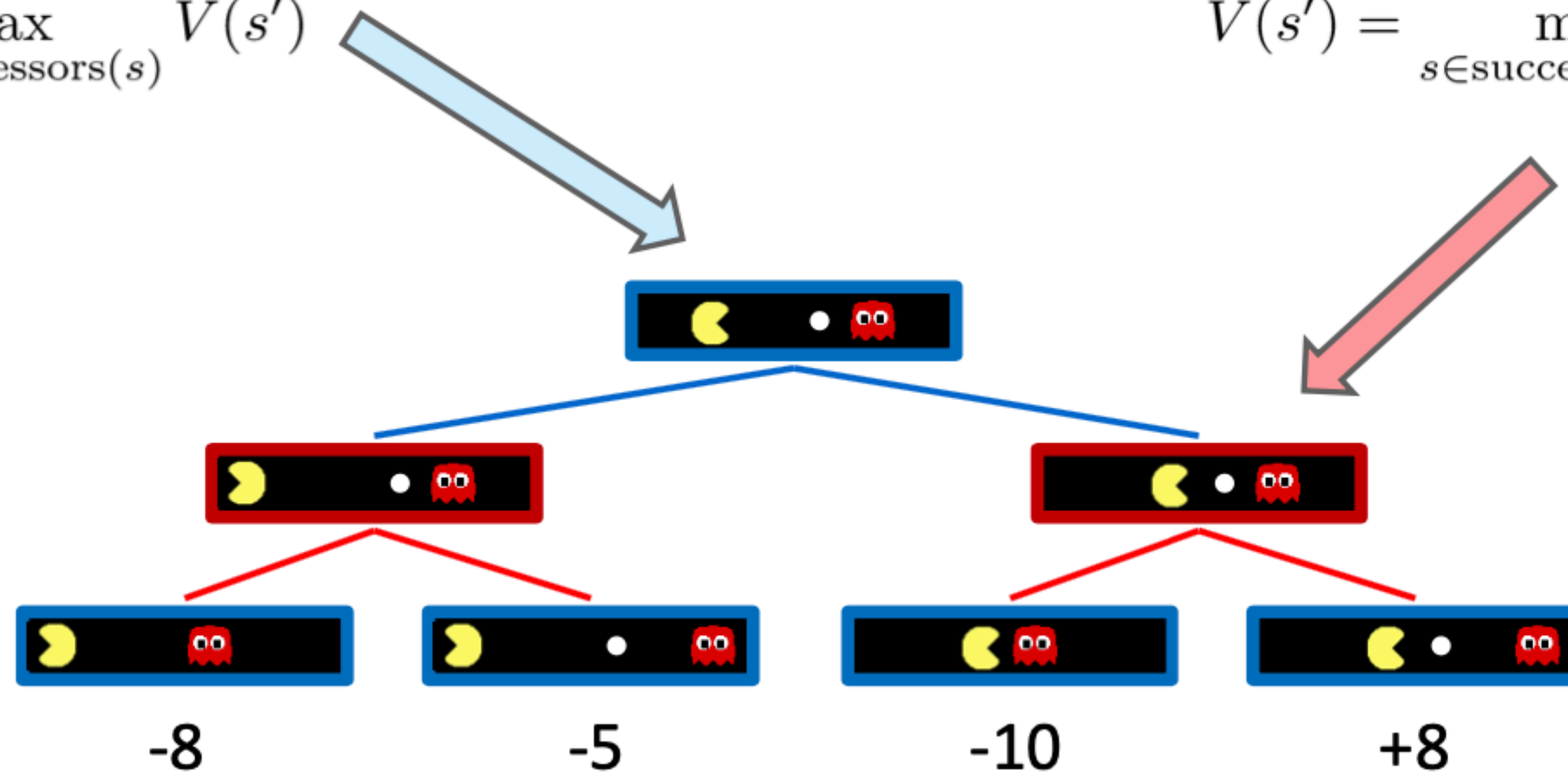
# 16 Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

$$V(s) = \text{known}$$

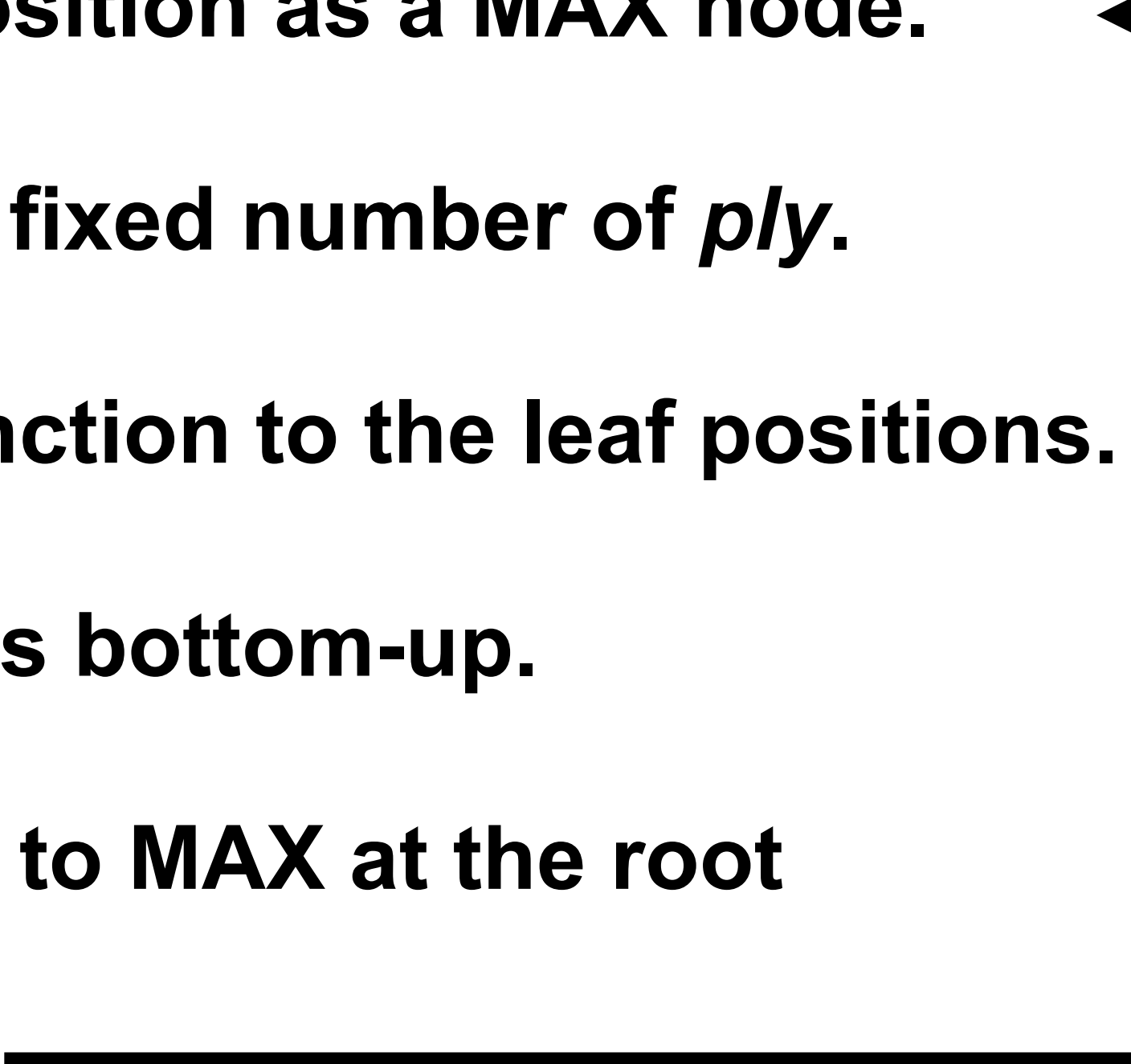


## 17 Minimax Algorithm

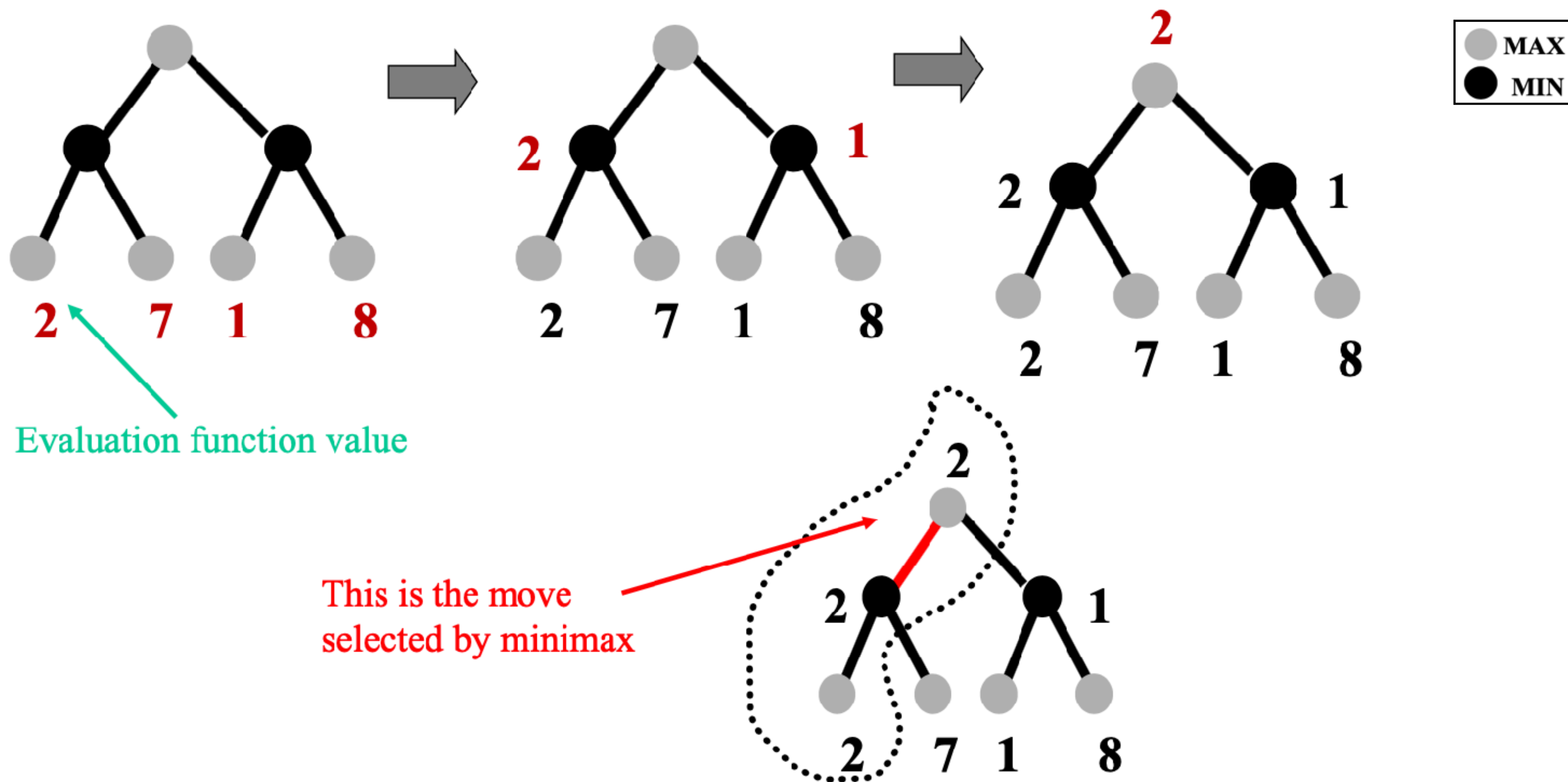
- **Idea:** Make the best move for MAX assuming that *MIN always replies with the best move for MIN*
- **Easily computed by a recursive process:**
  - The **backed-up value (i.e., state value)** of each node in the tree is determined by the values of its children:
    - For a **MAX** node, the backed-up value is the maximum of the value of its children (*i.e., the best for MAX*)
    - For a **MIN** node, the backed-up value is the minimum of the values of its children (*i.e. the best for MIN*)

# The Minimax Procedure

**Until game is over:**

- 1. Start with the current position as a MAX node.**
  - 2. Expand the game tree a fixed number of *ply*.**
  - 3. Apply the evaluation function to the leaf positions.**
  - 4. Calculate back-up values bottom-up.**
  - 5. Pick the move assigned to MAX at the root**
  - 6. Wait for MIN to respond**
- 

# 19 2-ply Example: Backing up values

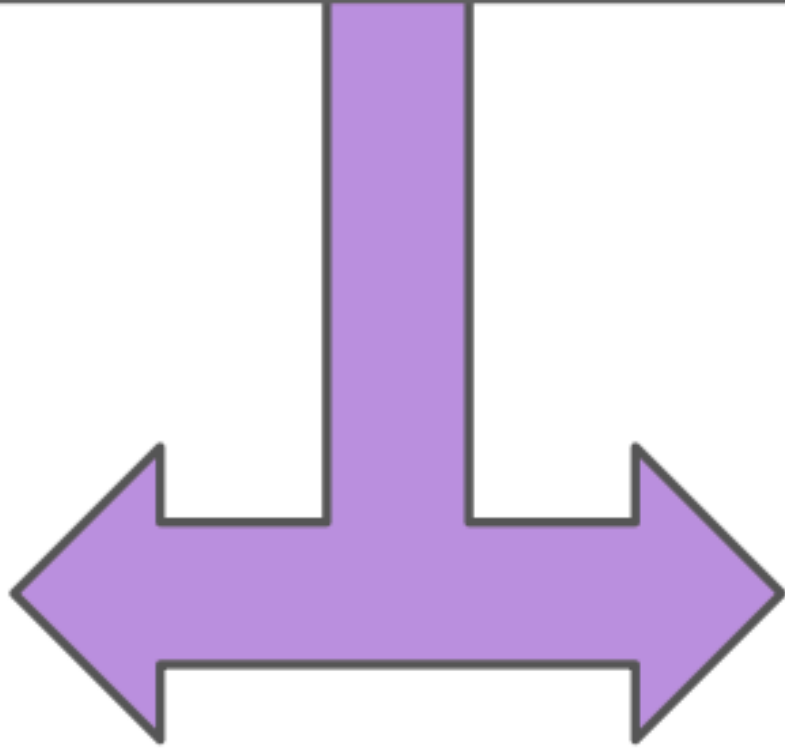


# Minimax Implementation

```
def value(state):  
    if the state is a terminal state: return the state's utility  
    if the next agent is MAX: return max-value(state)  
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):  
    initialize v = -∞  
    for each successor of state:  
        v = max(v, value(successor))  
    return v
```

```
def min-value(state):  
    initialize v = +∞  
    for each successor of state:  
        v = min(v, value(successor))  
    return v
```

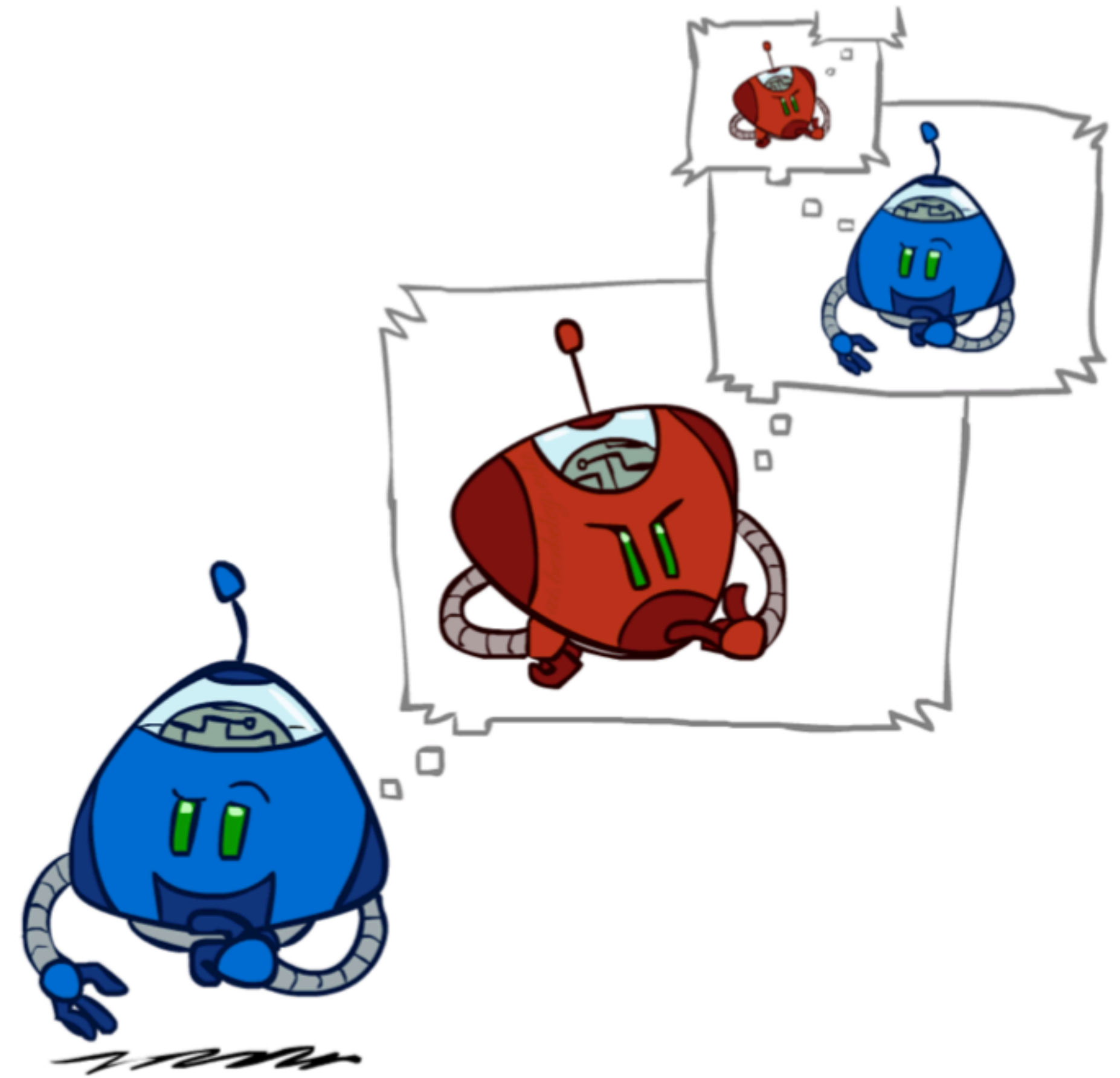


# Properties of Minimax Algorithm

- **Complete:** Minimax algorithm is **complete**. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal:** Minimax algorithm is **optimal** if both opponents are playing optimally.
- **Time complexity:** As it performs DFS for the game-tree, so the time complexity of Minimax algorithm is  $O(b^m)$ , where  $b$  is branching factor of the game-tree, and  $m$  is the maximum depth of the tree.
- **Space complexity:** Space complexity of Mini-max algorithm is also similar to DFS which is  $O(bm)$ .

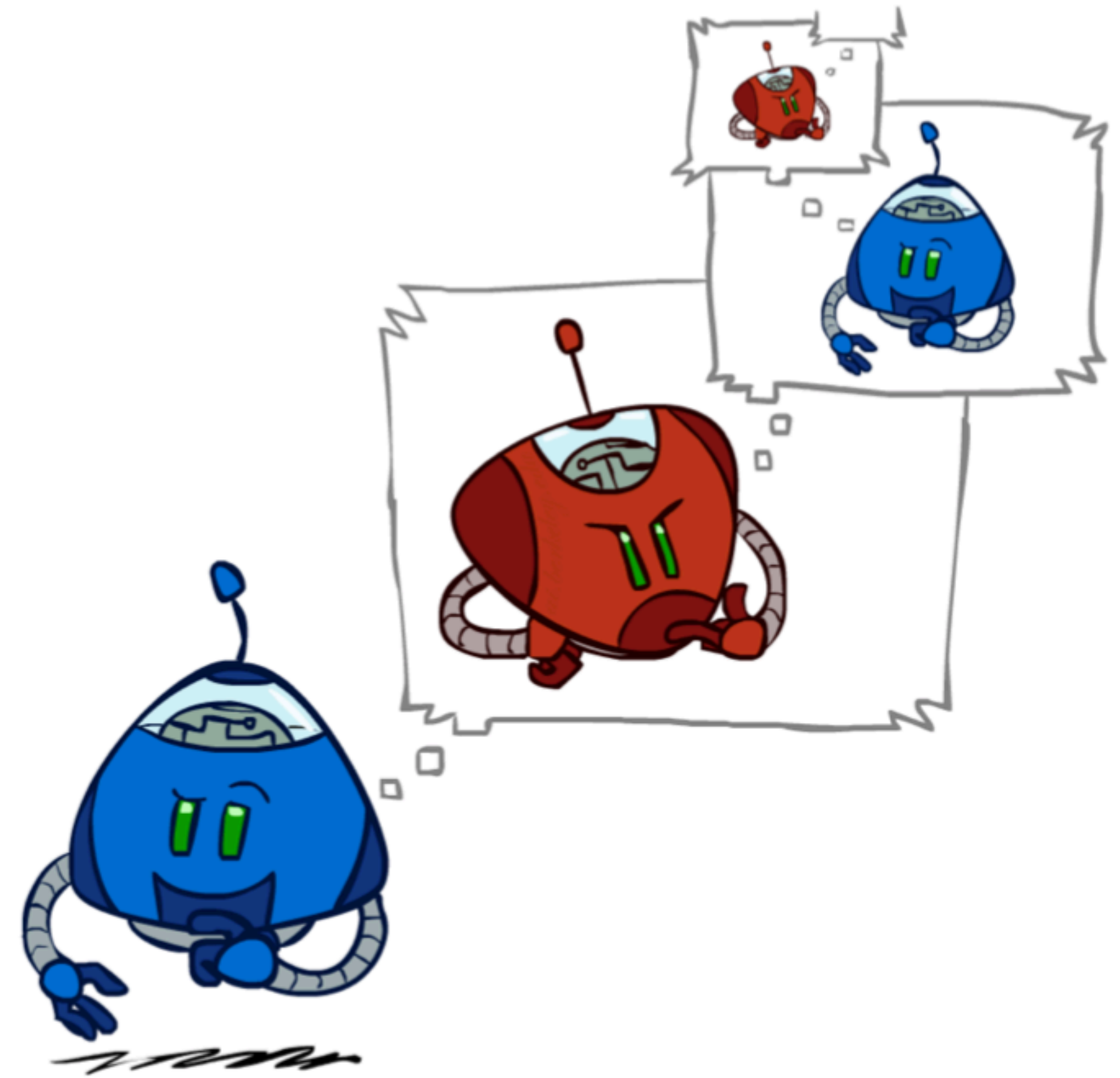
## 22 What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally:
  - **Maximizes worst-case outcome** for MAX.
  - (Classic game theoretic strategy)
- **But if MIN does not play optimally, what will happen?**



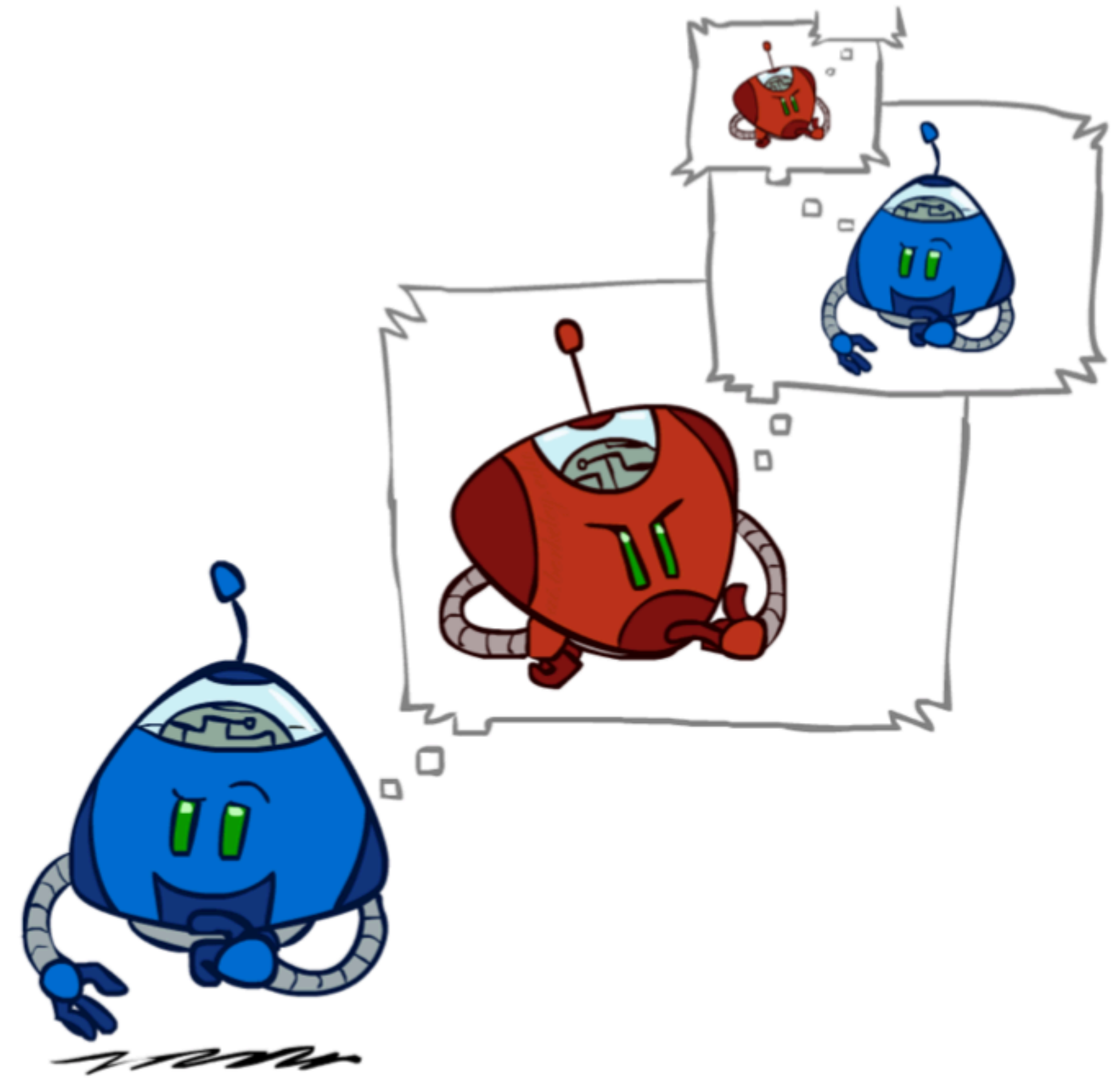
# What if MIN does not play optimally?

- MAX will do even better.
- Consider a MIN node whose children are terminal nodes. If MIN plays suboptimally, then the value of the node is greater than or equal to the value it would have if MIN played optimally. Hence, the value of the MAX node that is the MIN node's parent can only be increased.
- This argument can be extended by a simple induction all the way to the root.
- If the suboptimal play by MIN is predictable, then one can do better than a minimax strategy. For example, if MIN always falls for a certain kind of trap and loses, then setting the trap guarantees a win even if there is actually a devastating response for MIN.



# What if MIN does not play optimally?

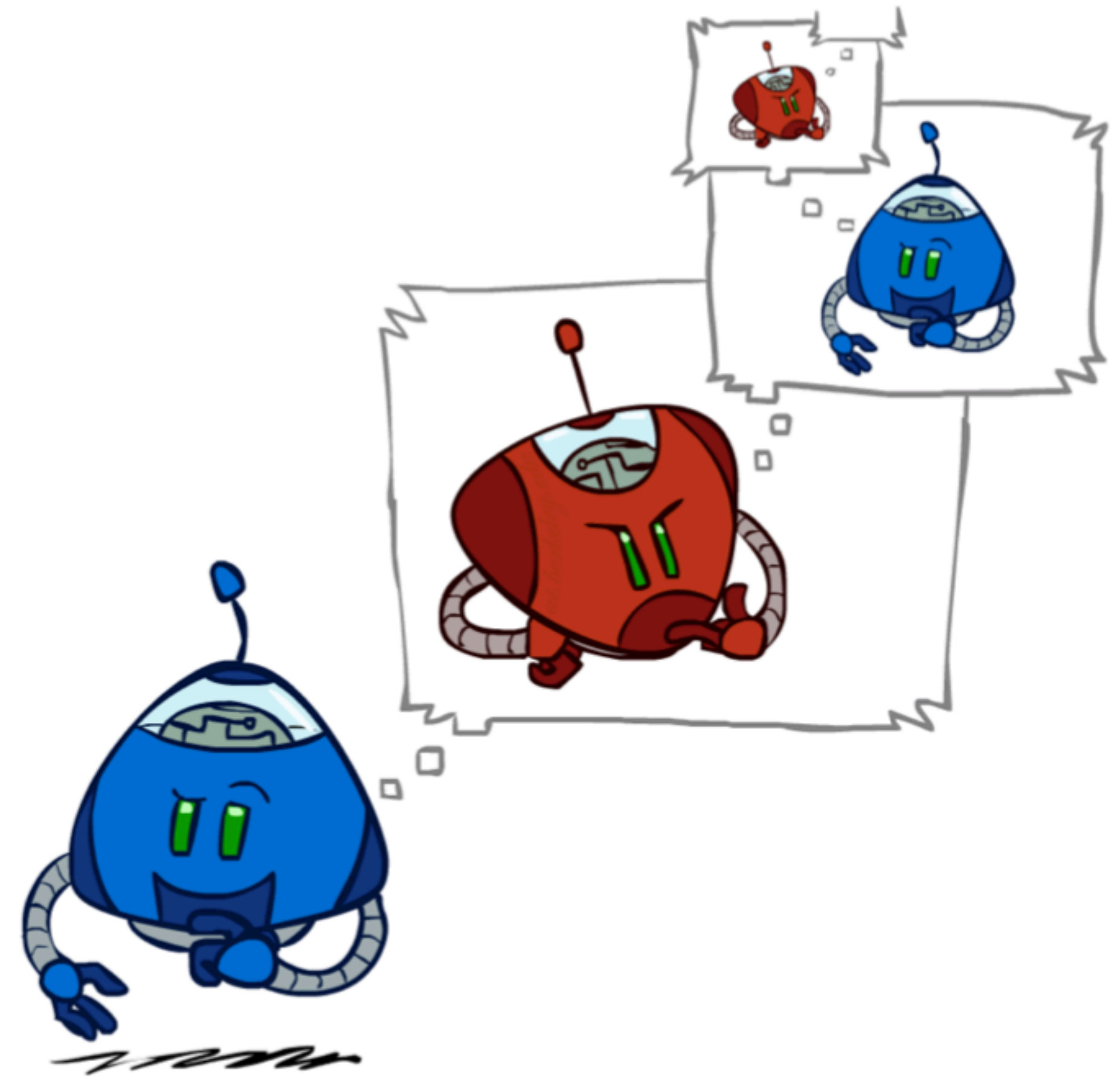
- Is it always best to play the minimax optimal move when facing a suboptimal opponent?





# What if MIN does not play optimally?

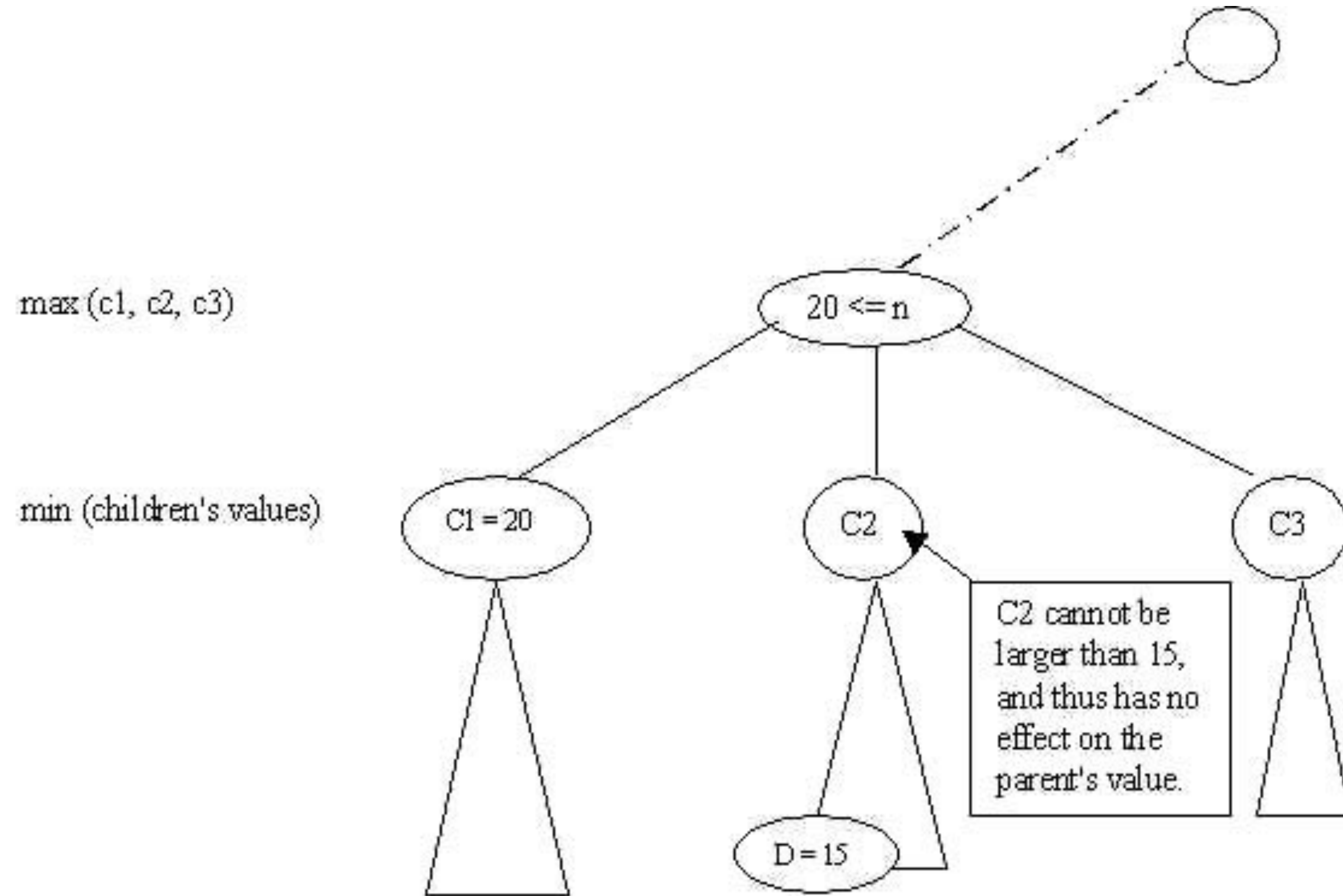
- **Is it always best to play the minimax optimal move when facing a suboptimal opponent? NO**
- Consider a situation where optimal play by both sides will lead to a draw, but there is one risky move for MAX that leads to a state in which there are 10 possible response moves by MIN that all seem reasonable, but 9 of them are a loss for MIN and one is a loss for MAX.
- If MAX believes that MIN does not have sufficient computational power to discover the optimal move, MAX might want to try the risky move, on the grounds that a 9/10 chance of a win is better than a certain draw.



# Comments on Minimax Search

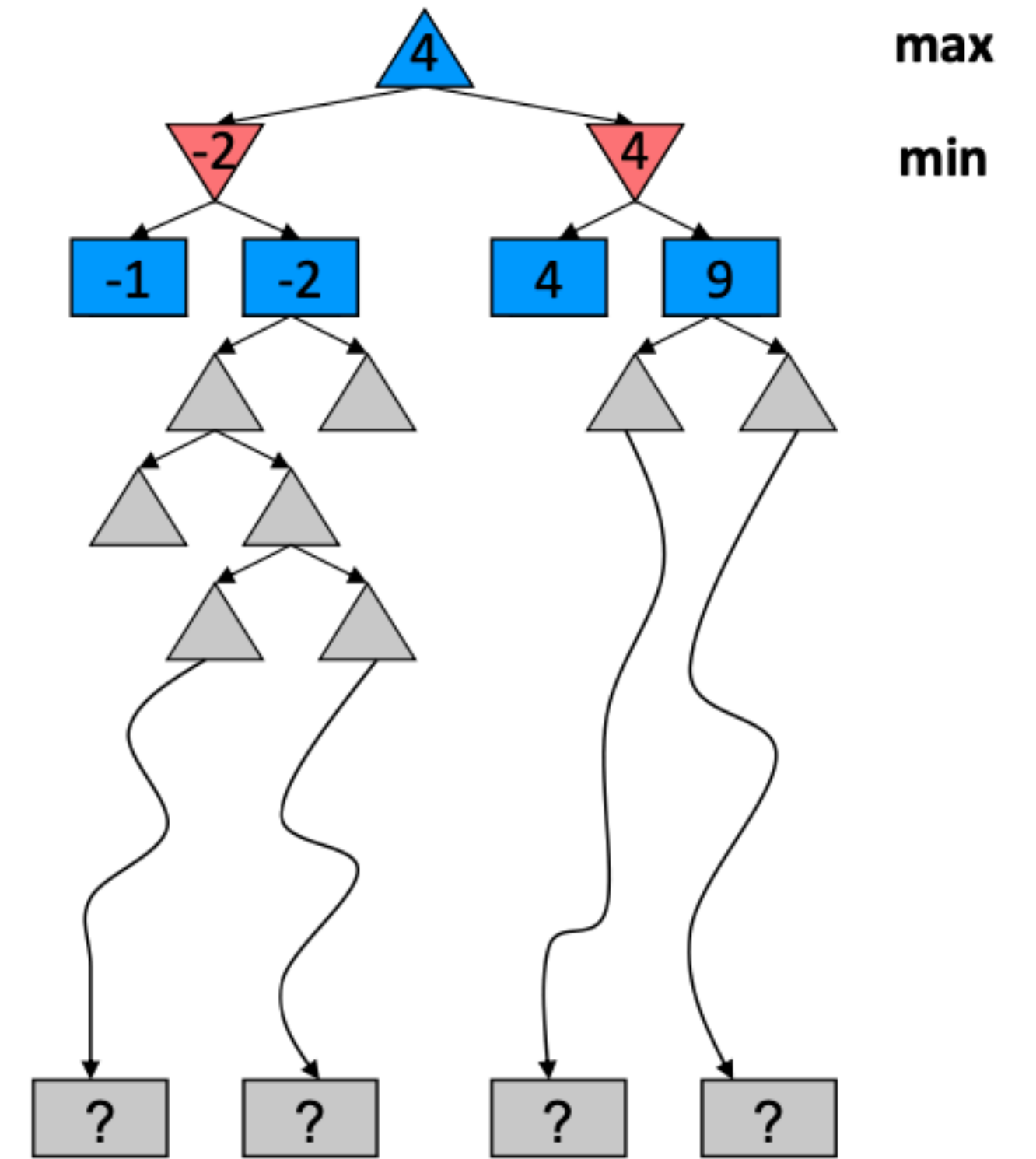
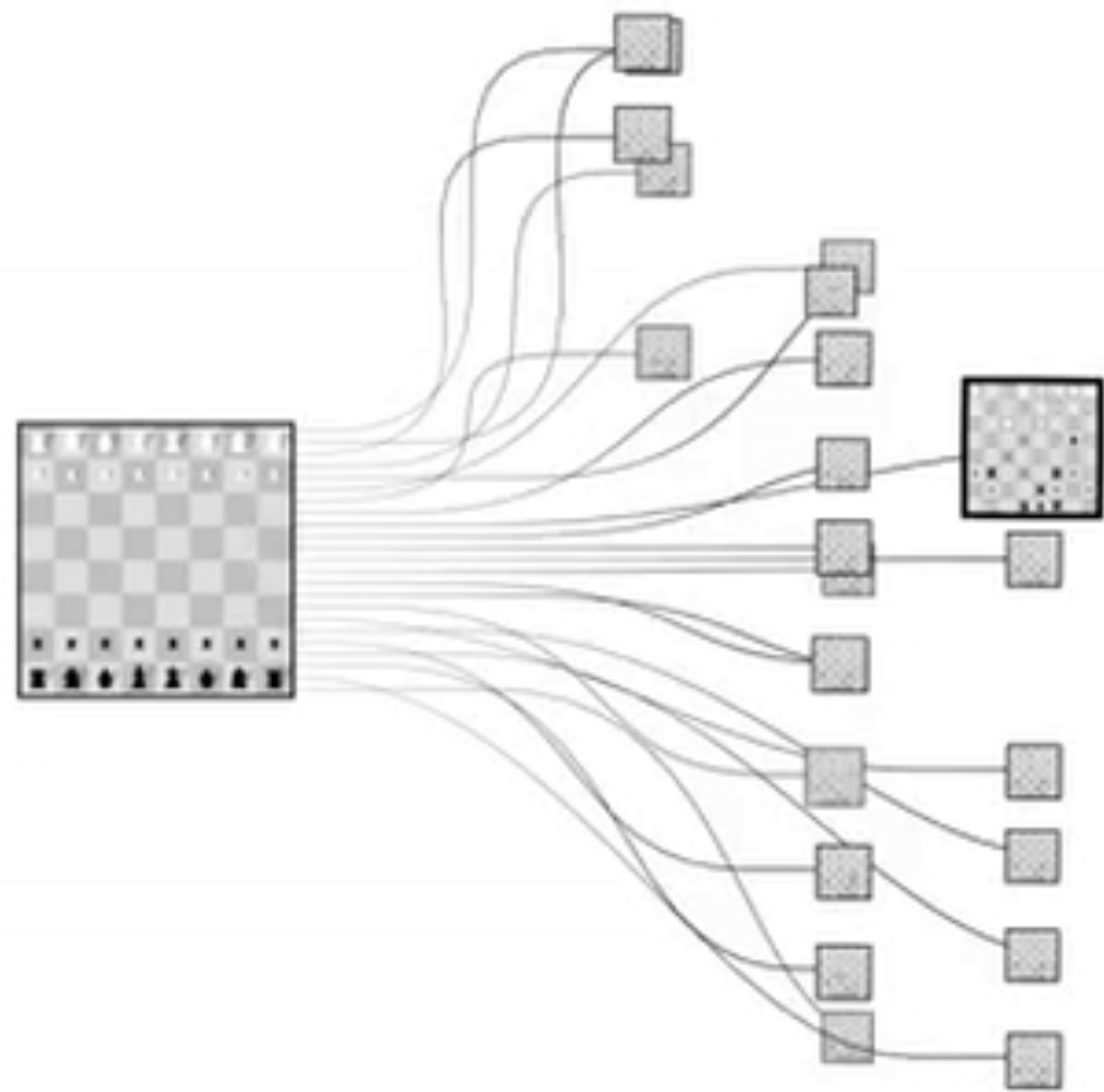
- ◎ **Performance will depend on**
  - the quality of the static evaluation function (expert knowledge)
  - depth of search (computing power and search algorithm)
- ◎ **Differences from normal state space search**
  - Looking to make one move only, despite deeper search
  - No cost on arcs – costs from backed-up static evaluation
  - MAX can't be sure how MIN will respond to his moves
- ◎ **Minimax forms the basis for other game tree search algorithms.**

# Alpha-Beta Pruning

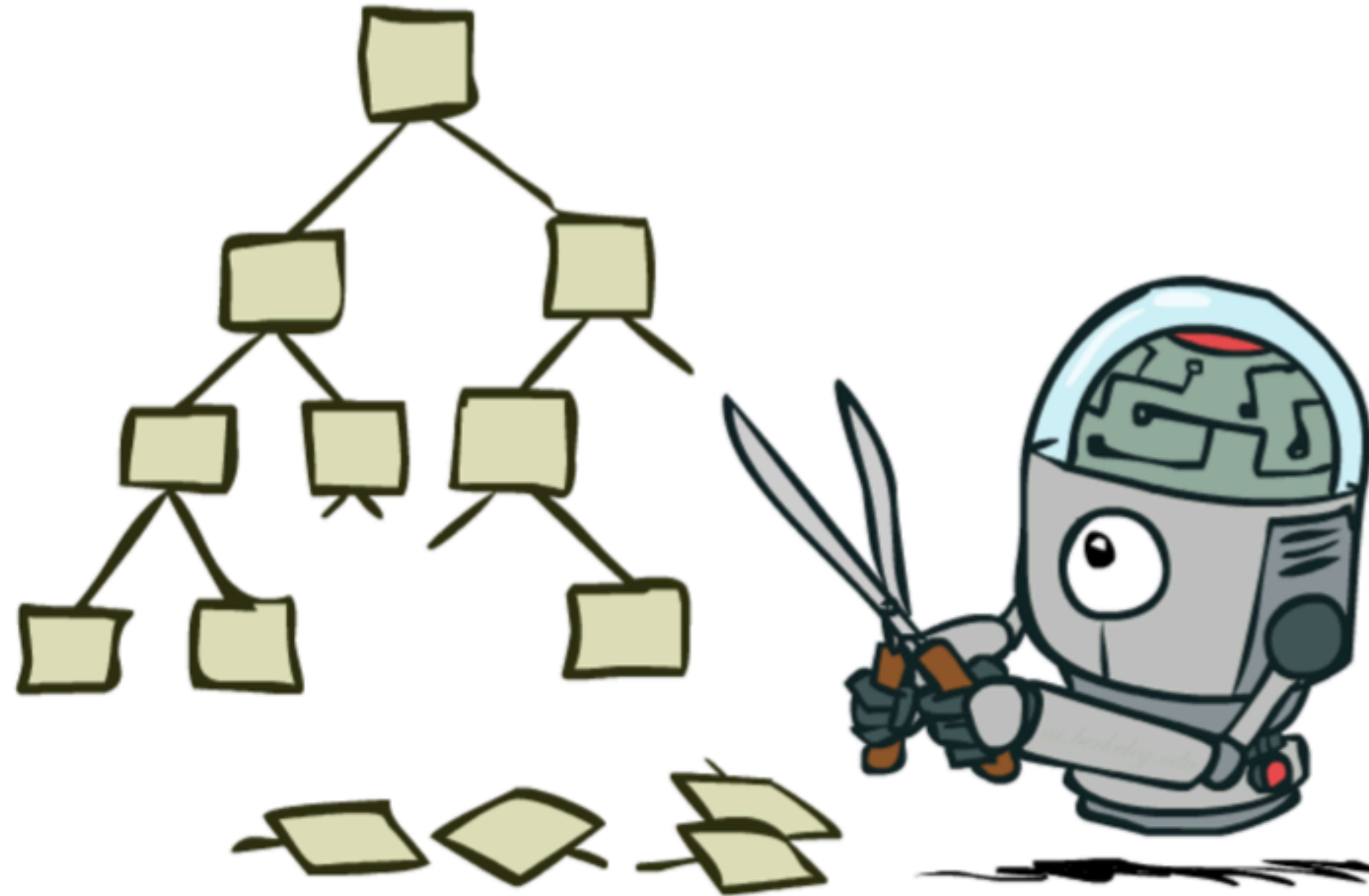


# Resource Limits

- **Problem: In realistic games, cannot search to leaves!**

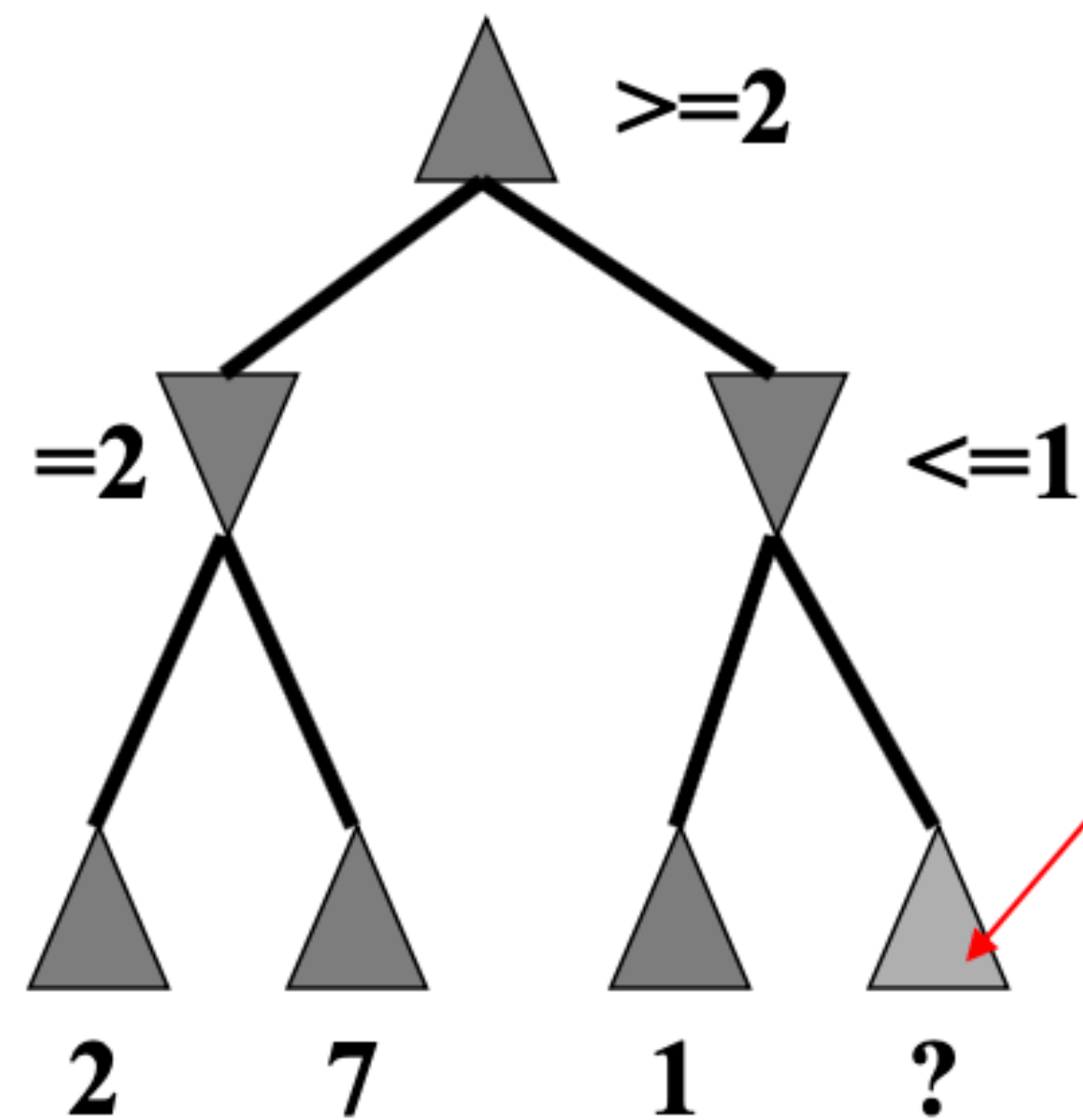


# Game Tree Pruning



# Alpha-Beta Pruning

- A way to improve the performance of the Minimax Procedure
- Basic idea: *“If you have an idea which is surely bad, don’t take the time to see how truly awful it is”* ~ Pat Winston



- We don't need to compute the value at this node.
- No matter what it is it can't effect the value of the root node.

# Alpha-Beta Pruning

- ⊙ During Minimax, **keep track of two additional values:**
  - $\alpha$ : MAX's current **lower** bound on MAX's outcome
  - $\beta$ : MIN's current **upper** bound on MIN's outcome
- ⊙ MAX will never allow a move that could lead to a worse score (for MAX) than  $\alpha$
- ⊙ MIN will never allow a move that could lead to a better score (for MAX) than  $\beta$
- ⊙ Therefore, **stop evaluating a branch whenever:**
  - When evaluating a MAX node: a value  $v \geq \beta$  is backed-up
    - MIN will never select that MAX node
  - When evaluating a MIN node: a value  $v \leq \alpha$  is found
    - MAX will never select that MIN node

# Alpha-Beta Pruning

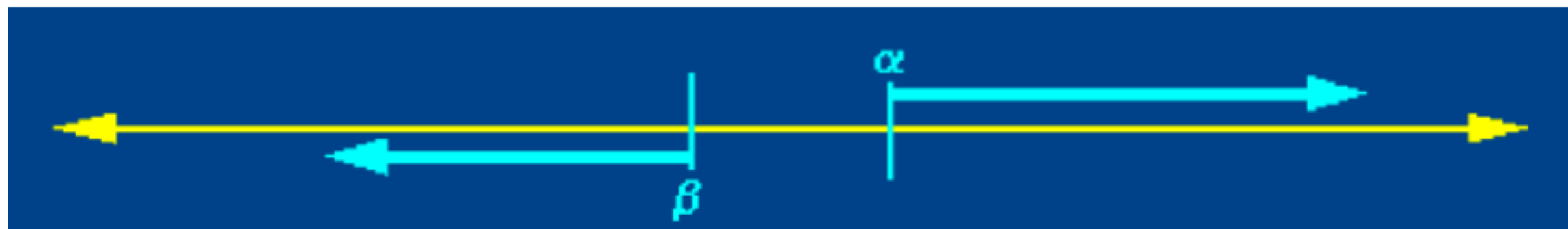
- Based on observation that for all viable paths utility value  $f(n)$  will be  $\alpha \leq f(n) \leq \beta$
- Initially,  $\alpha = -\infty$ ,  $\beta = \infty$



- As the search tree is traversed, the possible utility value window shrinks as  $\alpha$  increases,  $\beta$  decreases



- Whenever the current ranges of alpha and beta no longer overlap, it is clear that the current node is a dead end





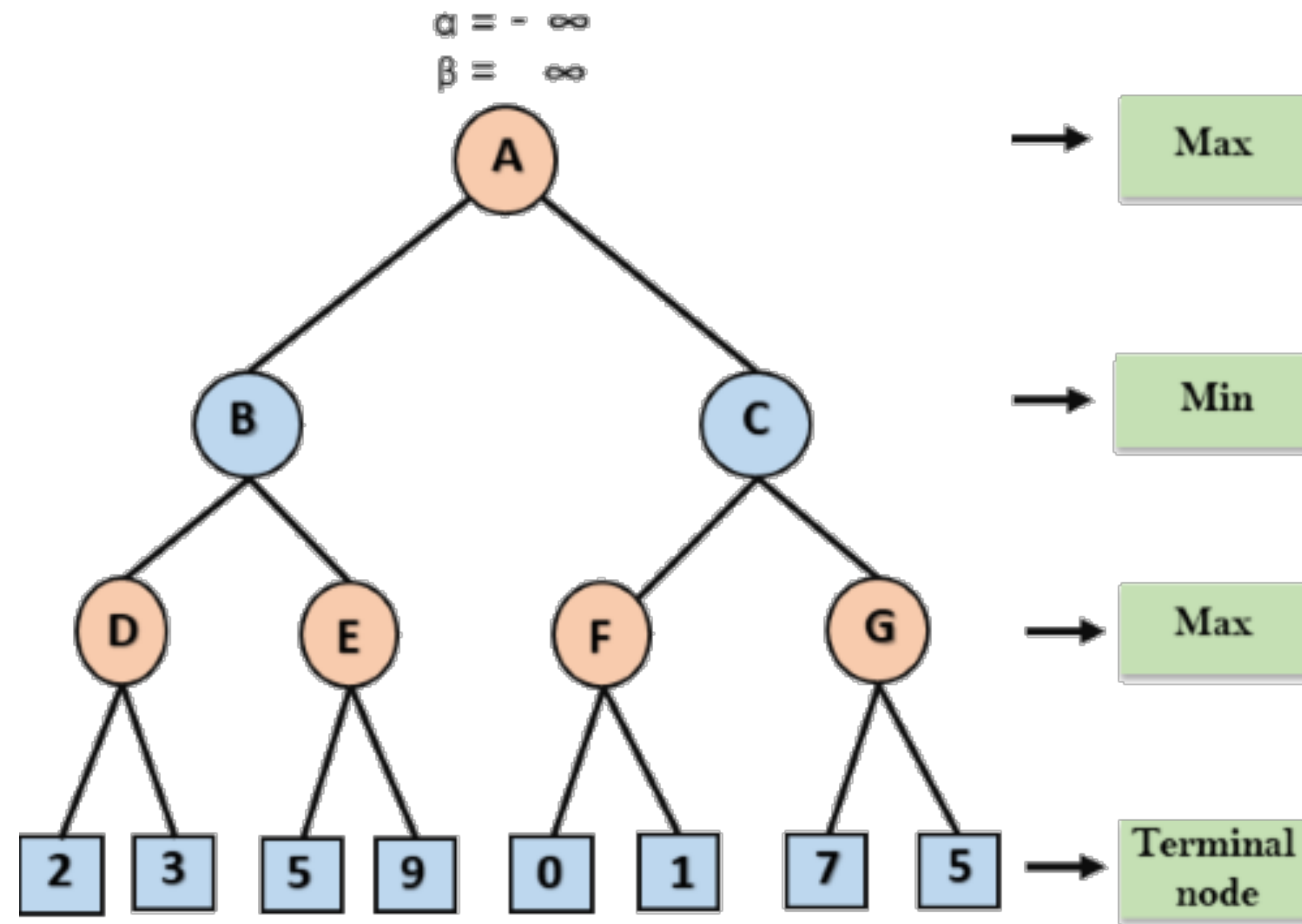
## When to Prune

**Prune whenever  $\alpha \geq \beta$ .**

- ⊙ Prune below a Max node when its  $\alpha$  value becomes  $\geq$  the  $\beta$  value of its ancestors.
  - — **Max nodes update  $\alpha$**  based on children's returned values.
  - — Idea: Player MIN at node above won't pick that value anyway, he can force a worse value.
- ⊙ Prune below a Min node when its  $\beta$  value becomes  $\leq$  the  $\alpha$  value of its ancestors.
  - — **Min nodes update  $\beta$**  based on children's returned values.
  - — Idea: Player MAX at node above won't pick that value anyway; she can do better.

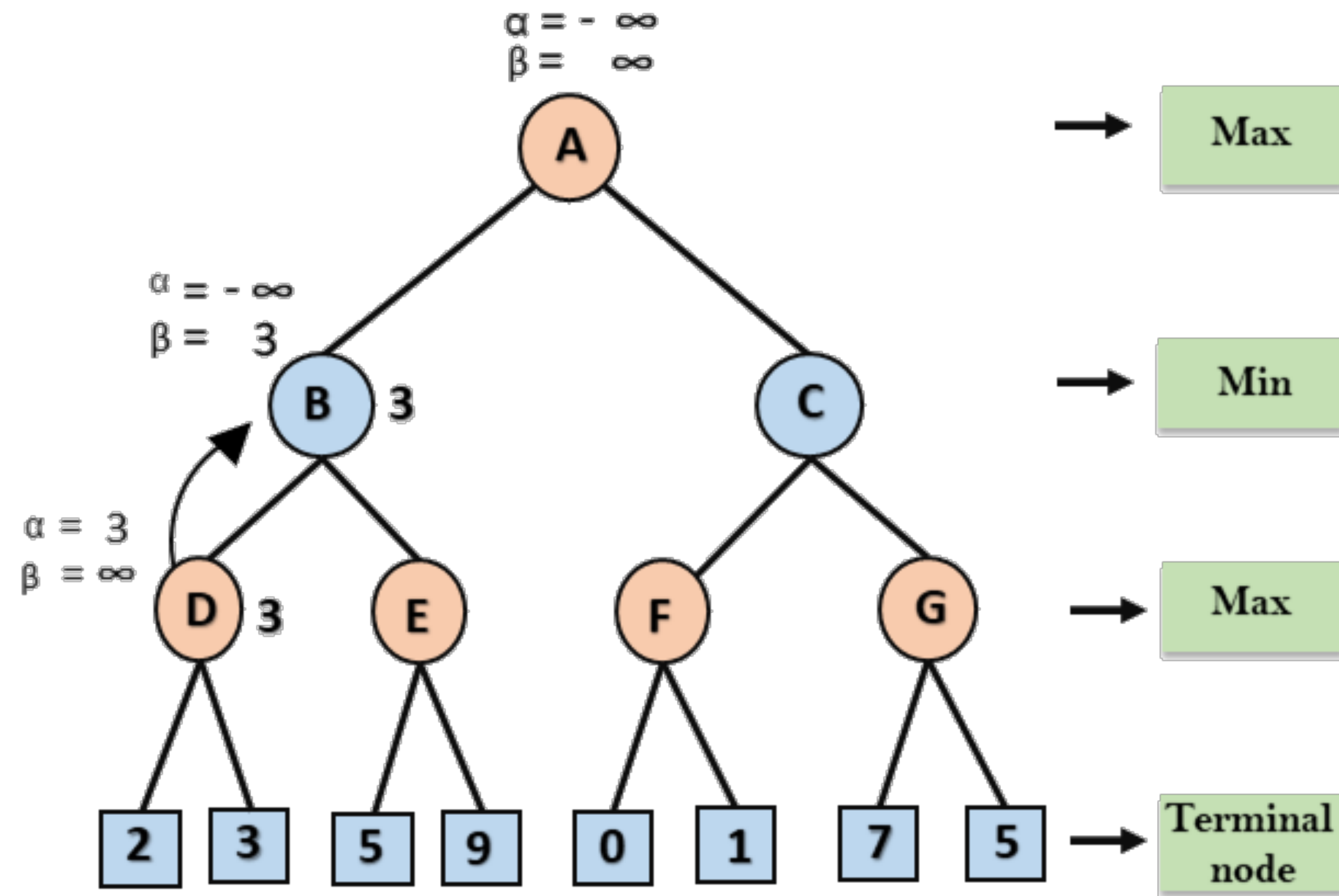
# 34 Example

- Max player will start first move from node A where  $\alpha = -\infty$  and  $\beta = +\infty$ , these value of alpha and beta passed down from A to B, then B to D.



# 35 Example

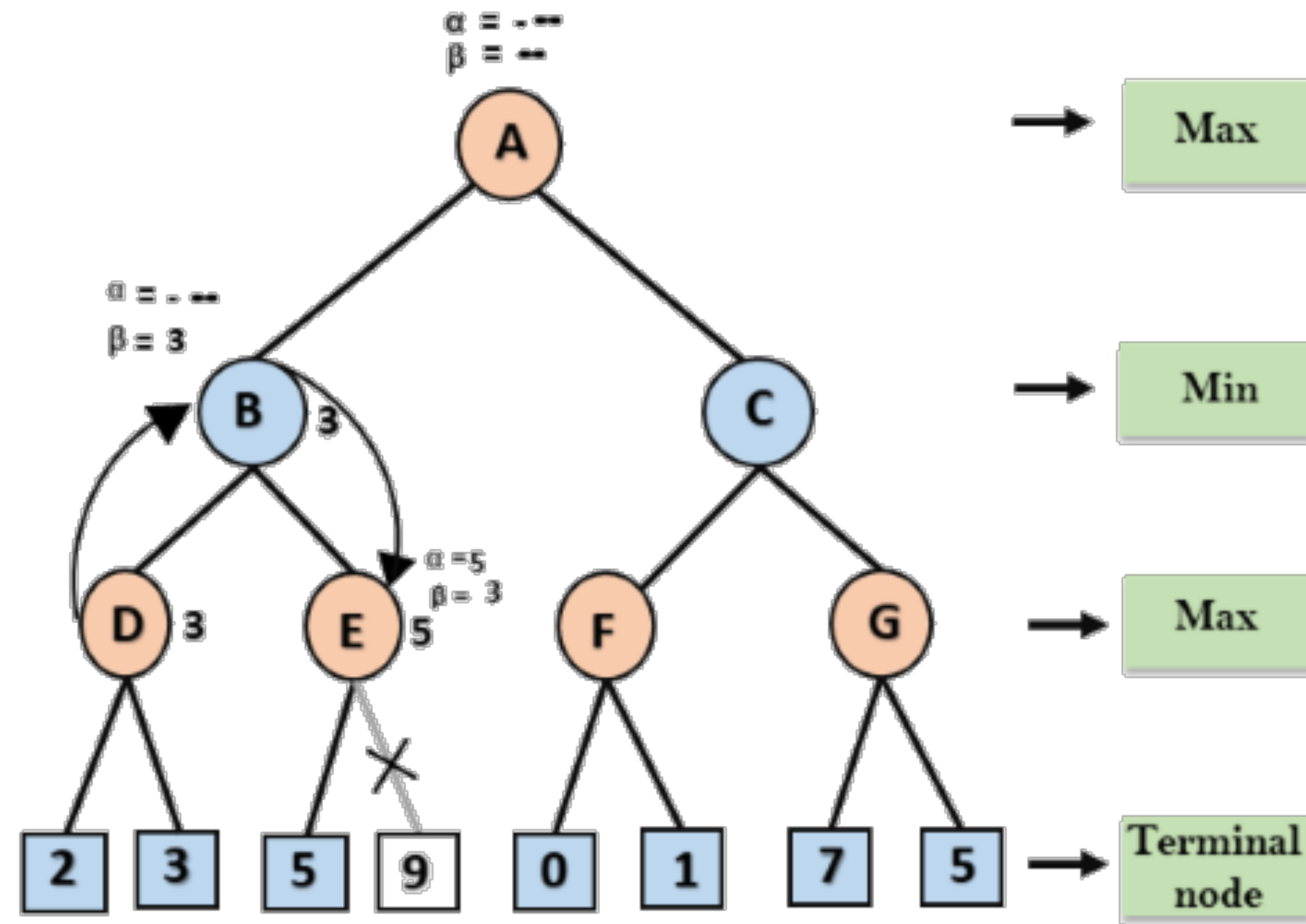
- At Node D, the value of  $\alpha$  will be calculated as its turn for Max:  $\max(2, 3) = 3$  will be the value of  $\alpha$  at node D and node value will also 3. Now algorithm backtrack to node B, where the value of  $\beta$  will change as this is a turn of Min:  $\beta = \min(\infty, 3) = 3$ , hence at node B,  $\alpha = -\infty$  and  $\beta = 3$ .



- In the next step, algorithm traverse the next successor of Node B which is node E, and the values of  $\alpha = -\infty$ , and  $\beta = 3$  will also be passed.

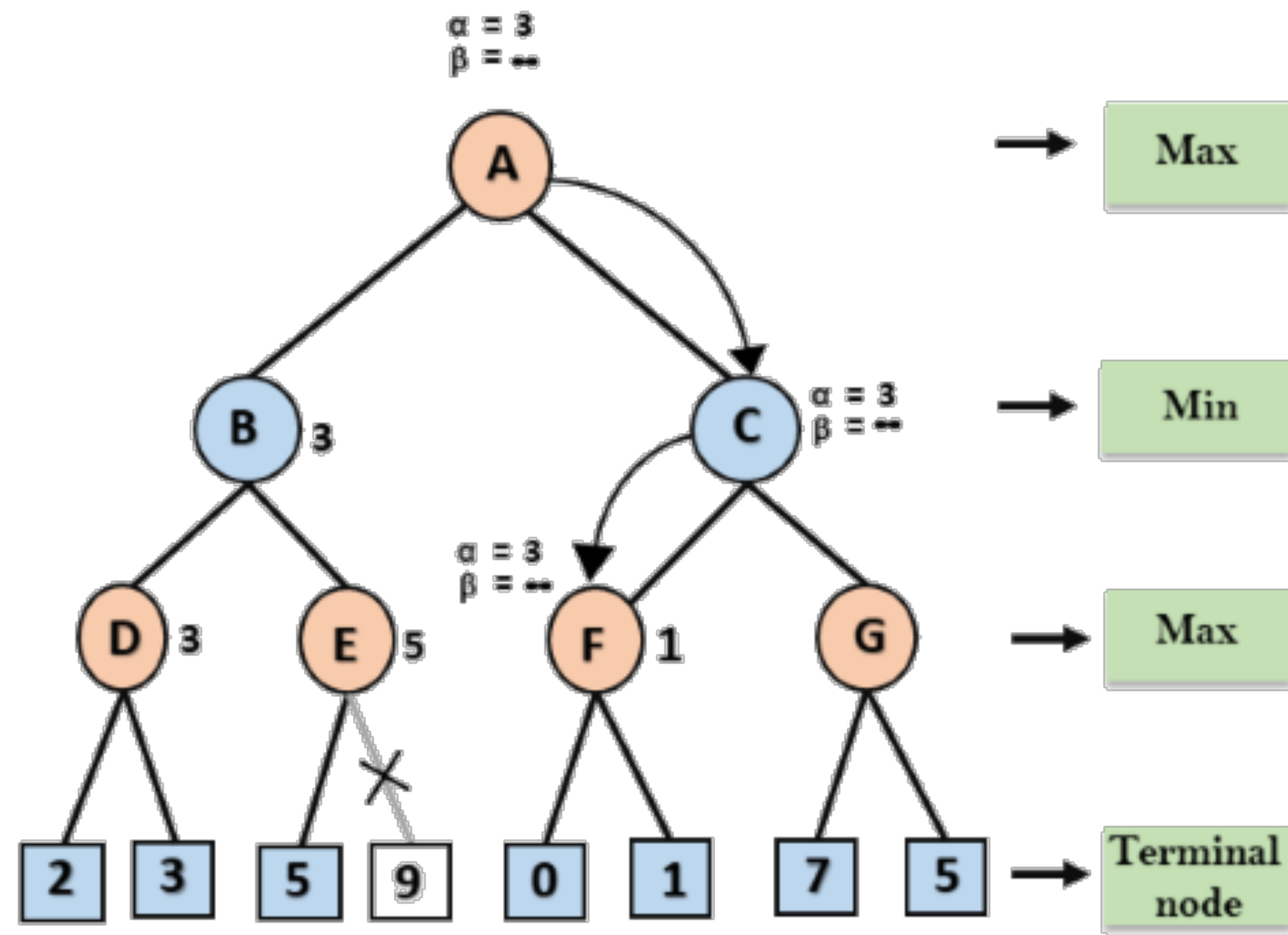
# 36 Example

- At node E, Max will take its turn:  $\alpha = \max(-\infty, 5) = 5$ , hence at node E  $\alpha = 5$  and  $\beta = 3$ , where  $\alpha \geq \beta$ , so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



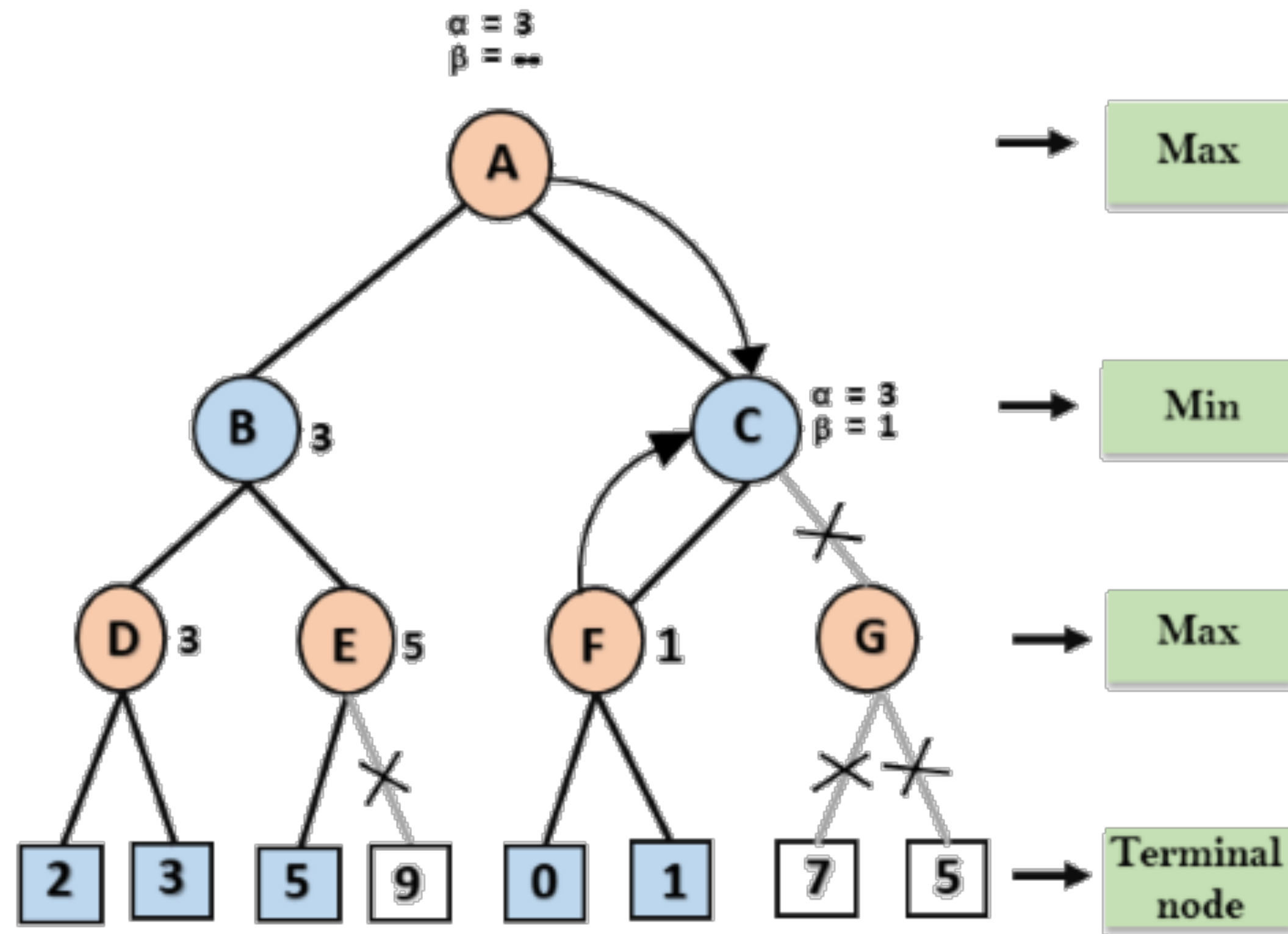
# 37 Example

- Next step, algorithm backtrack the tree, from node B to node A. At node A:  $\alpha = \max(-\infty, 3) = 3$ ,  $\beta = +\infty$ ; then pass to Node C. At node C,  $\alpha = 3$  and  $\beta = +\infty$ , then passed to node F. At node F: compare to left and right child,  $\alpha$  remains 3. The node value of F will become 1.



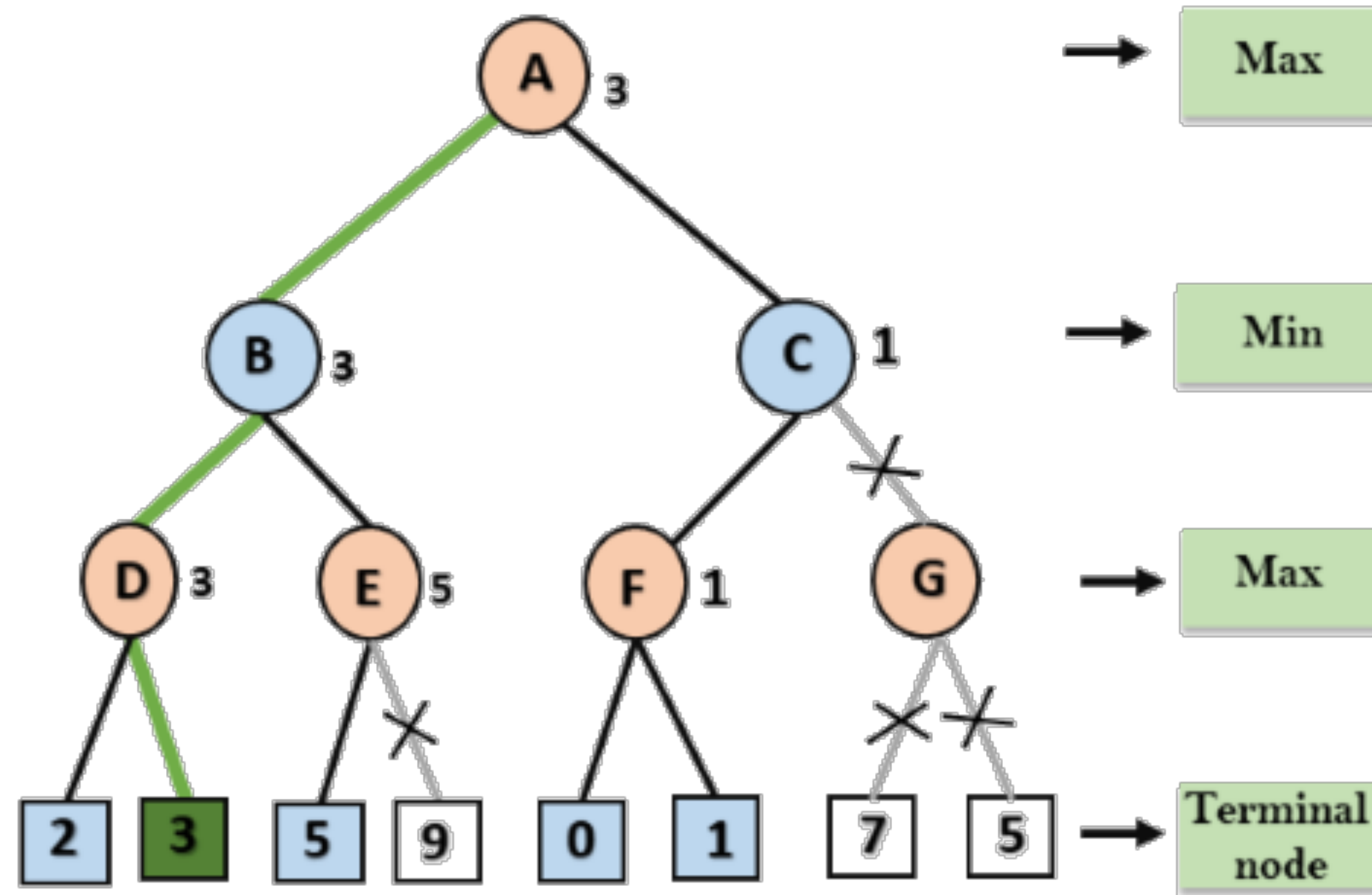
# 38 Example

- Node F returns the node value 1 to node C, at C  $\alpha=3$  and  $\beta=+\infty$ , here the value of beta will be changed, it will compare with 1 so  $\min(\infty, 1) = 1$ . Now at C,  $\alpha=3$  and  $\beta=1$ ,  $\alpha \geq \beta$ , so the next child of C which is G will be pruned. The algorithm will not compute the entire sub-tree G.



# 39 Example

- C now returns the value of 1 to A. The best value for A is  $\max(3, 1) = 3$ . Hence the optimal value for the maximizer is 3 for this example.



# Alpha-Beta Implementation

$\alpha$ : MAX's best option on path to root  
 $\beta$ : MIN's best option on path to root

```
def max-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = -\infty$   
    for each successor of state:  
         $v = \max(v, \text{value}(\text{successor}, \alpha, \beta))$   
        if  $v \geq \beta$  return  $v$   
         $\alpha = \max(\alpha, v)$   
    return  $v$ 
```

```
def min-value(state,  $\alpha$ ,  $\beta$ ):  
    initialize  $v = +\infty$   
    for each successor of state:  
         $v = \min(v, \text{value}(\text{successor}, \alpha, \beta))$   
        if  $v \leq \alpha$  return  $v$   
         $\beta = \min(\beta, v)$   
    return  $v$ 
```

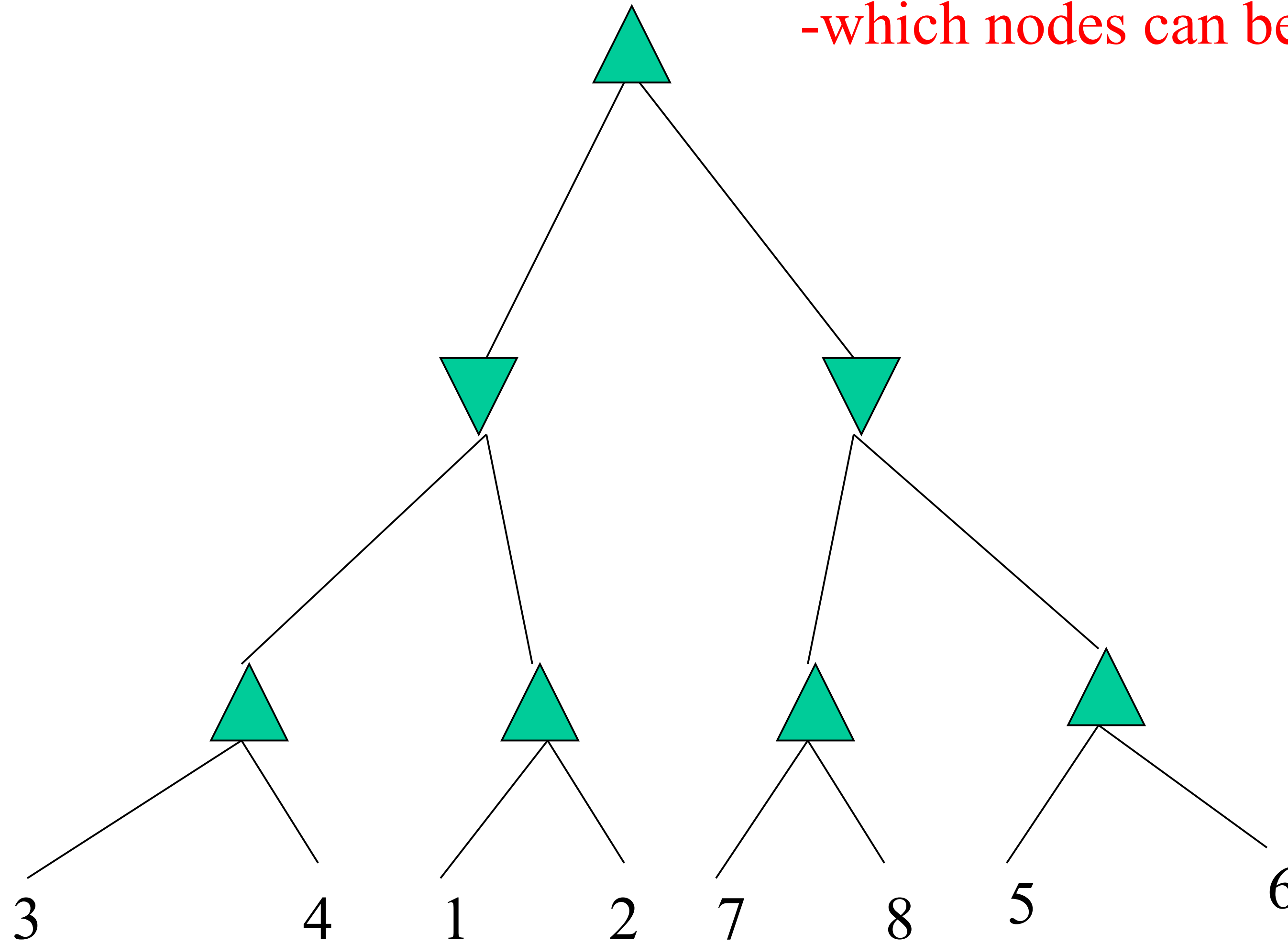


## 41 Move Ordering in Alpha-Beta pruning

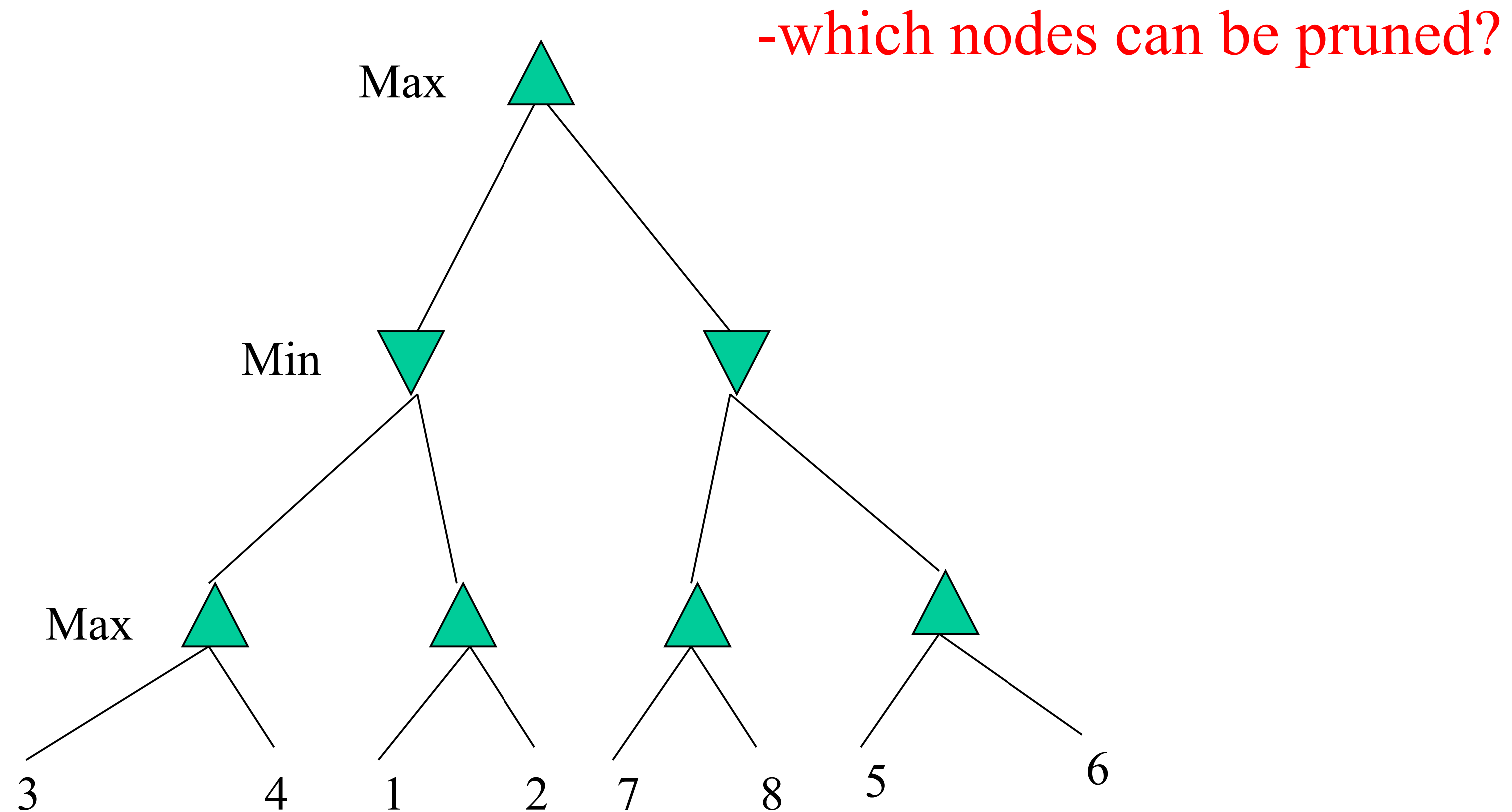
- ⦿ The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined.
- ⦿ **Worst ordering:**
  - In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm.
  - In this case, the best move occurs on the right side of the tree. The time complexity for such an order is  $O(b^m)$ .
- ⦿ **Ideal ordering:**
  - The ideal ordering for alpha-beta pruning occurs when best moves occur at 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. Complexity in ideal ordering is  $O(b^{\frac{m}{2}})$  (Best-Case Analysis of Alpha-Beta Pruning).

42 Test Example...

-which nodes can be pruned?



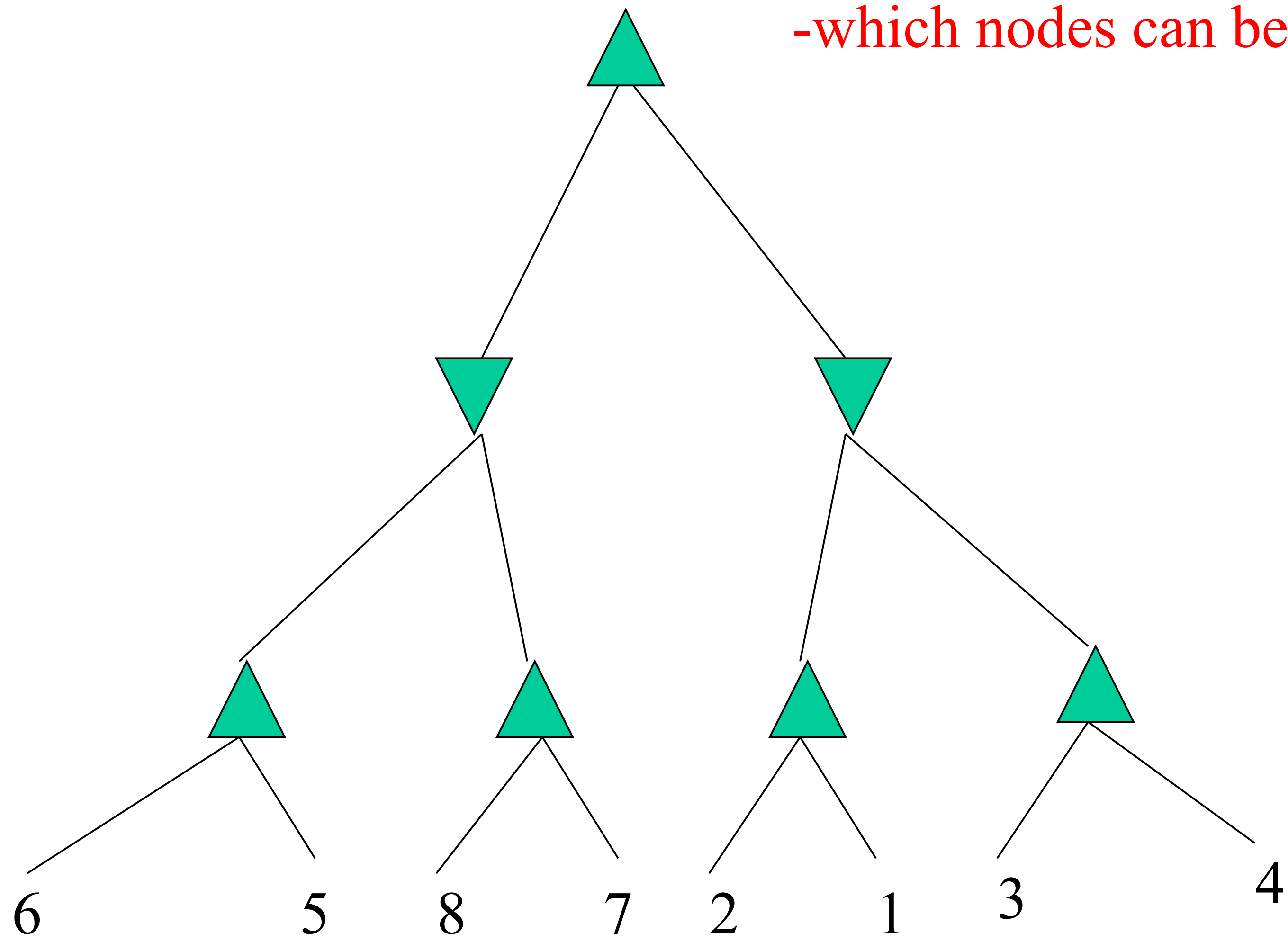
## 43 Test Example...



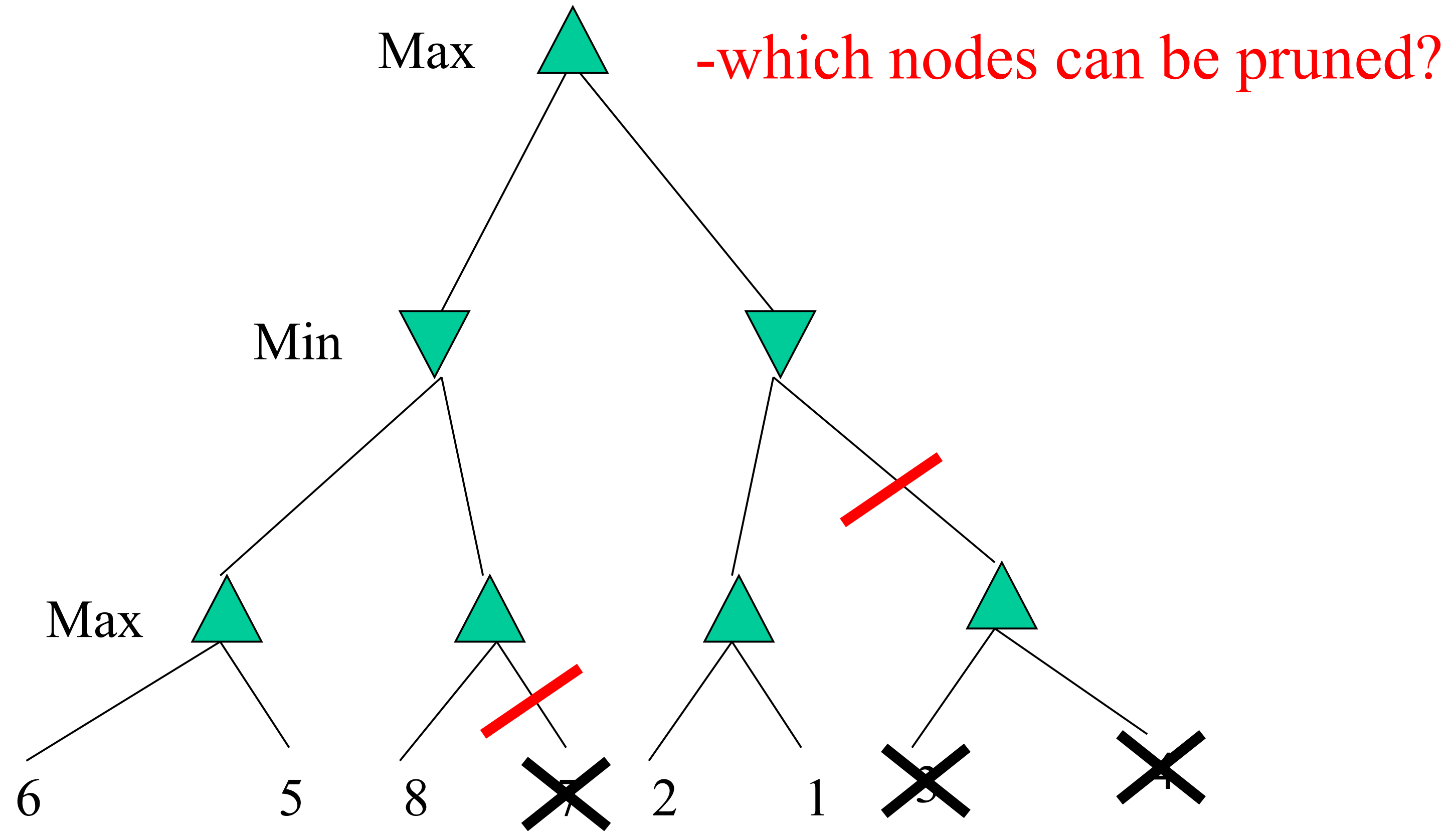
Answer: **NONE!** Because the most favorable nodes for both are explored **last** (i.e., in the diagram, are on the right-hand side).

44 Test Example 2...

-which nodes can be pruned?



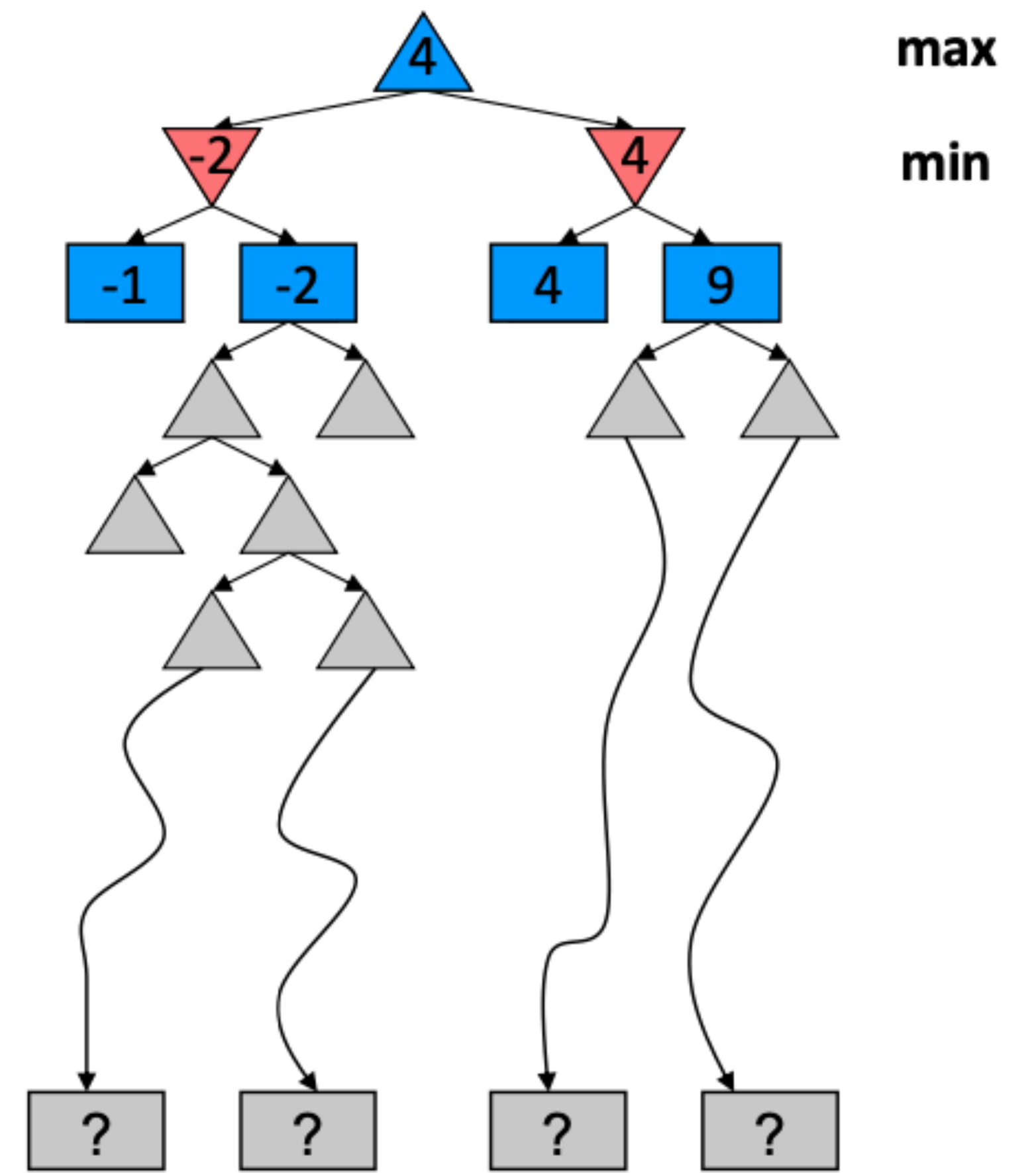
45 Test Example 2...



Answer: **LOTS!** Because the most favorable nodes for both are explored **first** (i.e., in the diagram, are on the left-hand side).

# Resource Limits

- **Problem: In realistic games, cannot search to leaves!**
- **Solution: Depth-limited search**
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - $\alpha$ - $\beta$  reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



## 47 Applications

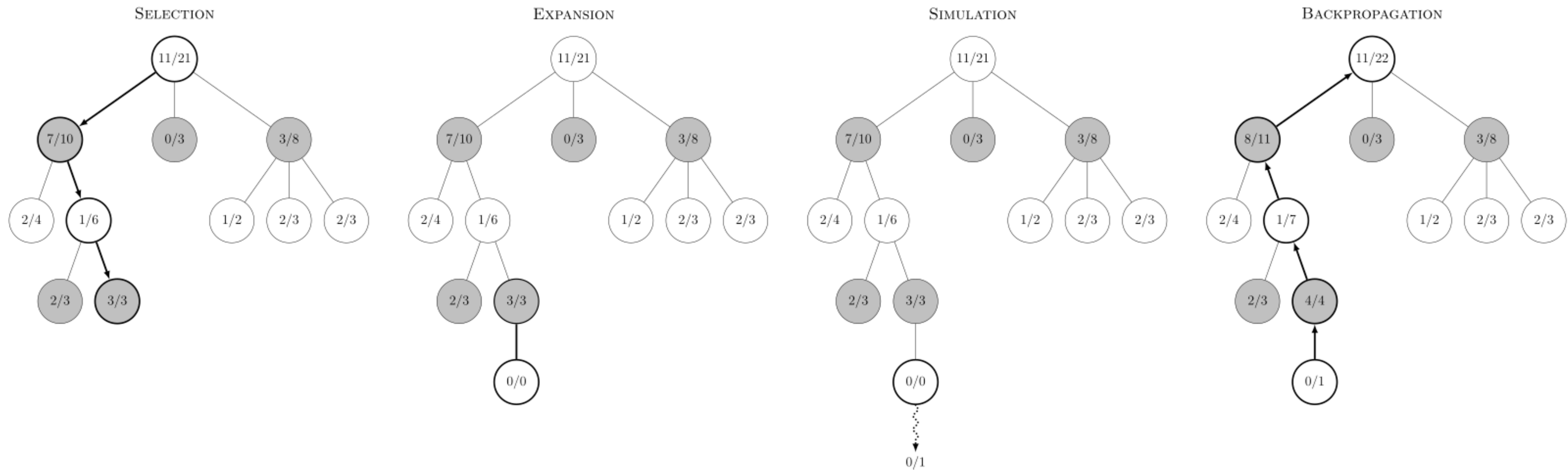
### ◎ Deep Blue

- Evaluate positions using **features handcrafted** by human grandmasters and carefully tuned weights
- Combined with a high-performance **alpha-beta search** that expands a vast search tree using a large number of clever **heuristics** and **domain-specific adaptations**.
- Uses a parallel array of 256 special chess-specific processors
- Evaluates 200 billion moves every 3 minutes; 12-ply search depth
- 8000 factor evaluation function tuned from hundreds of thousands of grandmaster games
- Tends to play for tiny positional advantages.

### ◎ Chinook

- The World Man-Made Checkers Champion, developed at the University of Alberta.
- Competed in human tournaments, earning the right to play for the human world championship, and defeated the best players in the world.

# Monte-Carlo Tree Search





# The Game of Go

- For quite a long time, a common opinion in academic world was that machine achieving human master performance level in the game of Go was far from realistic.
- It was considered a ‘holy grail’ of AI – a milestone we were quite far away from reaching within upcoming decade.
- Surprisingly, in march 2016 an algorithm invented by Google DeepMind called **Alpha Go** **defeated Korean world champion in Go 4-1** proving fictional and real-life skeptics wrong.
- Around a year after that, **Alpha Go Zero** – the next generation of **Alpha Go Lee** (the one beating Korean master) – was **reported to destroy its predecessor 100-0**, being very doubtfully reachable for humans.



# AlphaGo

- 2016: AlphaGO (created by DeepMind) defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.



<https://www.alphagomovie.com/>



# AlphaZero, MuZero, and More...



MuZero: Mastering Go, chess, shogi and Atari without rules

# Alpha Go/Zero

- ◎ **Alpha Go/Zero** system is a mix of several methods assembled into one great engineering piece of work. **The core components of the Alpha Go/Zero** are:
  - **Monte Carlo Tree Search** (certain variant with PUCT function for tree traversal)
  - Residual Convolutional **Neural Networks** – policy and value network(s) used for game evaluation and move prior probability estimation
  - **Reinforcement learning** used for training the network(s) via self-plays
- ◎ Here we will focus on **Monte Carlo Tree Search** only.

# Monte Carlo Tree Search

- Monte Carlo Tree Search was introduced by Rémi Coulom in 2006 as a building block of Crazy Stone – Go playing engine with an impressive performance.



- From a helicopter view Monte Carlo Tree Search has one main purpose: given a **game state** to choose **the most promising next move**.

# Monte Carlo Tree Search

## 1. Selection

Start from root  $R$  and select successive child nodes until a leaf node  $L$  is reached. The root is the current game state and a leaf is any node that has a potential child from which no simulation (playout) has yet been initiated.

## 2. Expansion

Unless  $L$  ends the game decisively (e.g. win/loss/draw) for either player, create one (or more) child nodes and choose node  $C$  from one of them. Child nodes are any valid moves from the game position defined by  $L$ .

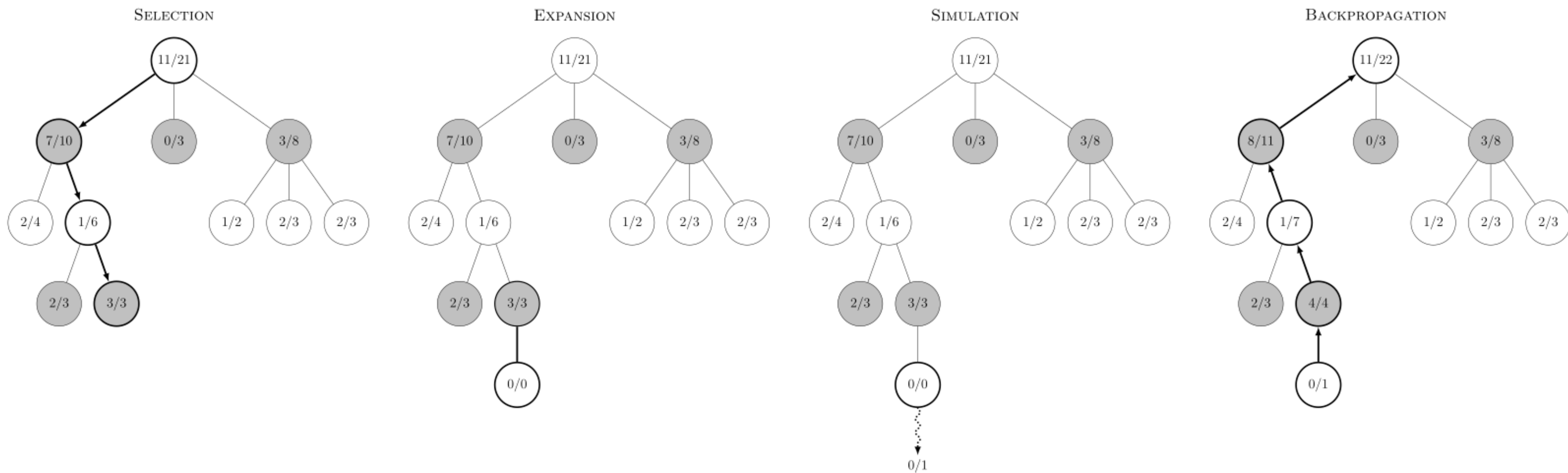
## 3. Simulation

Complete one random playout from node  $C$ . This step is sometimes also called playout or rollout. A playout may be as simple as choosing uniform random moves until the game is decided (for example in chess, the game is won, lost, or drawn).

## 4. Backpropagation

Use the result of the playout to update information in the nodes on the path from  $C$  to  $R$ .

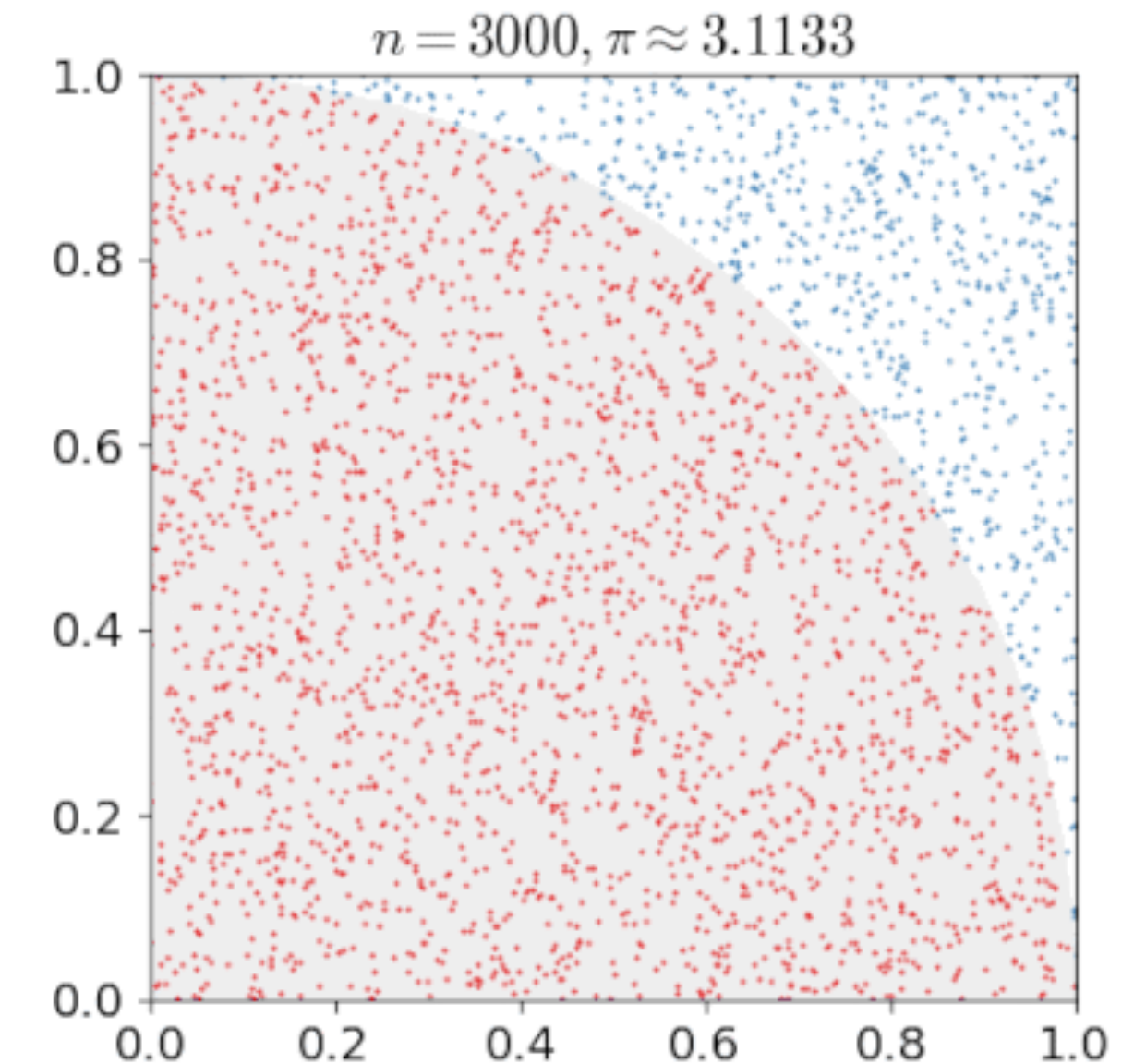
# 56 Monte Carlo Tree Search





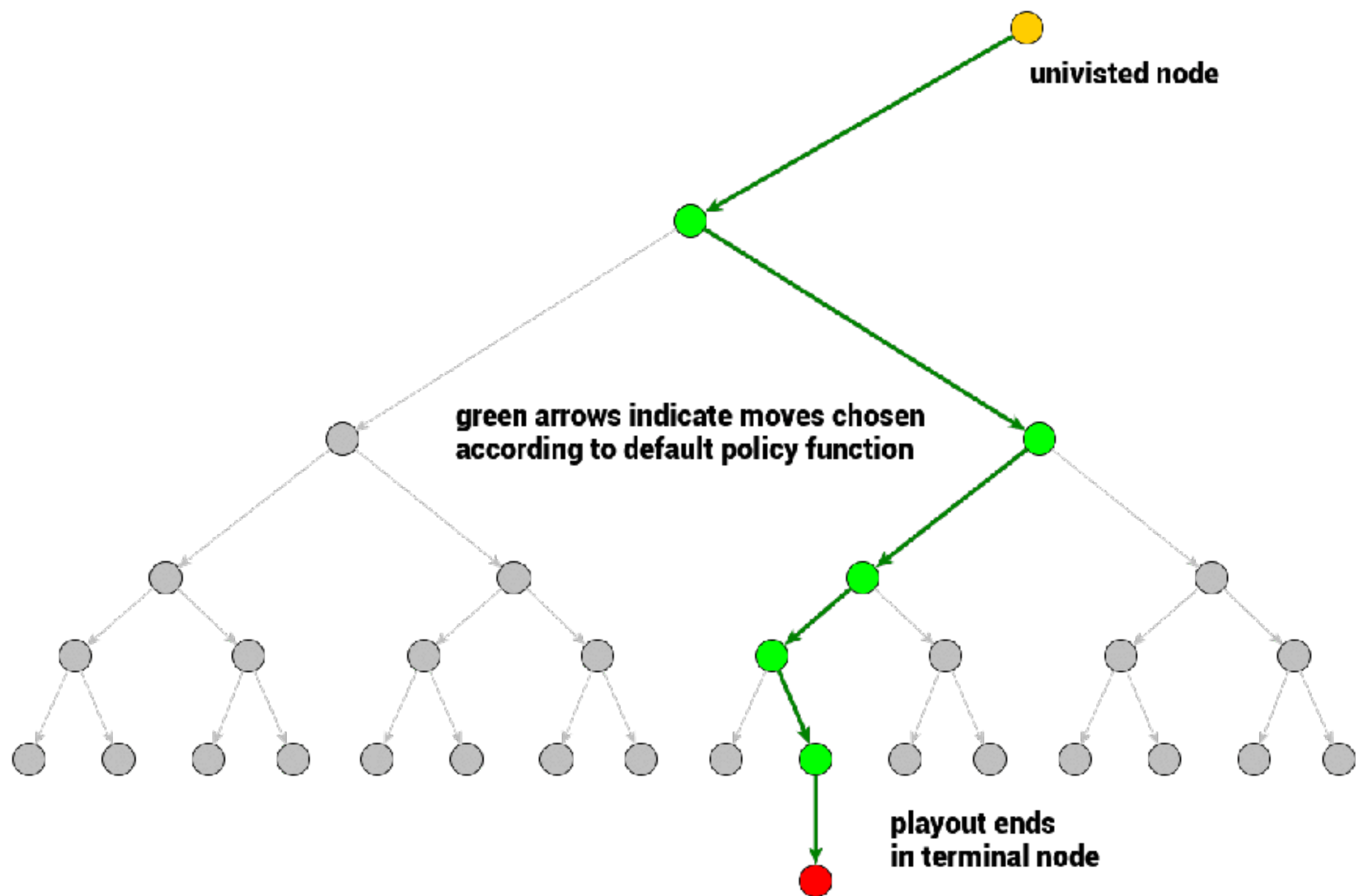
## 57 Choosing the Most Promising Move: Monte Carlo

- In Monte Carlo Tree Search algorithm, **the most promising move** is computed in a slightly different fashion.
- As the name suggests (especially its monte-carlo component) – Monte Carlo Tree Search **simulates the games many times** and tries to predict the most promising move based on the simulation results.
- **Monte Carlo method:**
  - A broad class of computational algorithms that rely on **repeated random sampling** to obtain numerical results.
  - The underlying concept is to use randomness to solve problems that might be deterministic in principle.



*Monte Carlo method applied to approximating the value of  $\pi$ .*

# Simulation / Playout



## Playout/Simulation is Alpha Go and Alpha Zero

In **Alpha Go Lee** evaluation of the leaf  $S_L$  is a weighted sum of two components:

- standard rollout evaluation  $z_L$  with custom **fast rollout policy** that is a shallow softmax neural network with handcrafted features
- position evaluation given by 13-layer convolutional neural network  $v_0$  called **Value Network** trained on 30mln of distinct (no two positions come from the same game) positions extracted from Alpha Go self-plays

$$V(S_L) = (1 - \alpha)v_0(S_L) + \alpha z_L$$

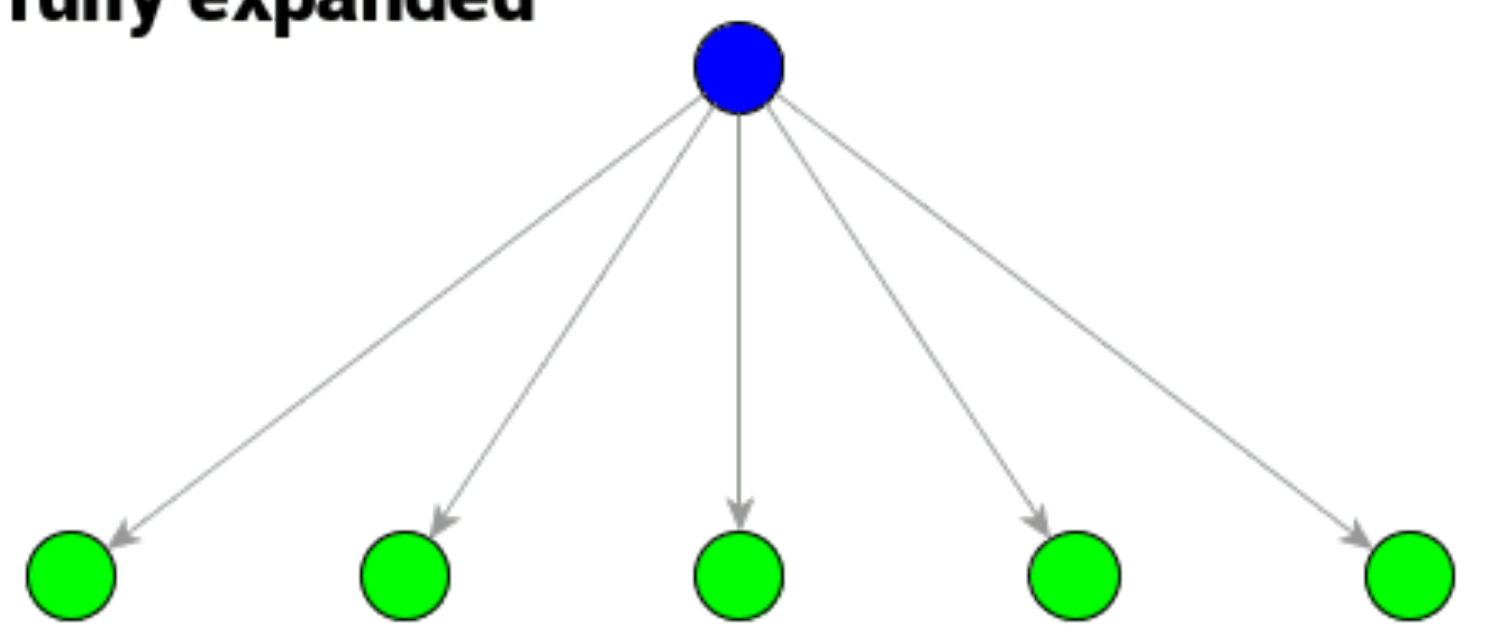
In **Alpha Zero** Deepmind's engineers went a step further, they **do not perform playouts at all**, but directly evaluate the current node with a 19-layer CNN Residual neural network (In Alpha Zero they have one network  $f_0$  that outputs position evaluation and moves probability vector at once)

$$V(S_L) = f_0(S_L)$$

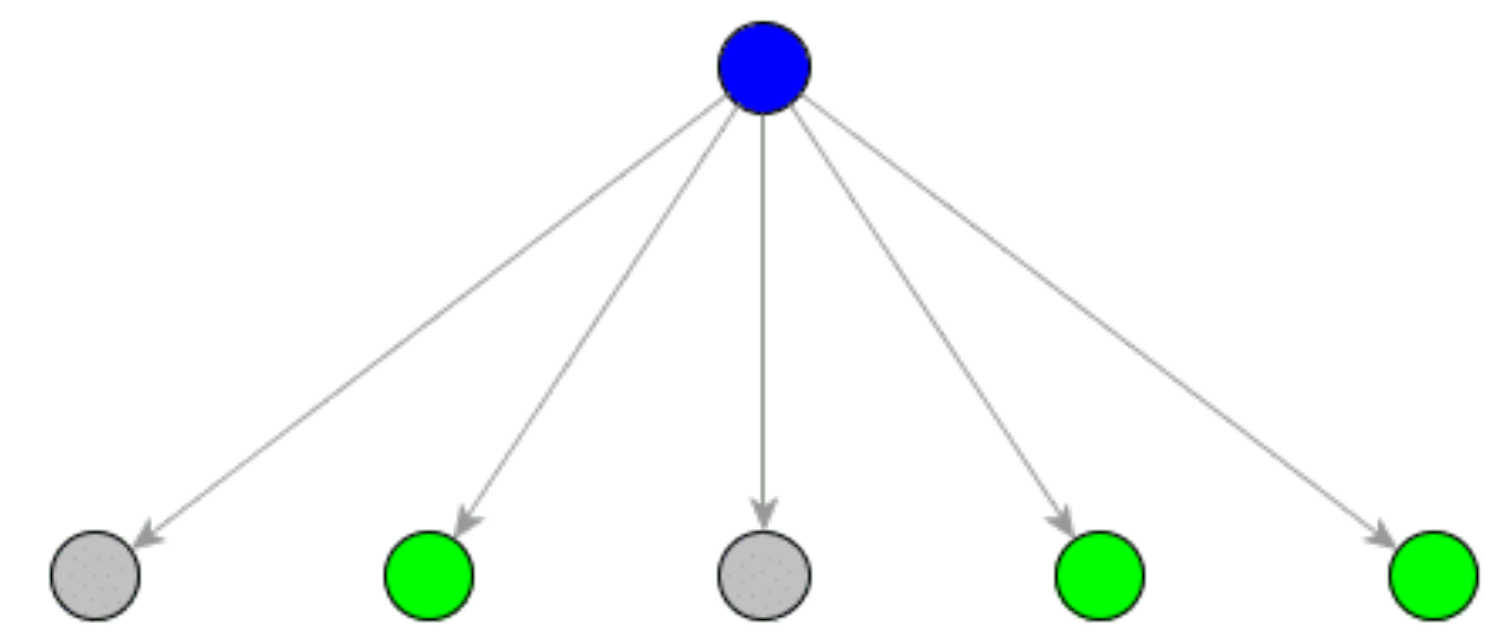
# Fully expanded and visited nodes

- Node is considered visited **if a playout has been started in that node** – meaning it has been evaluated at least once.
- If all children nodes of a node are visited node is considered **fully expanded**, otherwise – well – it is not fully expanded and further expansion is possible.
- nodes chosen** by rollout policy function **during simulation** are **not considered visited**.

**all children are marked visited - node is fully expanded**



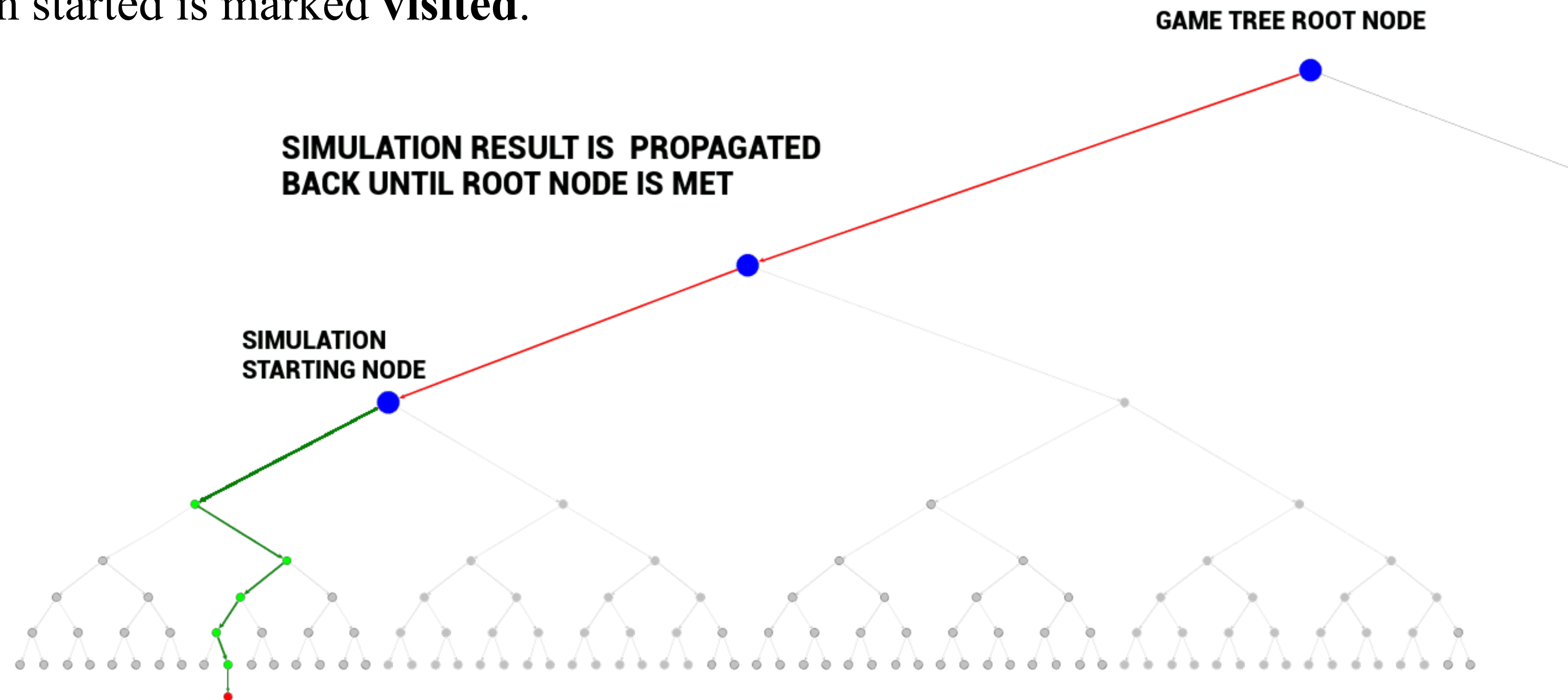
**simulation/game state evaluation has been computed in all green nodes, they are marked visited**



**there are two nodes from where no single simulation has started - these nodes are unvisited, parent is not fully expanded**

## 60 Backpropagation

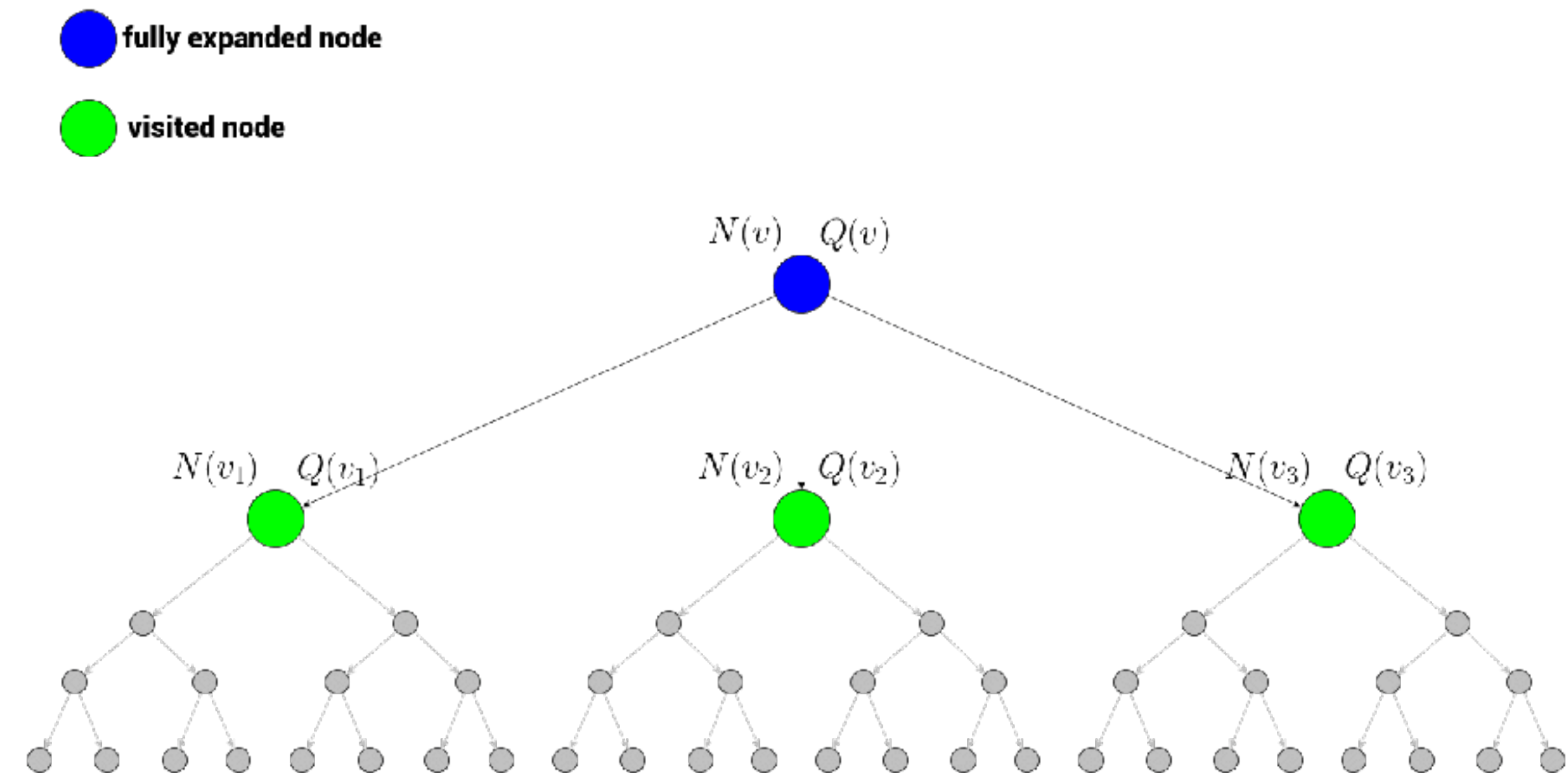
- Once simulation for a freshly visited node (sometimes called a **leaf**) is finished, its result is **ready to be propagated back** up to the current game tree root node. The node where simulation started is marked **visited**.



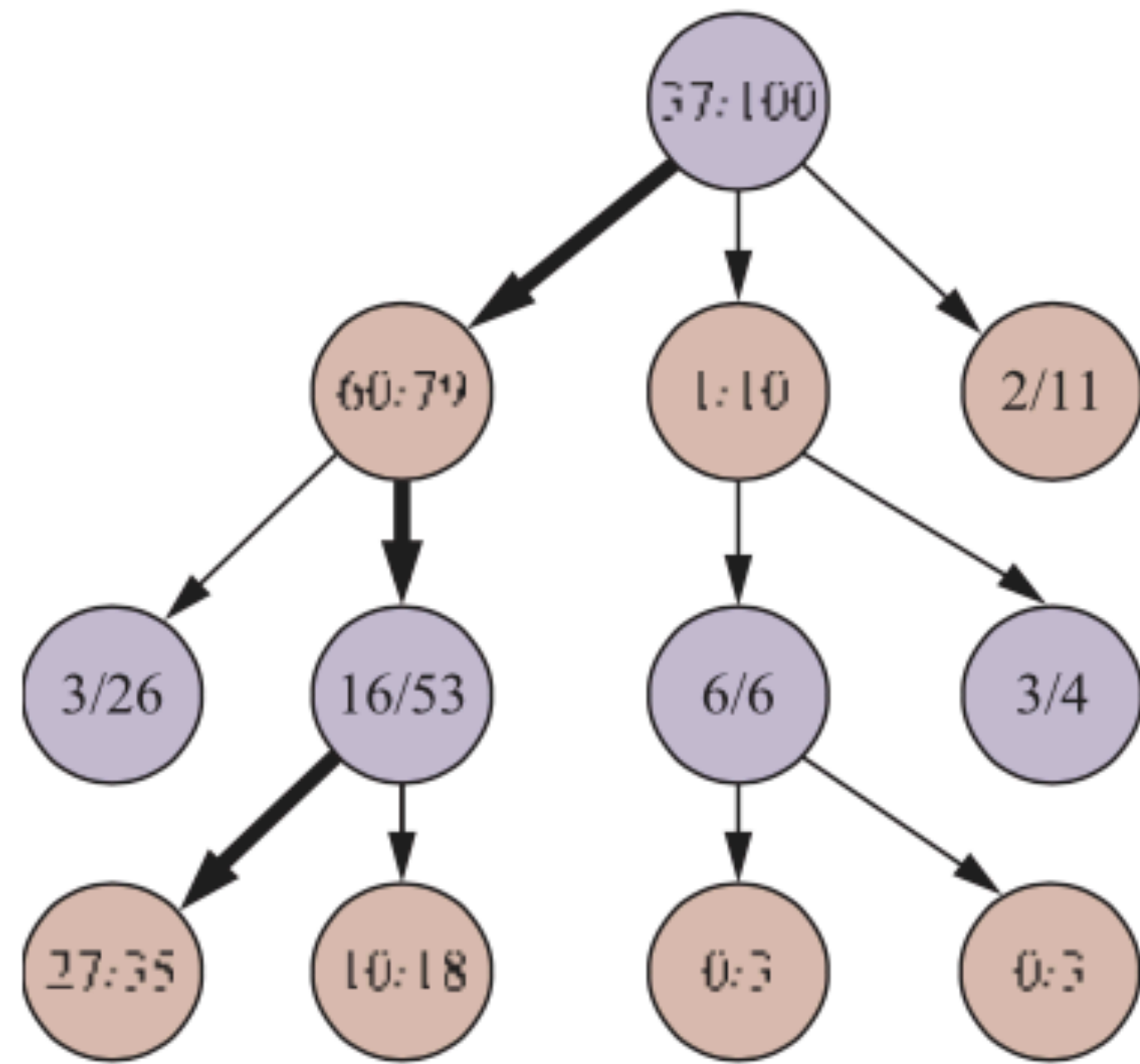
- For every node on the backpropagation path certain **statistics** are computed/updated

## 61 Node's Statistics

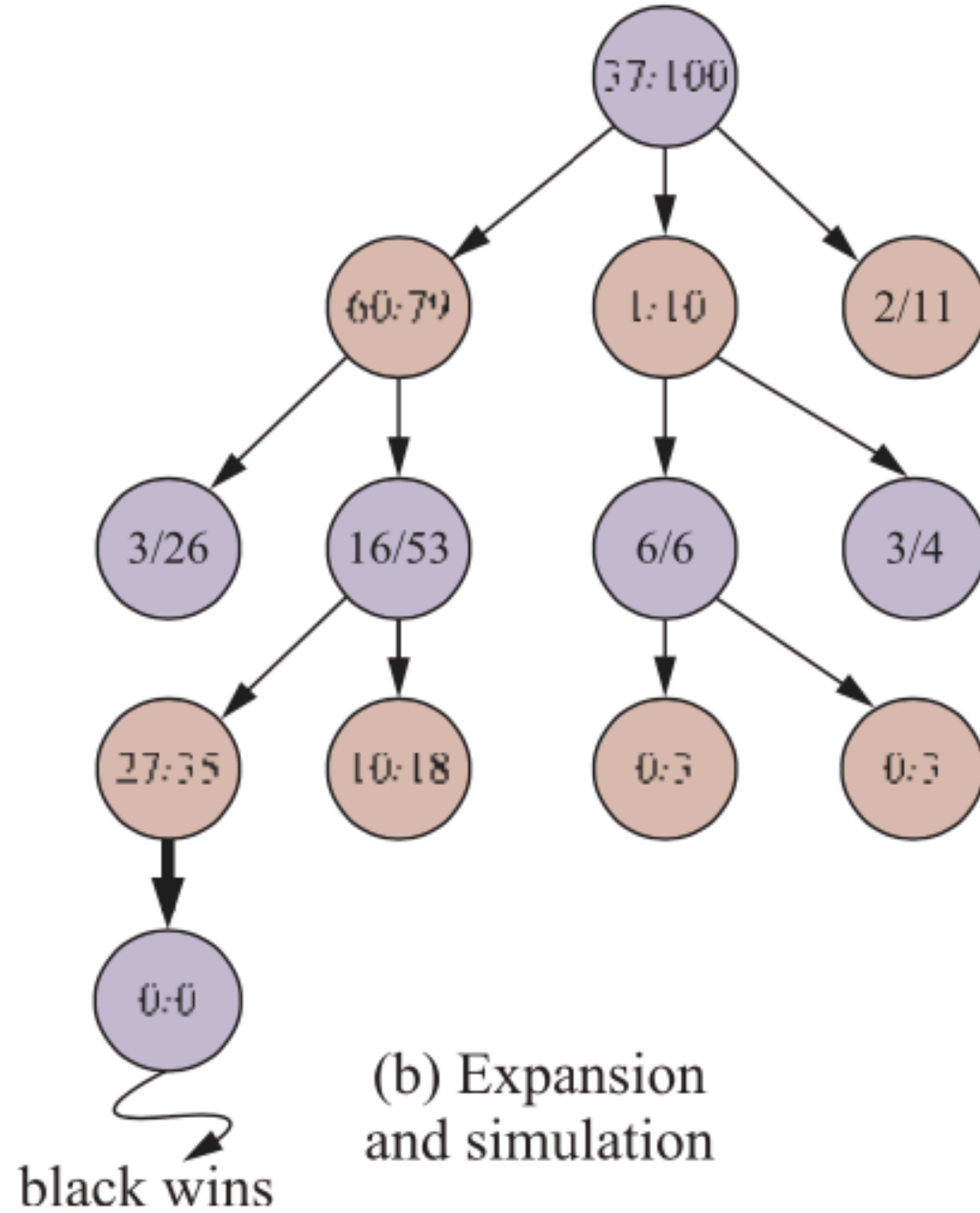
- Back-propagating updates the **total simulation reward**  $Q(v)$  and total number of visits  $N(v)$  for all nodes  $v$  on backpropagation path:
  - $Q(v)$  – **Total simulation reward**, e.g., sum of simulation results that passed through considered node.
  - $N(v)$  – **Total number of visits**, i.e., how many times a node has been on the backpropagation path
- Nodes with high reward are good candidates to follow (**exploitation**) but those with low amount of visits may be interesting too (because they are not **explored** well)



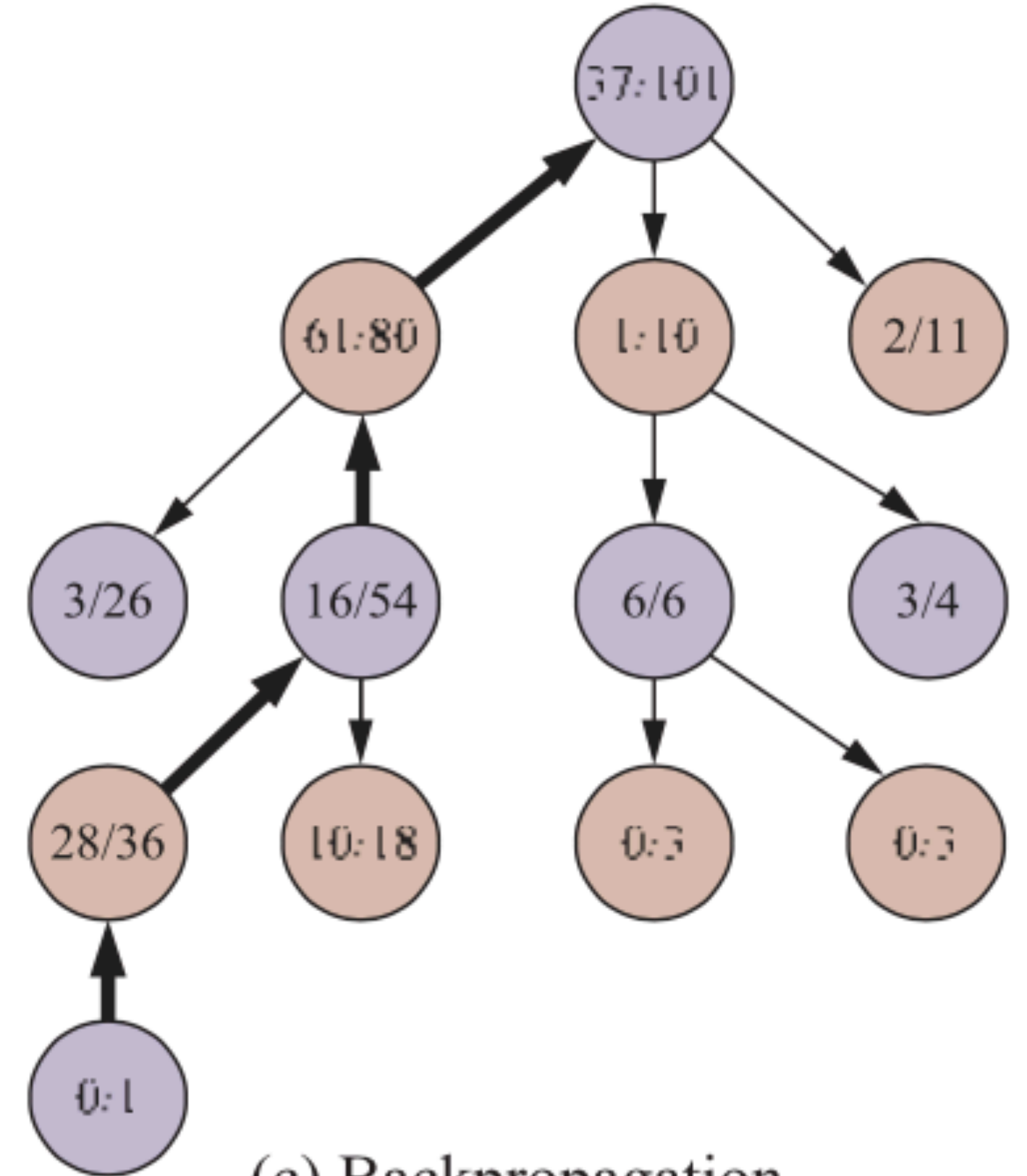
# 62 Example



(a) Selection



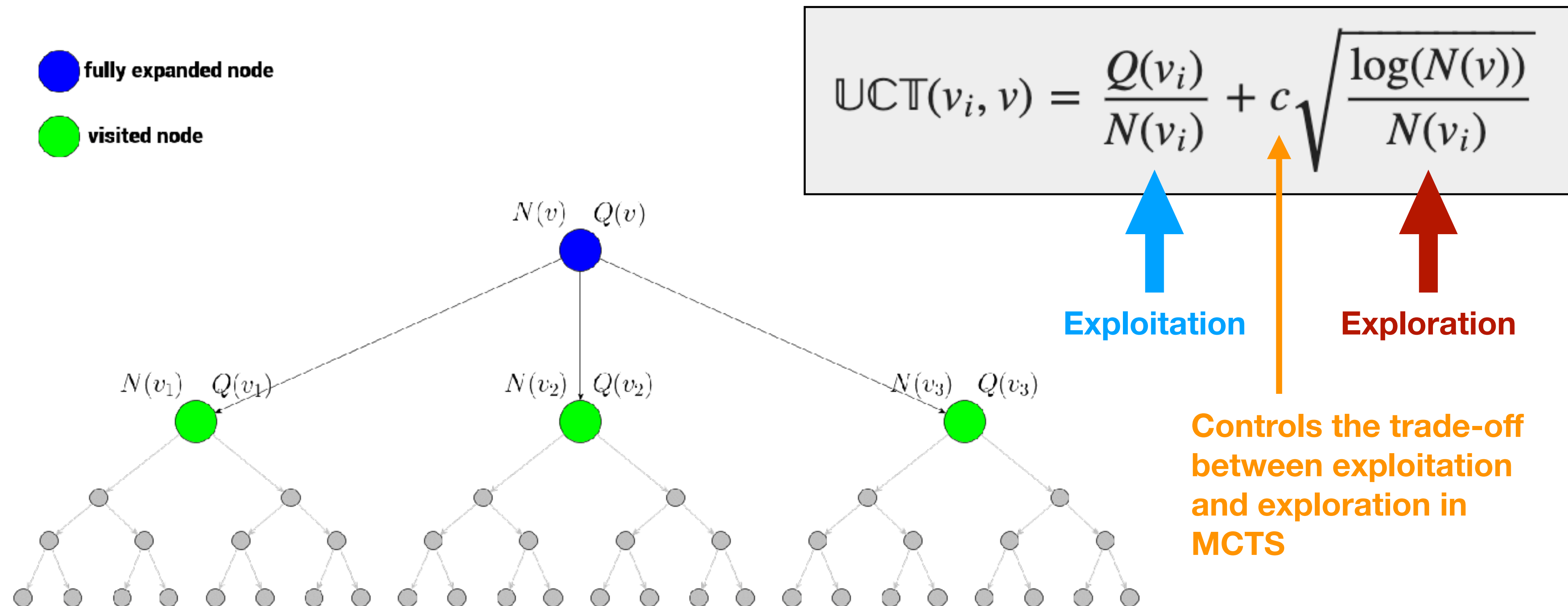
(b) Expansion and simulation



(c) Backpropagation

# 63 Game Tree Traversal: Upper Confidence Bound

- How do we get from a root node to the unvisited node to start a simulation?
- Upper Confidence Bound applied to trees (UCT) is a function that lets us **choose the next node among visited nodes** to traverse through – the core function of Monte Carlo Tree Search



## UCT in Alpha Go and Alpha Zero

In both **Alpha Go Lee** and **Alpha Zero** the tree traversal follows the nodes that maximize the following UCT variant:

$$\text{UCT}(v_i, v) = \frac{Q(v_i)}{N(v_i)} + cP(v, v_i) \frac{\sqrt{N(v)}}{1 + N(v_i)}$$

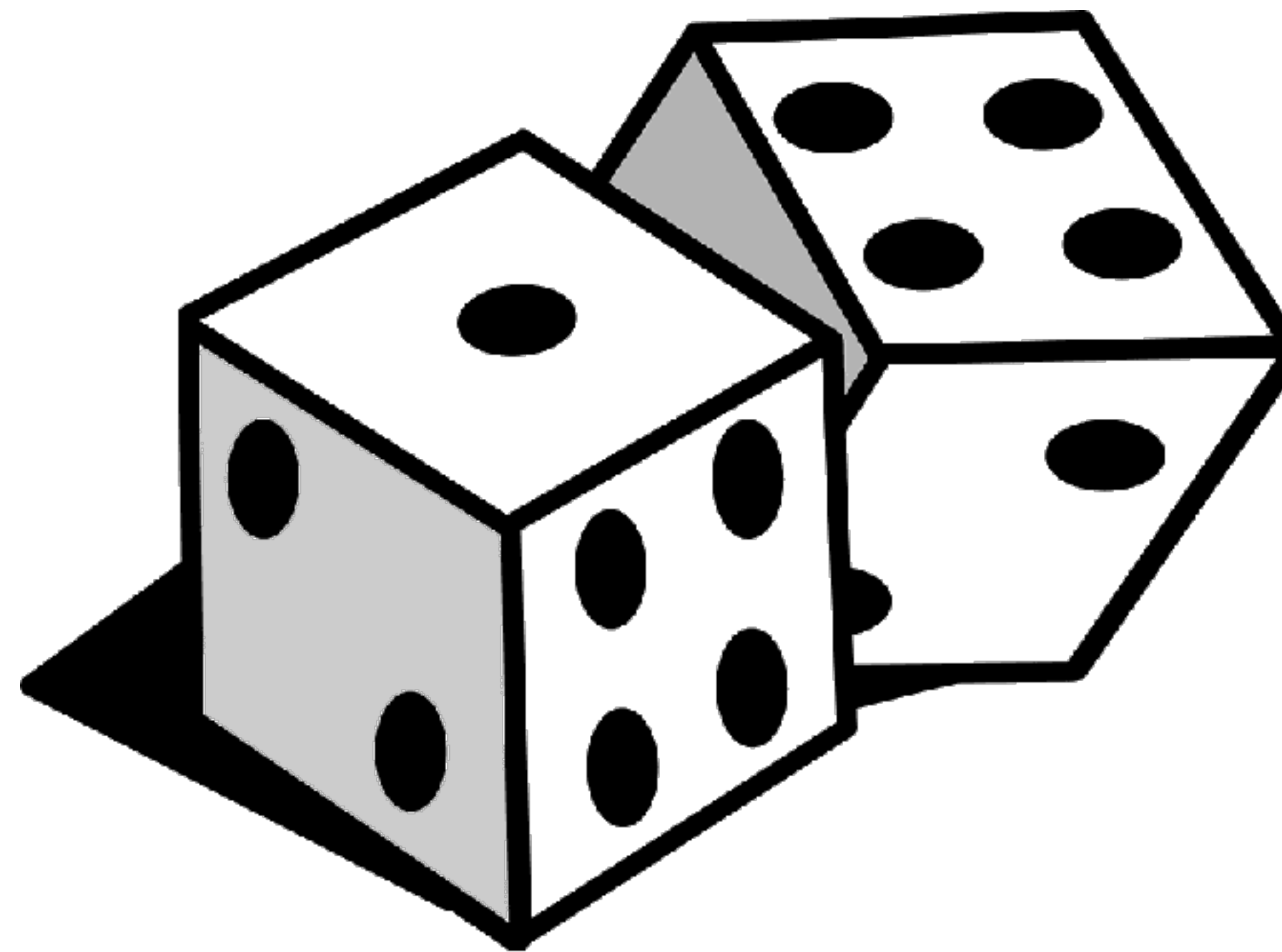
where  $P(v_i, v)$  is prior probability of the move (transition from  $v$  to  $v_i$ ), its value comes from the output of deep neural network called **Policy Network**. Policy Network is a function that consumes game state and produce probability distribution over possible moves (please note this one is different from fast rollout policy). The purpose here is to reduce the search space to the moves that are reasonable – adding it to exploration component directs exploration towards the reasonable moves.



# Terminating Monte Carlo Tree Search

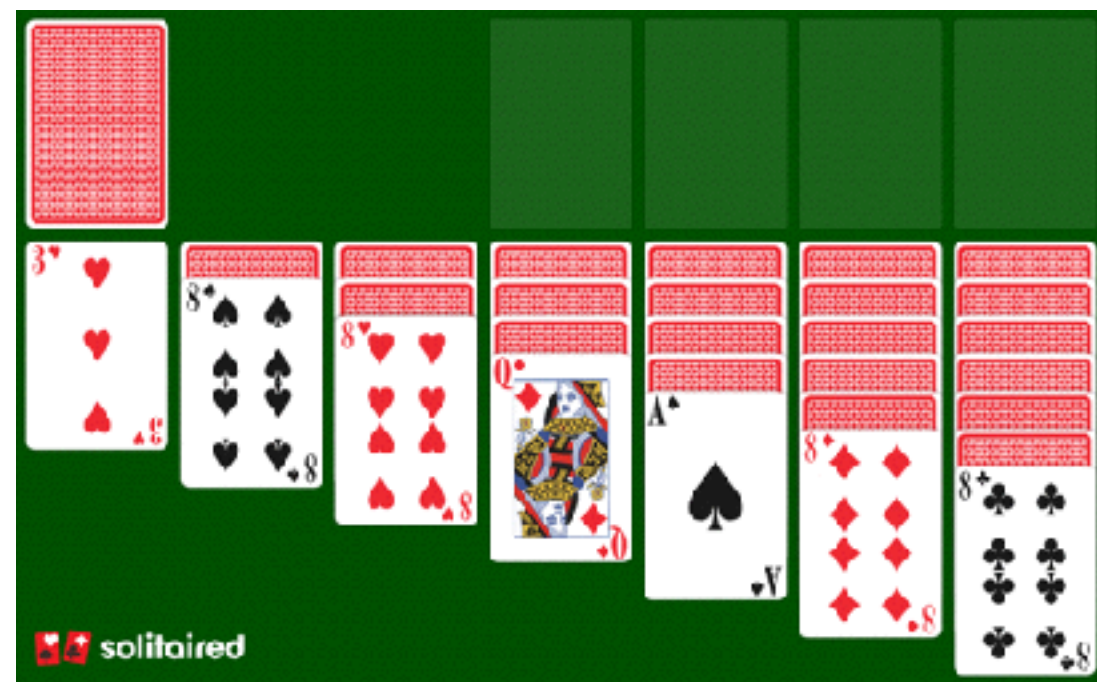
- **When do we actually end the MCTS procedure?**
  - **It depends on the context.** If you build a game engine then your “thinking time” is probably limited, plus your computational capacity has its boundaries, too. Therefore the safest bet is to run MCTS routine as long as your resources let you.
- The general idea of simulating moves into the future, observing the outcome, and using the outcome to determine which moves are good ones is one kind of **reinforcement learning** (we will cover in the future lectures).

# Search with Uncertainty

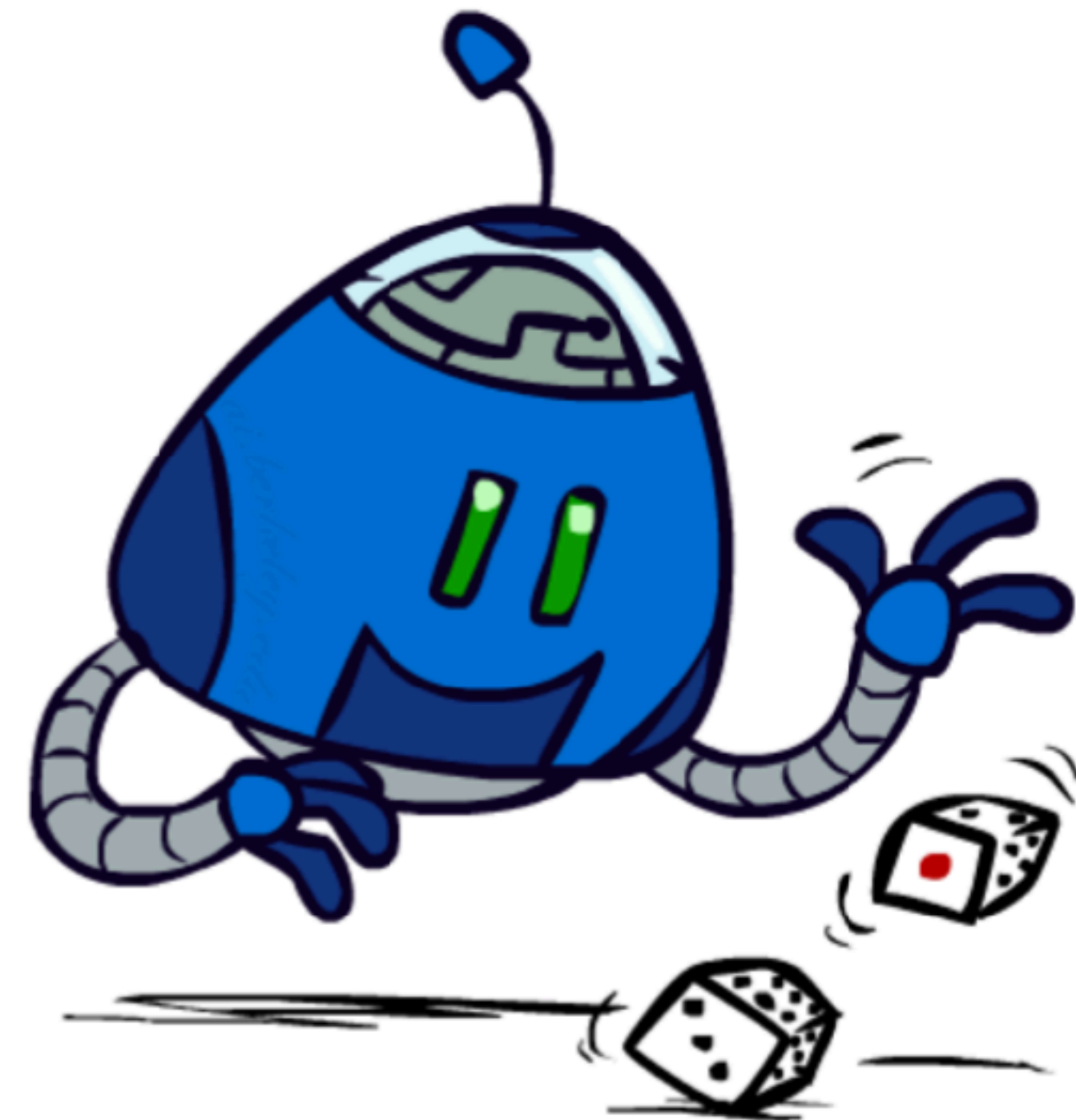
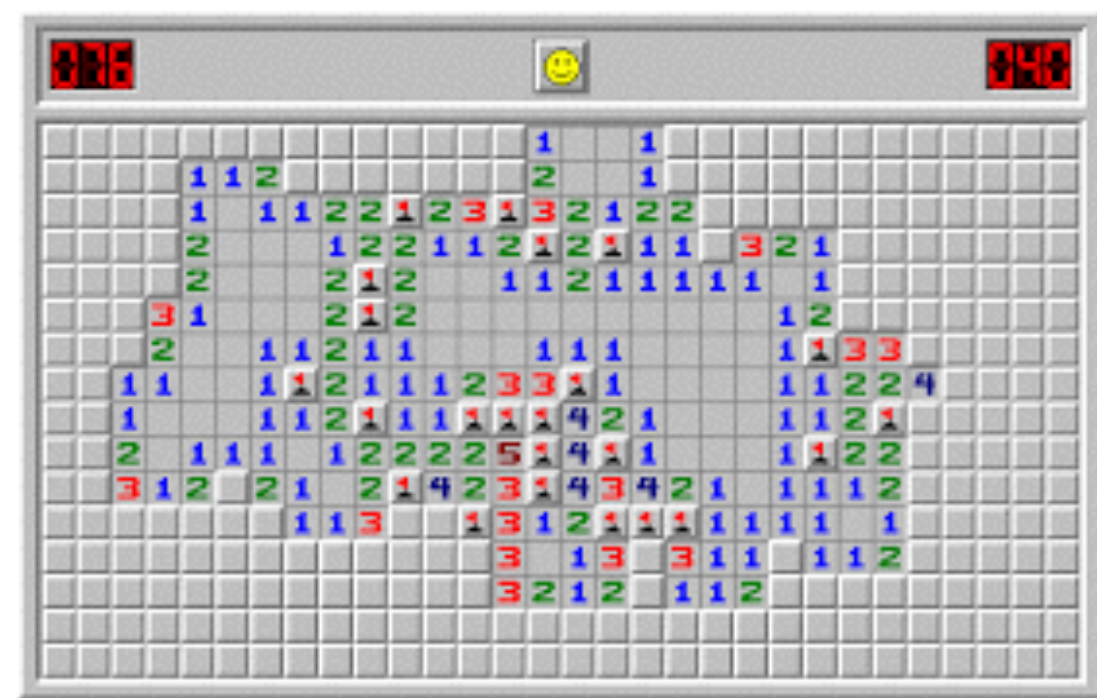


# Stochastic Games

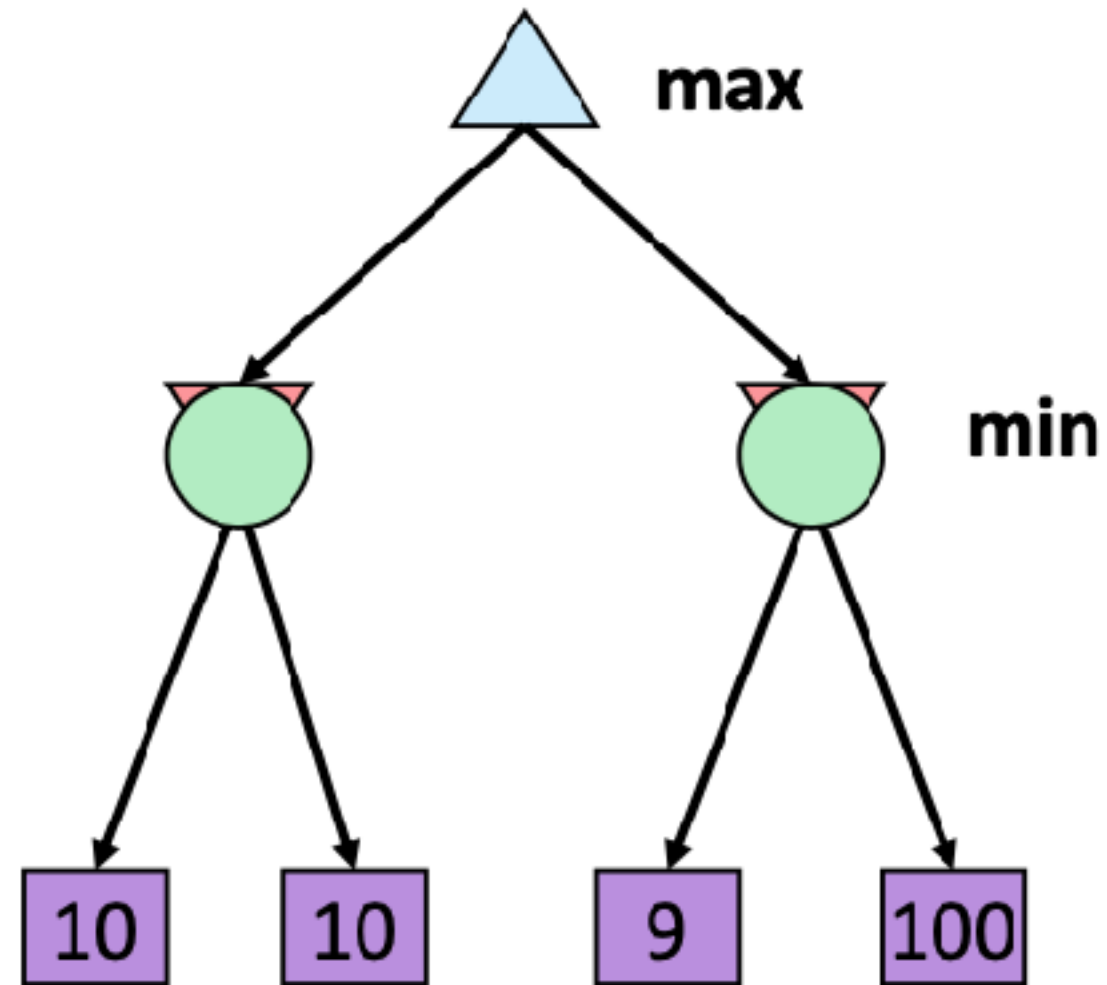
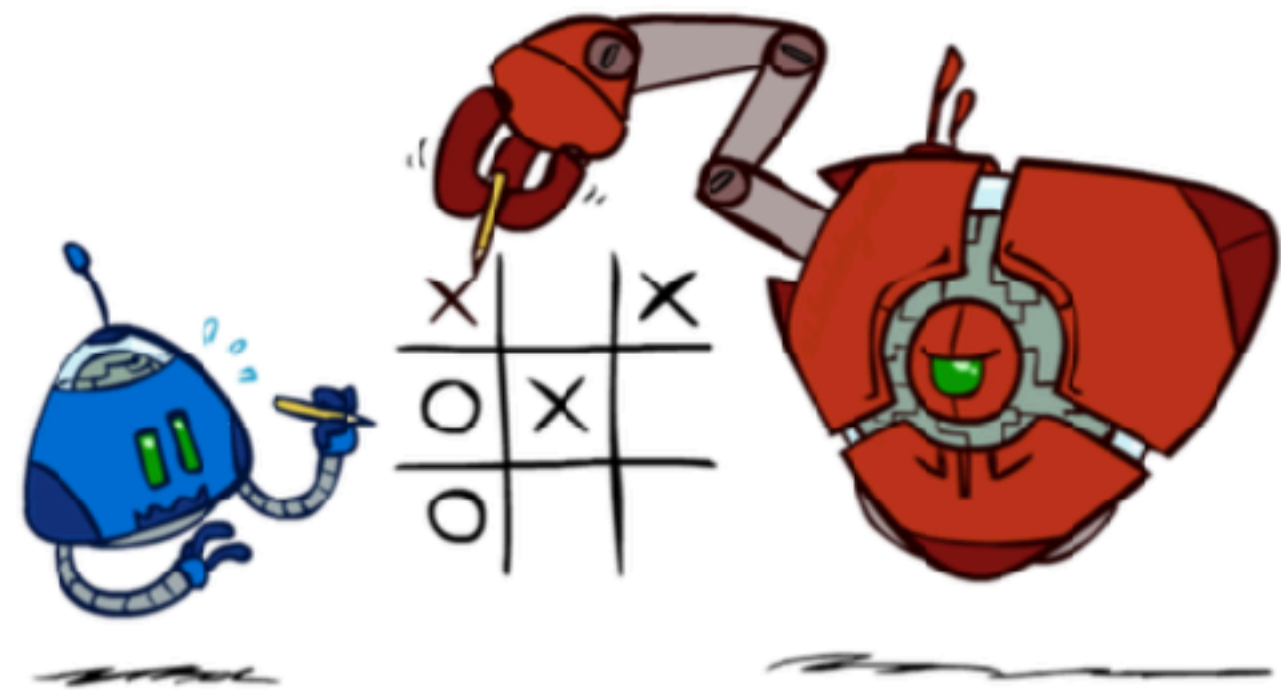
- What if we don't know what the result of an action will be? E.g.,
  - In solitaire, shuffle is unknown



- In minesweeper, mine locations



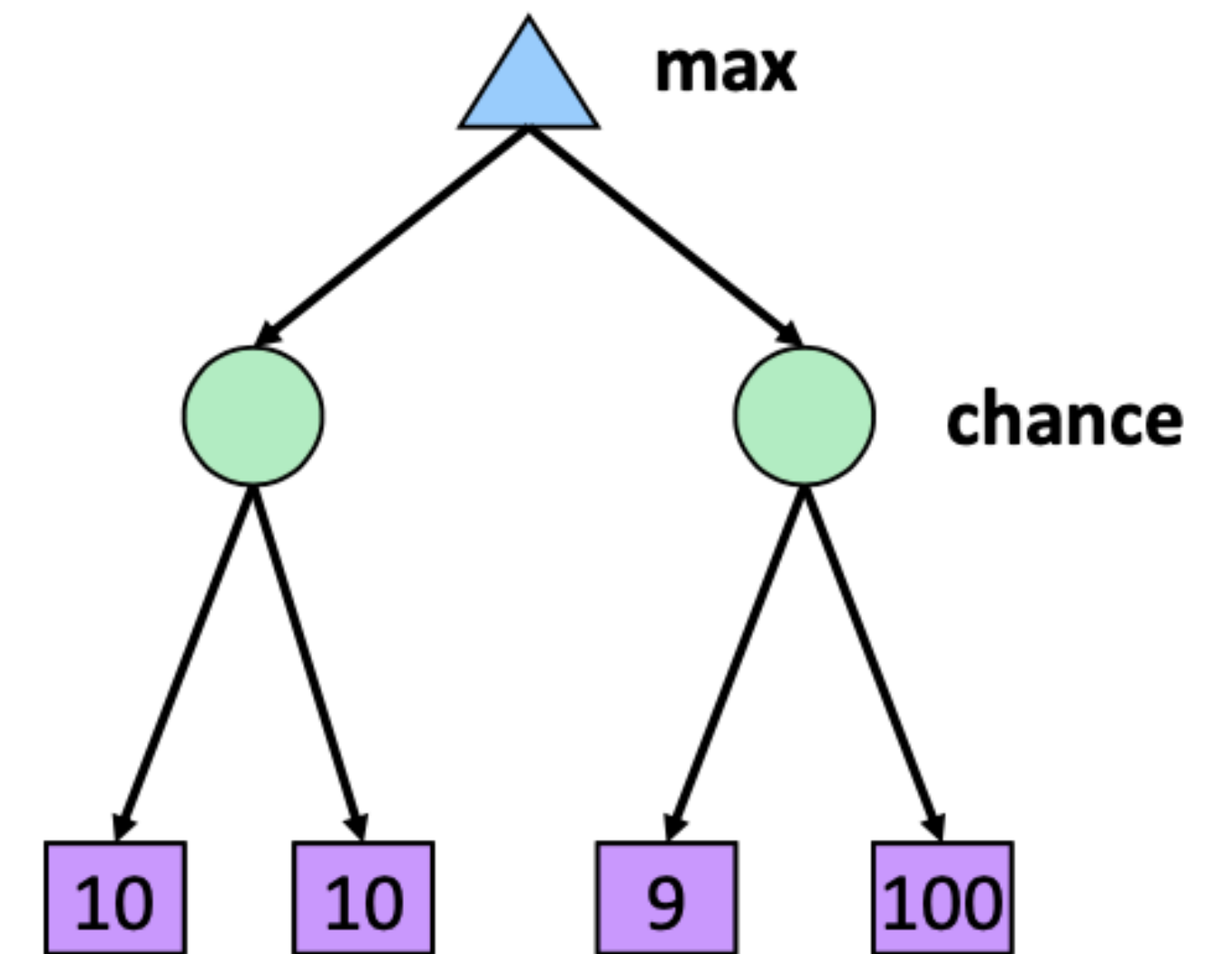
# Worst Case vs. Average Case



Idea: Uncertain outcomes controlled by chance, not an adversary!

# Expectimax Search

- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes
- **Expectimax search:** compute the **average score** under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - I.e. take weighted average (expectation) of children
- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes



# Reminder: Probabilities

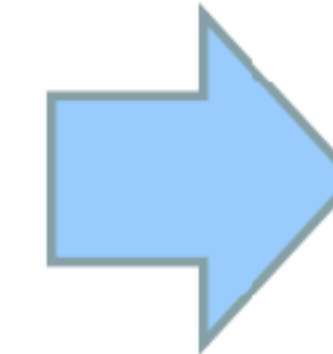
- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes
- Example: Traffic on freeway
  - Random variable:  $T$  = whether there's traffic
  - Outcomes:  $T$  in {none, light, heavy}
  - Distribution:  $P(T=none) = 0.25$ ,  $P(T=light) = 0.50$ ,  $P(T=heavy) = 0.25$
- Some laws of probability (more later):
  - Probabilities are always non-negative
  - Probabilities over all possible outcomes sum to one
- As we get more evidence, probabilities may change:
  - $P(T=heavy) = 0.25$ ,  $P(T=heavy | Hour=8am) = 0.60$



# 71 Reminder: Expectations

- The expected value of a function of a random variable is the average, weighted by the probability distribution over outcomes
- Example: How long to get to the airport?

Time:	20 min		30 min		60 min			
	x	+	x	+	x			
Probability:	0.25		0.50		0.25			35 min



# Expectimax Pseudocode

```
def value(state):
```

```
    if the state is a terminal state: return the state's utility
```

```
    if the next agent is MAX: return max-value(state)
```

```
    if the next agent is EXP: return exp-value(state)
```

```
def max-value(state):
```

```
    initialize v =  $-\infty$ 
```

```
    for each successor of state:
```

```
        v = max(v, value(successor))
```

```
    return v
```

```
def exp-value(state):
```

```
    initialize v = 0
```

```
    for each successor of state:
```

```
        p = probability(successor)
```

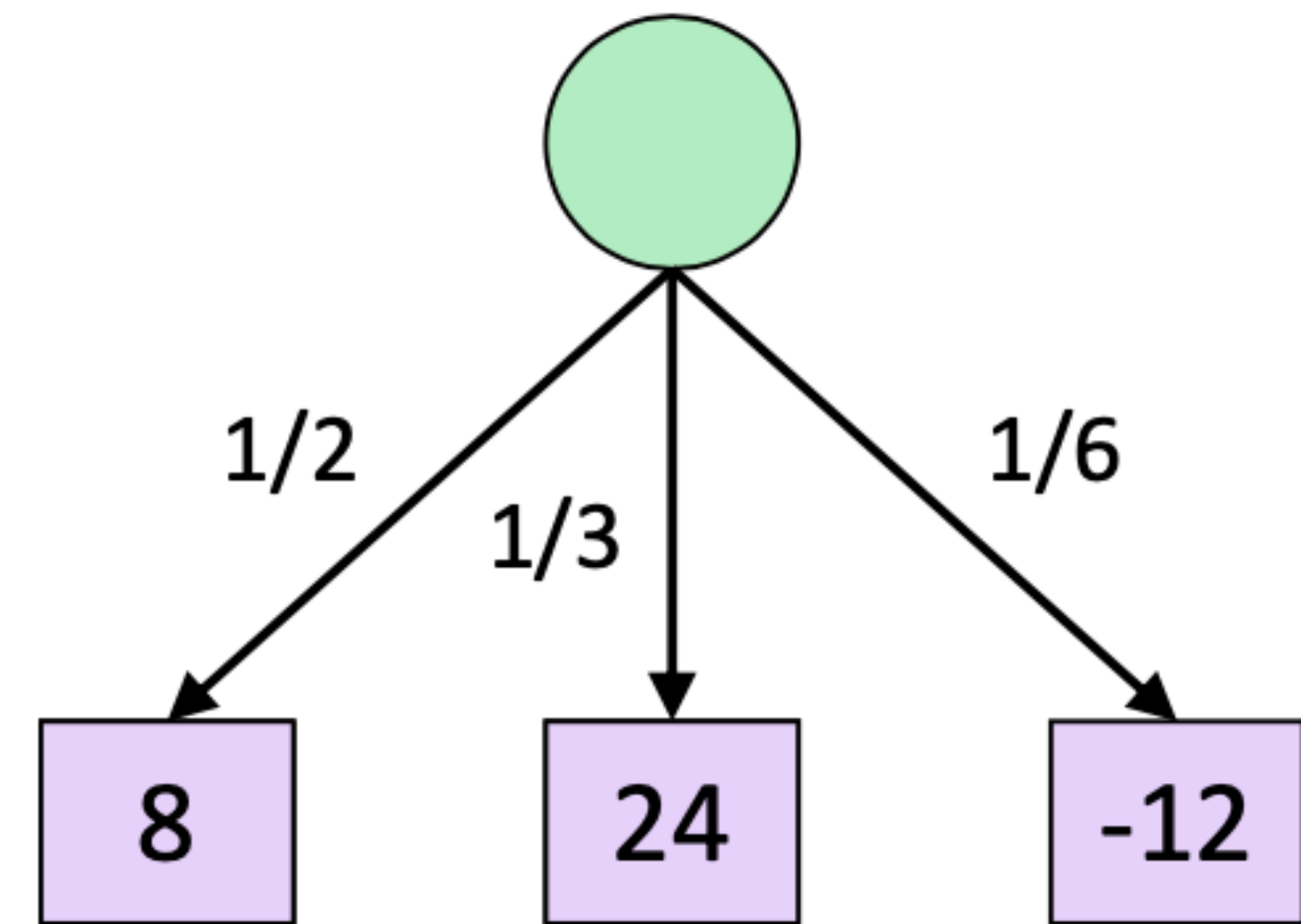
```
        v += p * value(successor)
```

```
    return v
```



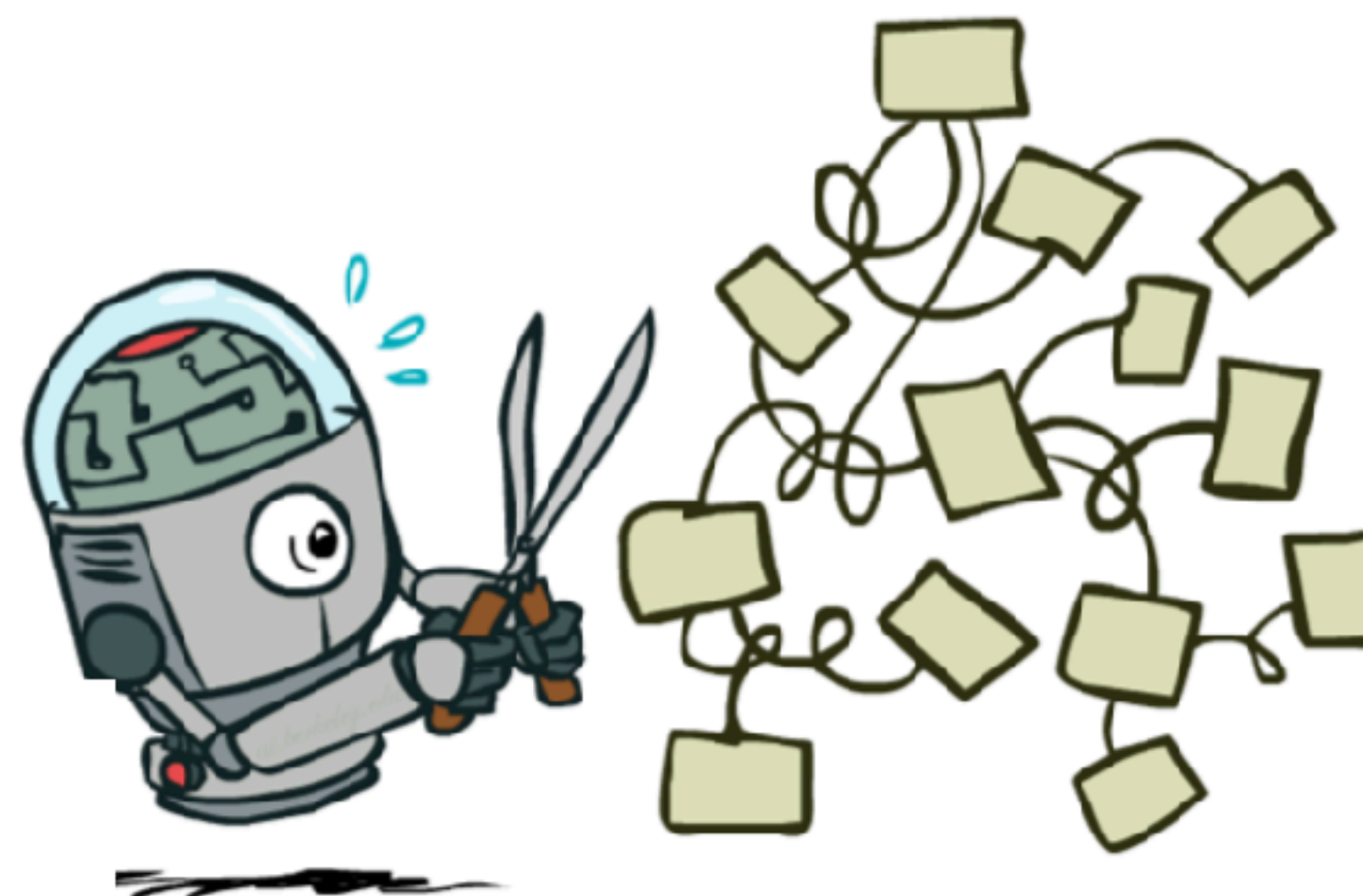
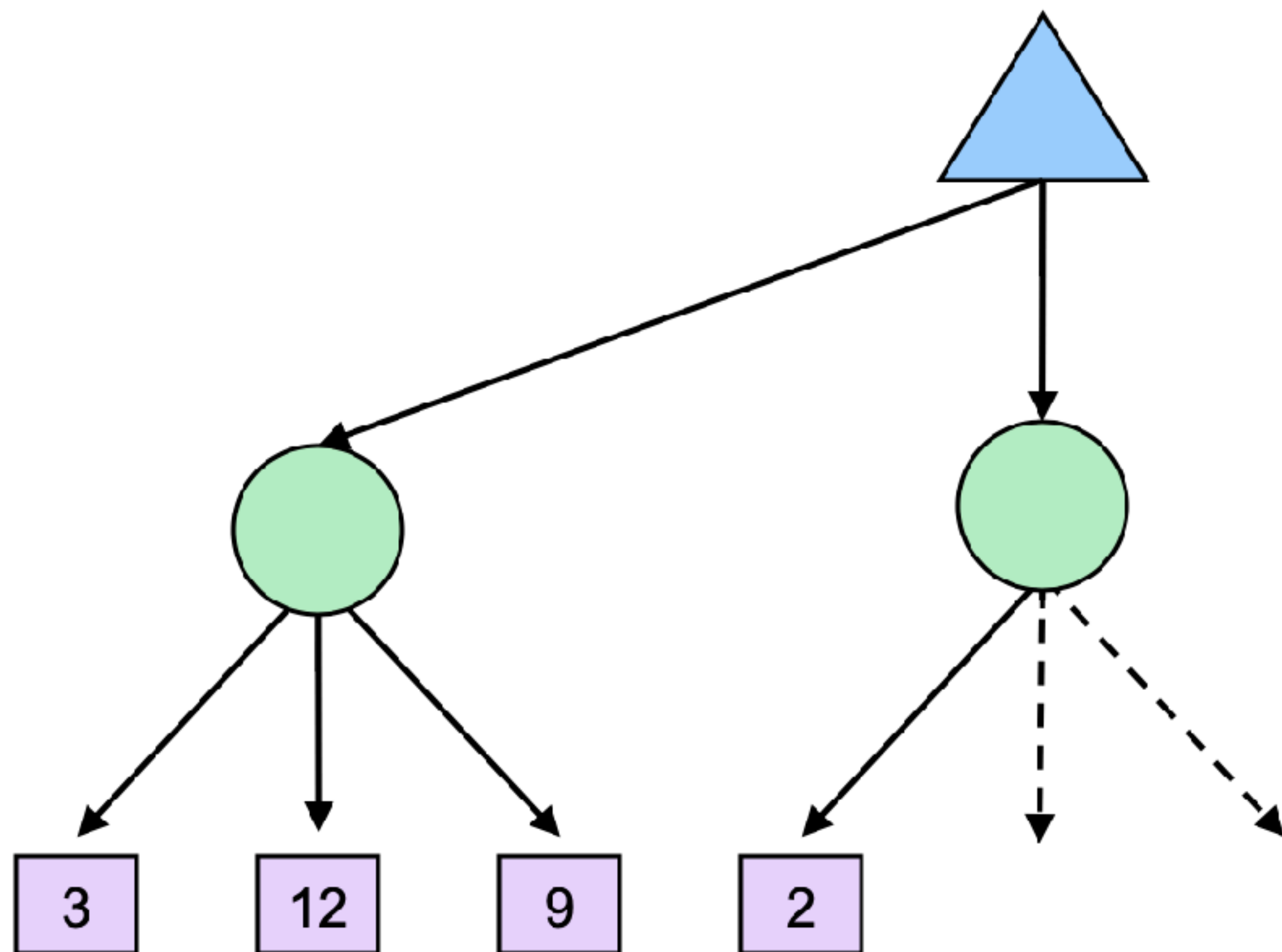
## 73 Expectimax Pseudocode

```
def exp-value(state):  
    initialize v = 0  
    for each successor of state:  
        p = probability(successor)  
        v += p * value(successor)  
    return v
```

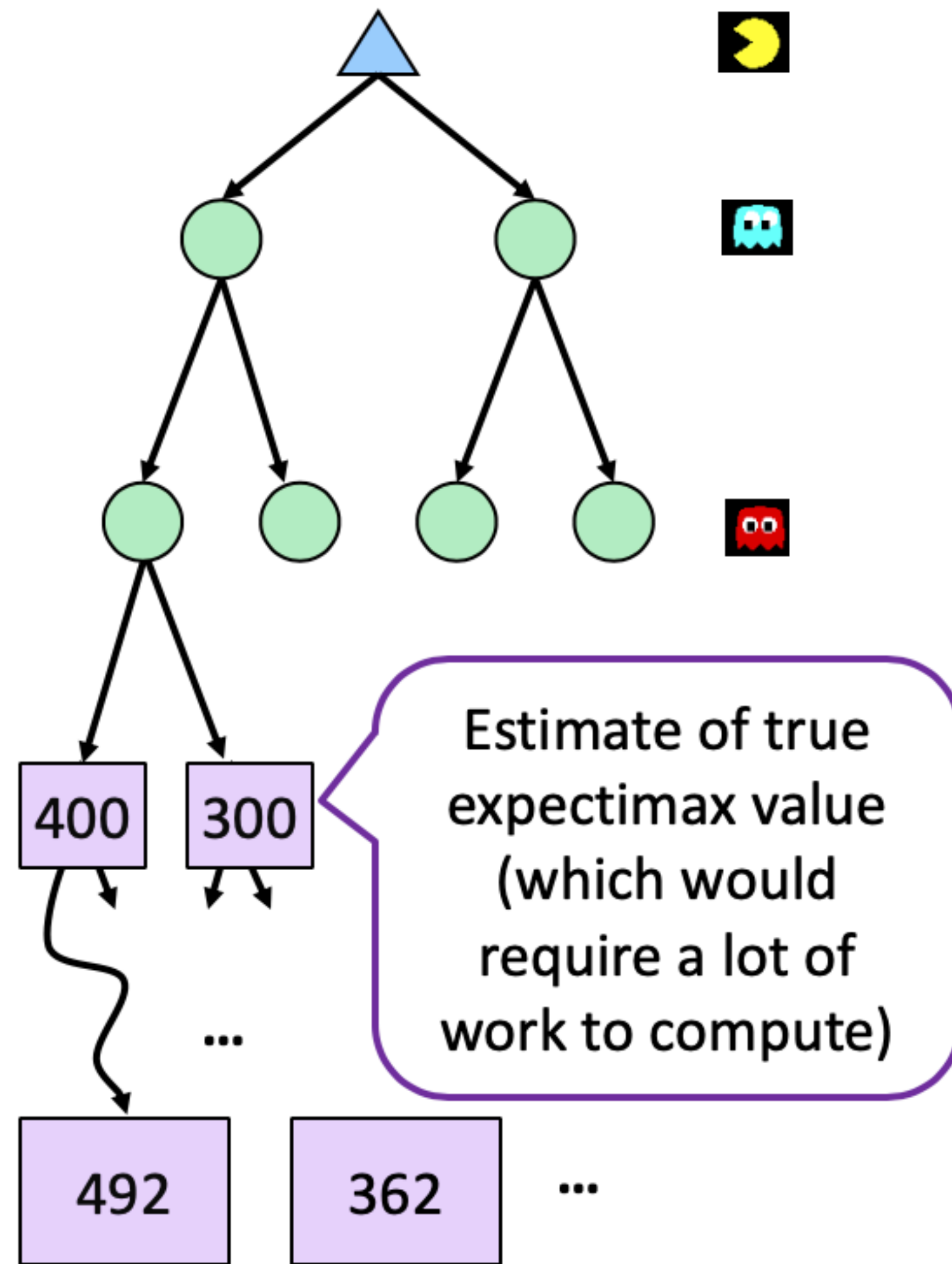


$$v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10$$

# 74 Expectimax Pruning?

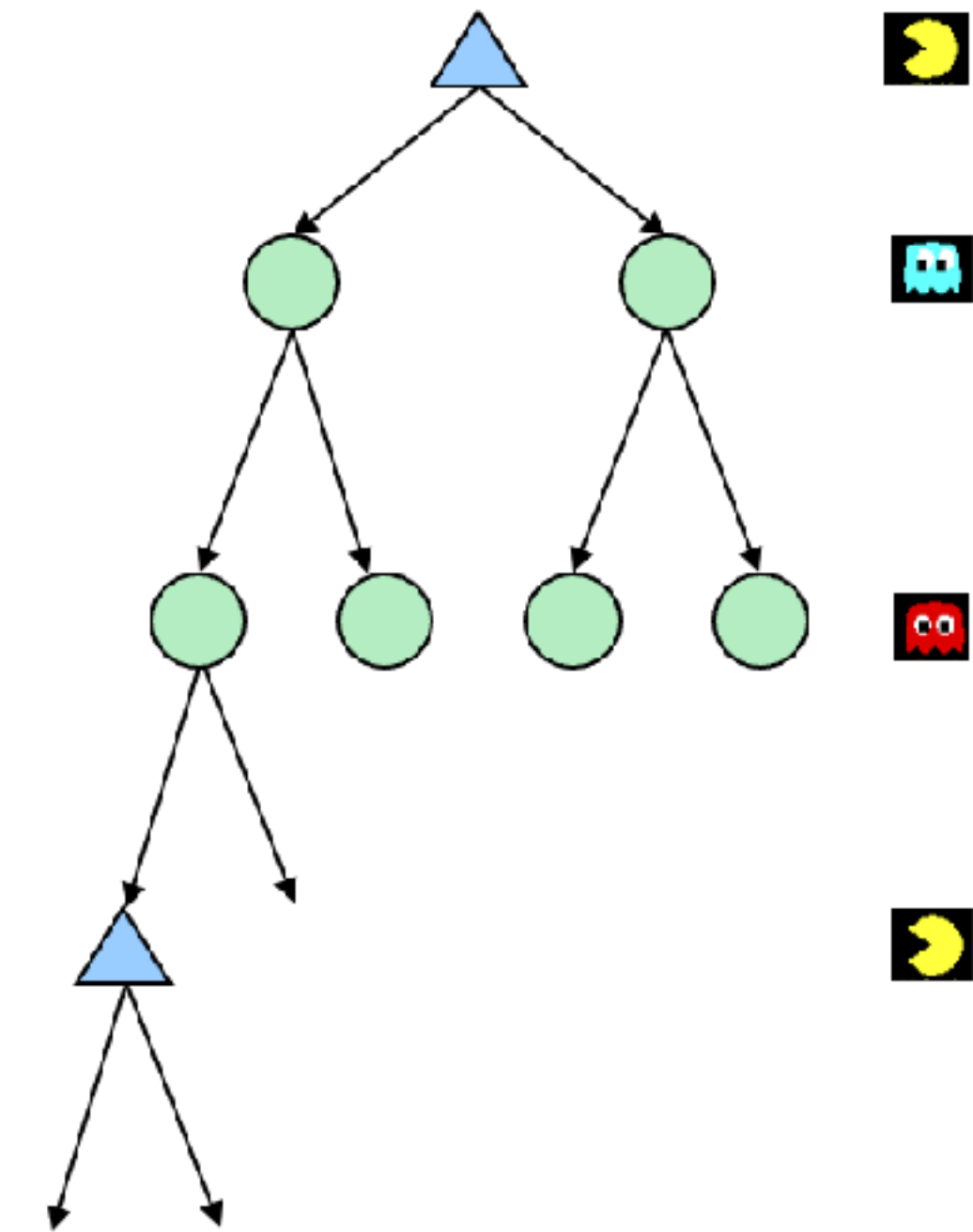


# 75 Depth-Limited Expectimax



# Expectimax Search

- ◎ In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!
  
- ◎ For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



*Having a probabilistic belief about another agent's action does not mean that the agent is flipping any coins!*

# 77 The Dangers of Optimism and Pessimism

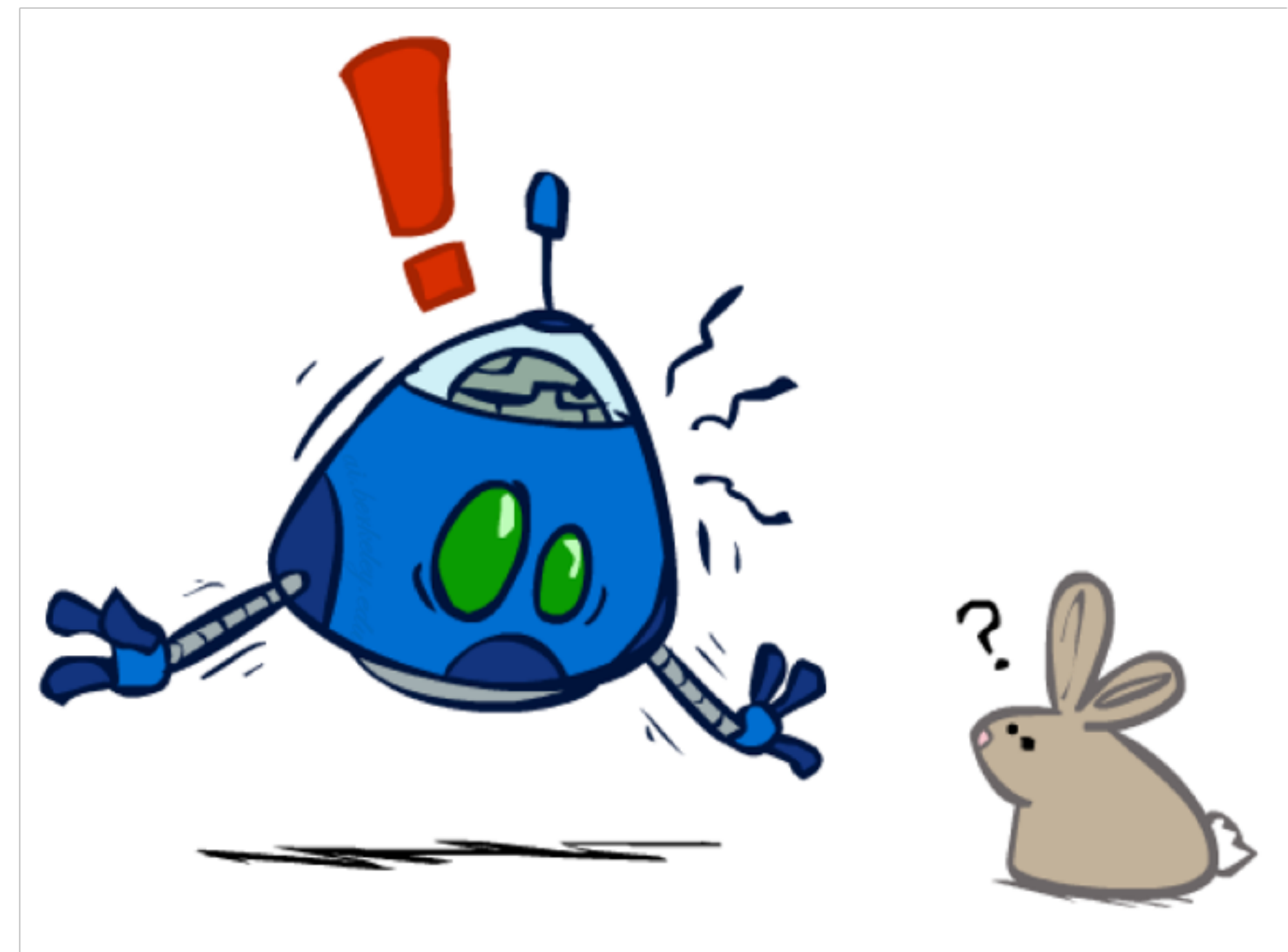
## Dangerous Optimism

Assuming chance when the world is adversarial



## Dangerous Pessimism

Assuming the worst case when it's not likely



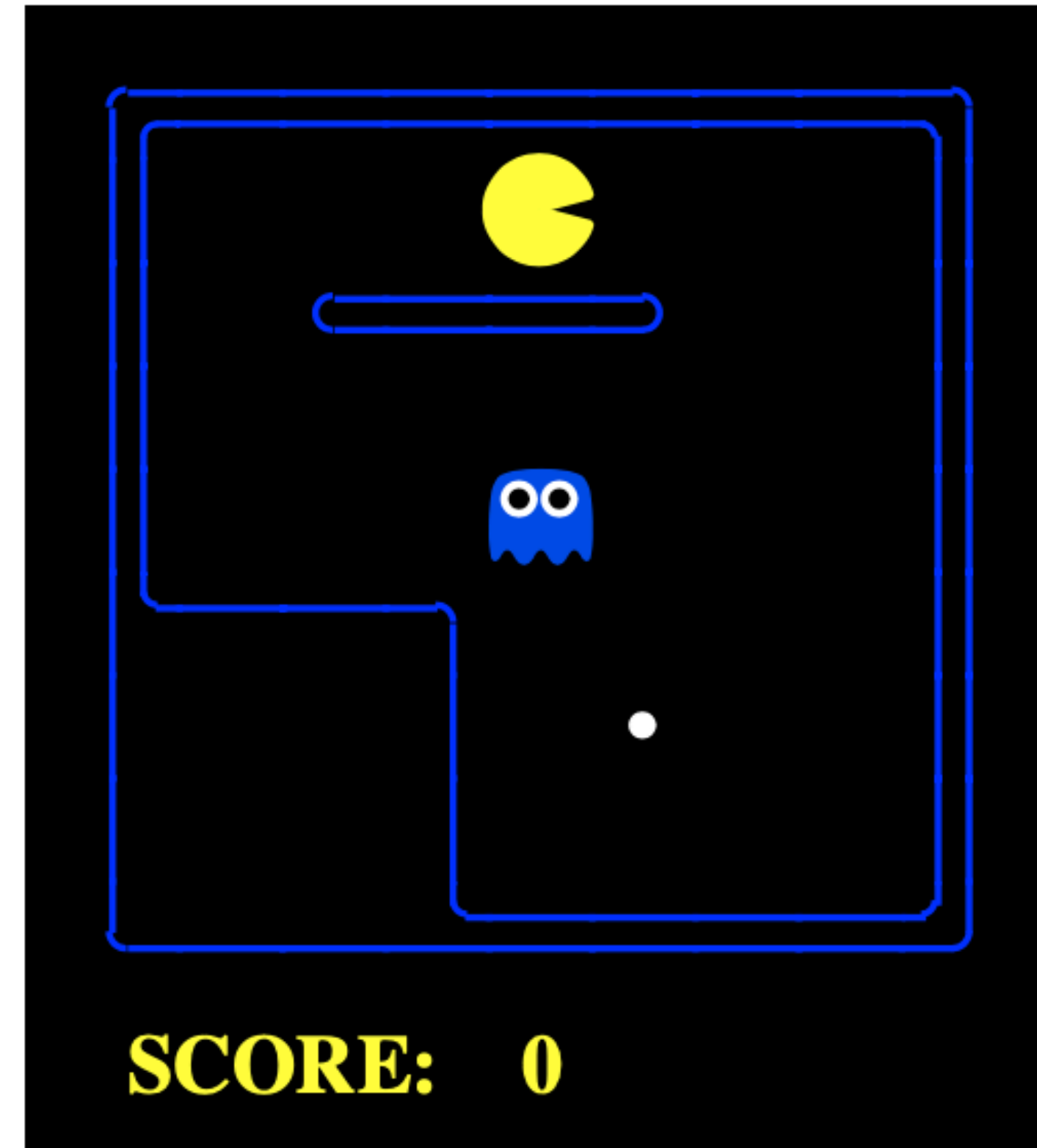
## 78 Stochastic Single-Player: Pacman

- Notice that we've gotten away from thinking that the ghosts are trying to minimize pacman's score
- Instead, they are now a part of the environment
- Pacman has a belief (distribution) over how they will act
- **Quiz:** Can we see minimax as a special case of expectimax?
- **Quiz:** what would pacman's computation look like if we assumed that the ghosts were doing 1-ply minimax and taking the result 80% of the time, otherwise moving randomly?

# Expectimax for Pacman

Results from playing 5 games

	Minimizing Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 493	Won 5/5 Avg. Score: 483
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

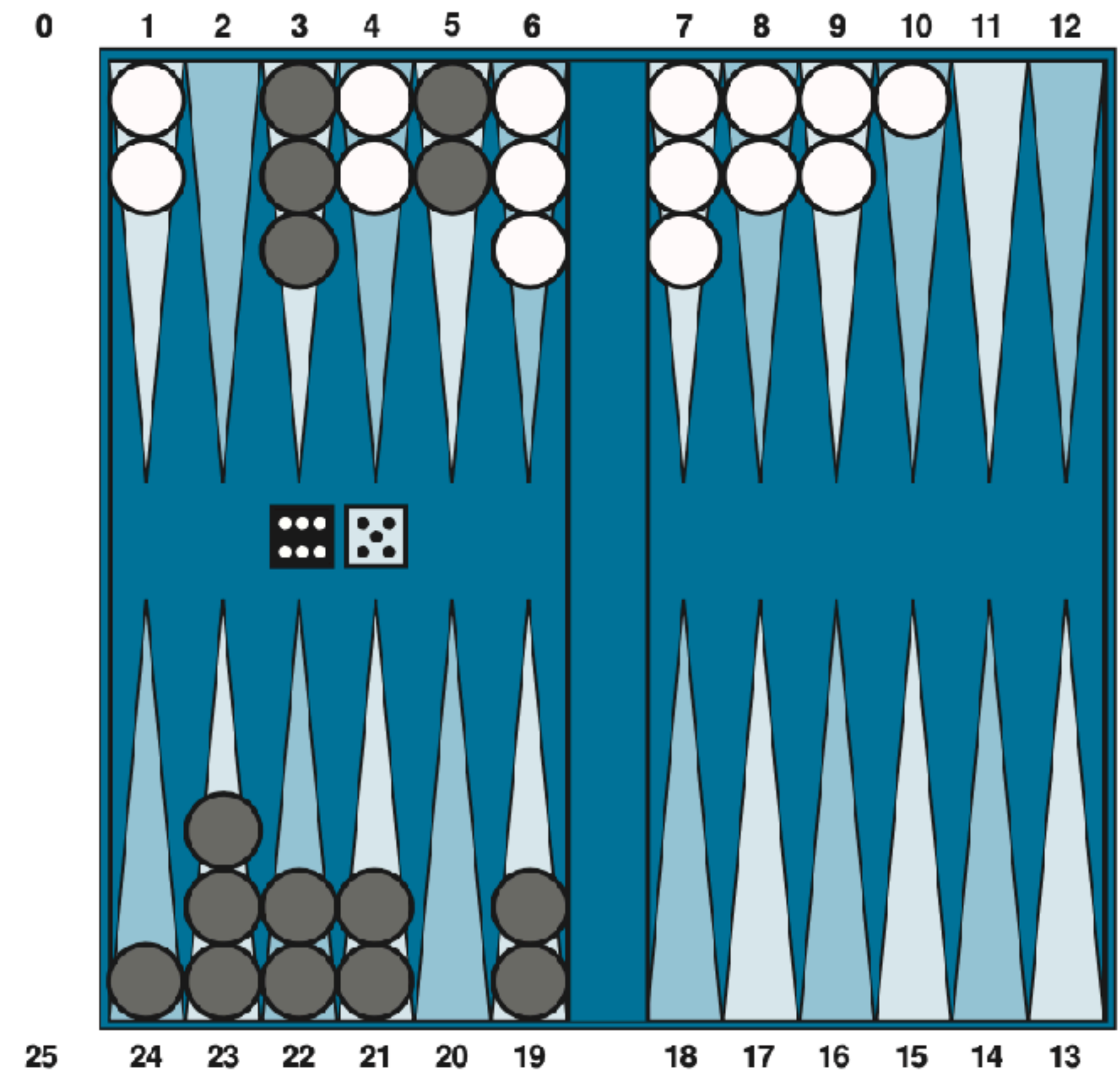


Pacman does depth 4 search with an eval function that avoids trouble  
 Minimizing ghost does depth 2 search with an eval function that seeks Pacman

# Stochastic Two-Player: Backgammon

- ◎ The goal of the game is to **move all one's pieces off the board**.

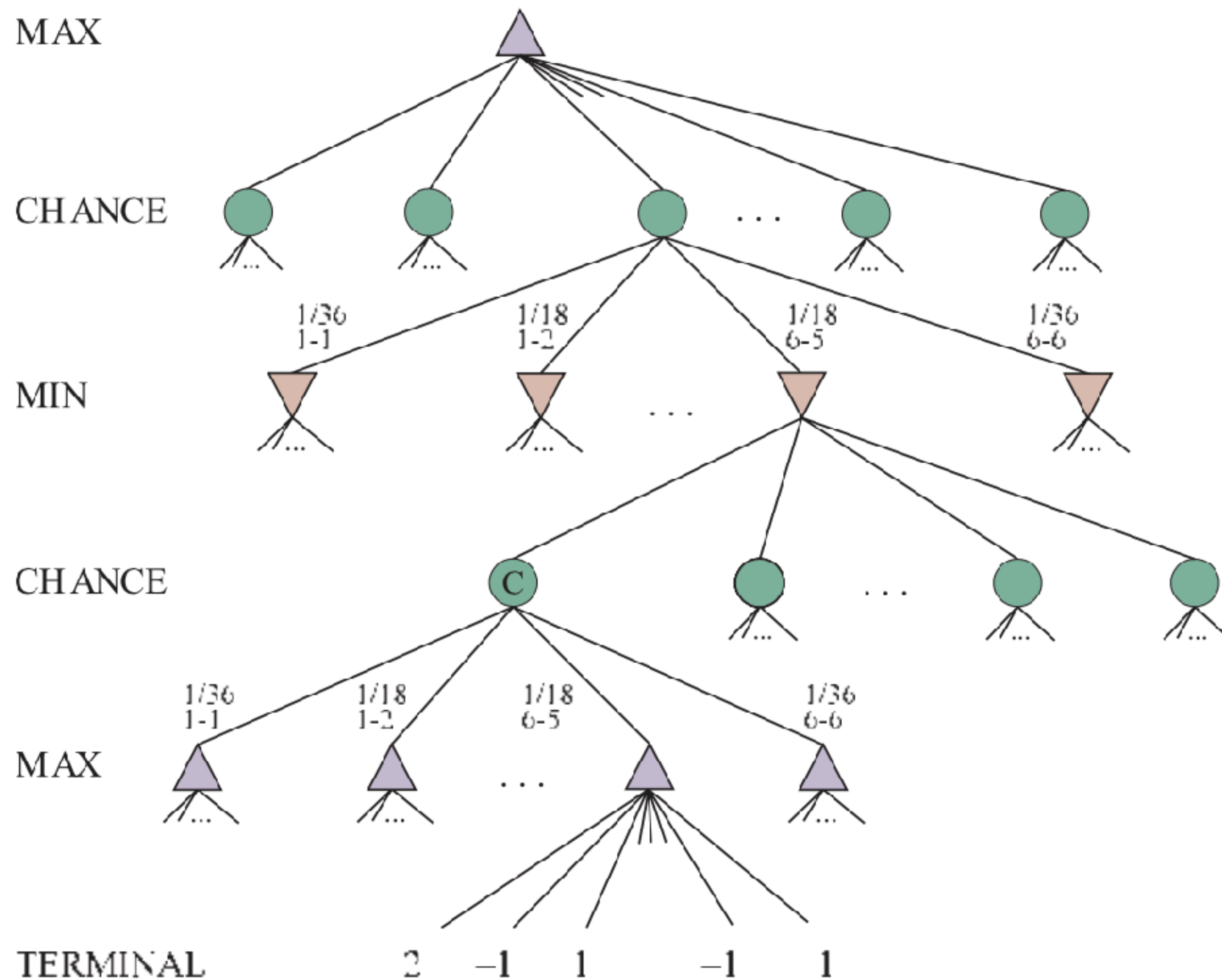
  - Black moves clockwise toward 25, and White moves counterclockwise toward 0.
  - A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over.
  
- ◎ In the position shown, Black has rolled 6–5 and must choose among four legal moves: (5–11,5–10), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.





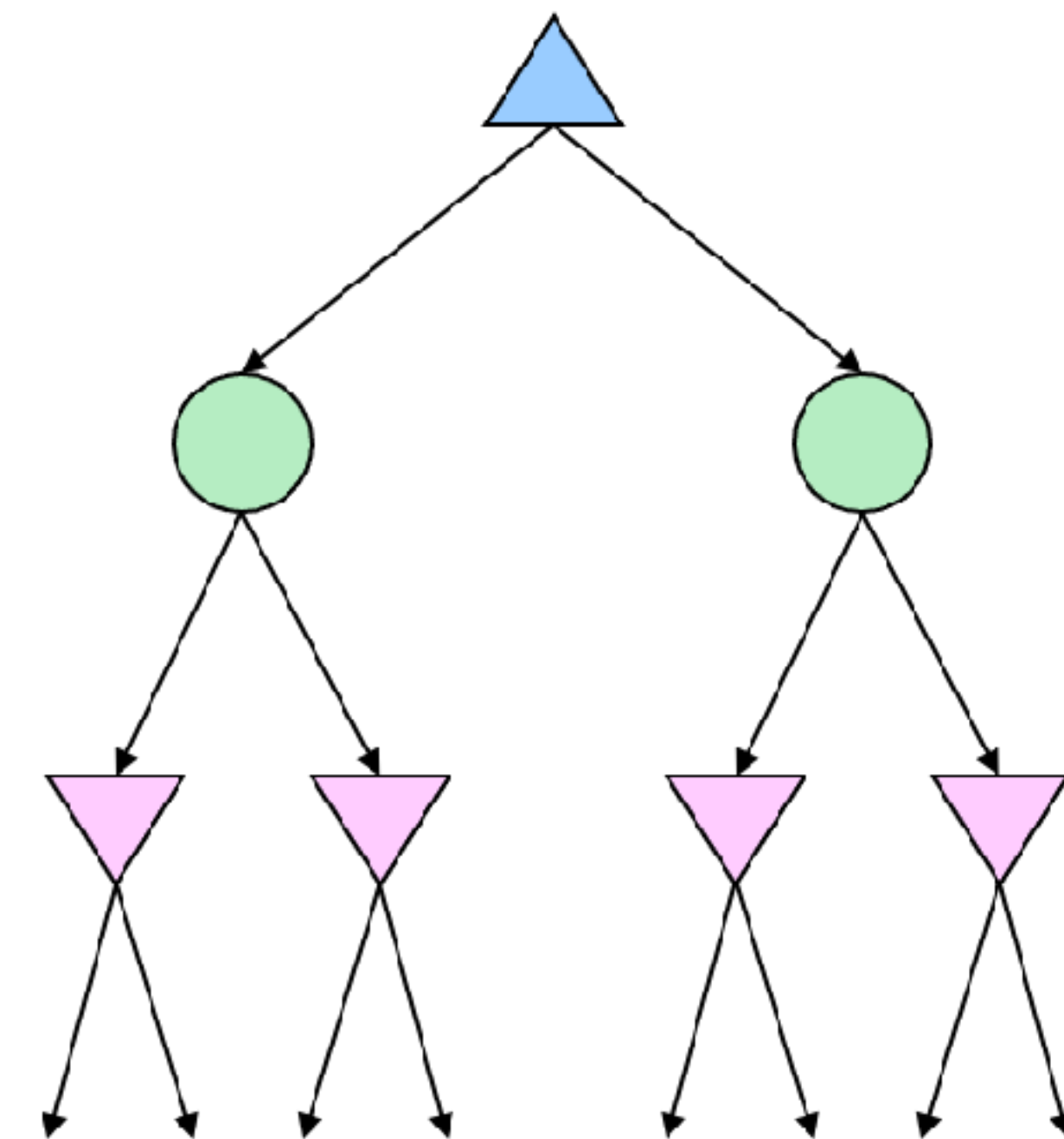
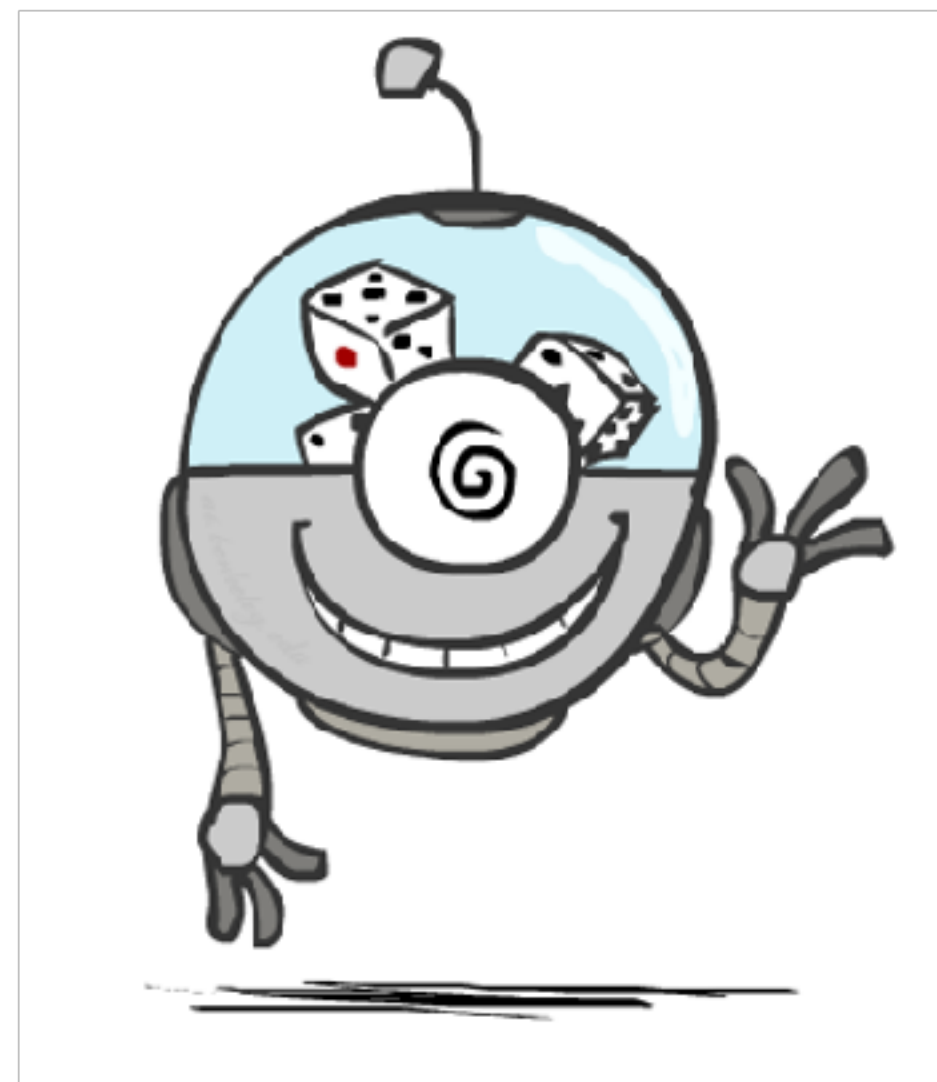
# Example: Backgammon

At this point Black knows what moves can be made, but does not know what White is going to roll and thus does not know what White's legal moves will be. That means Black cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe. A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure



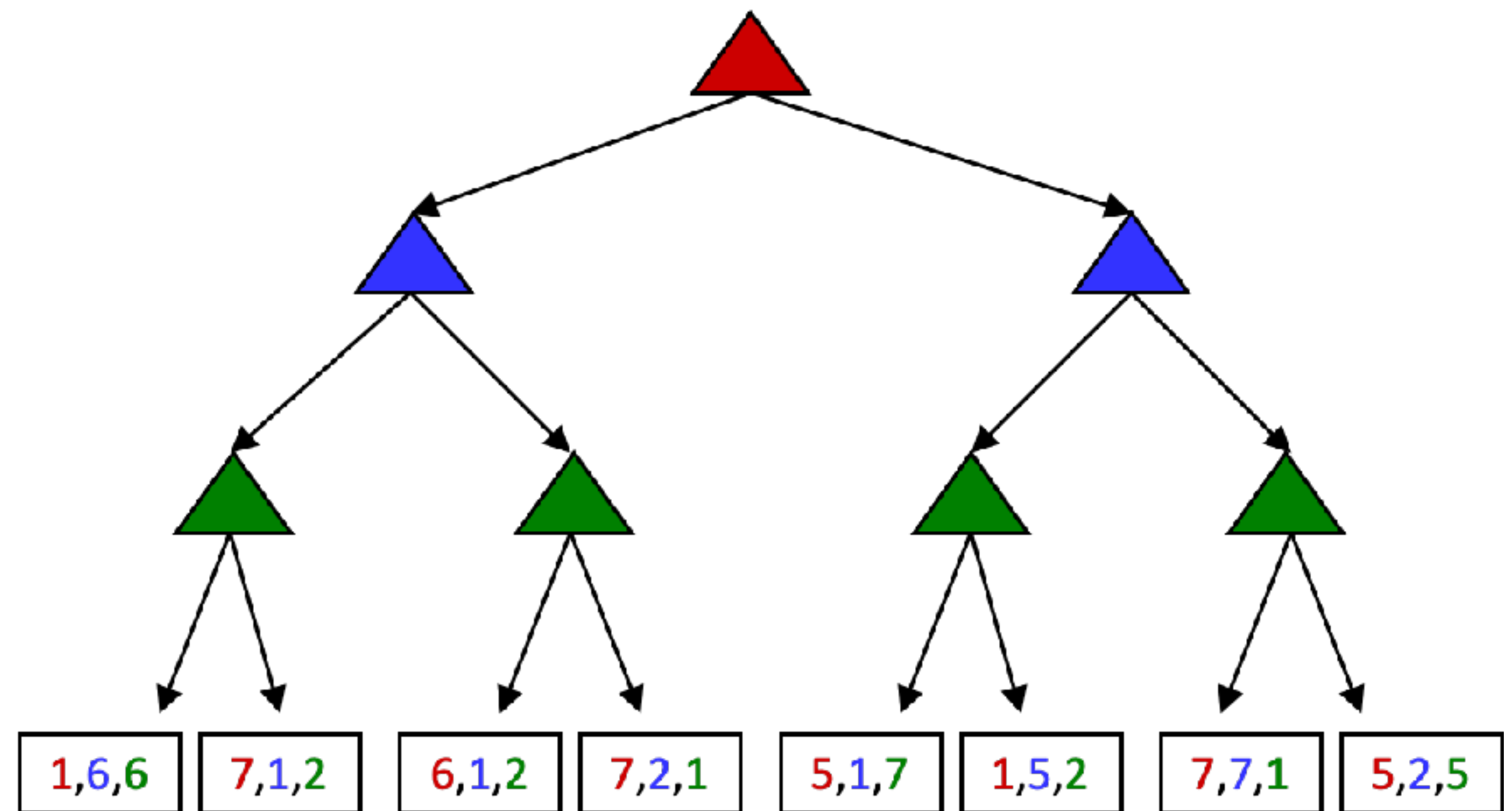
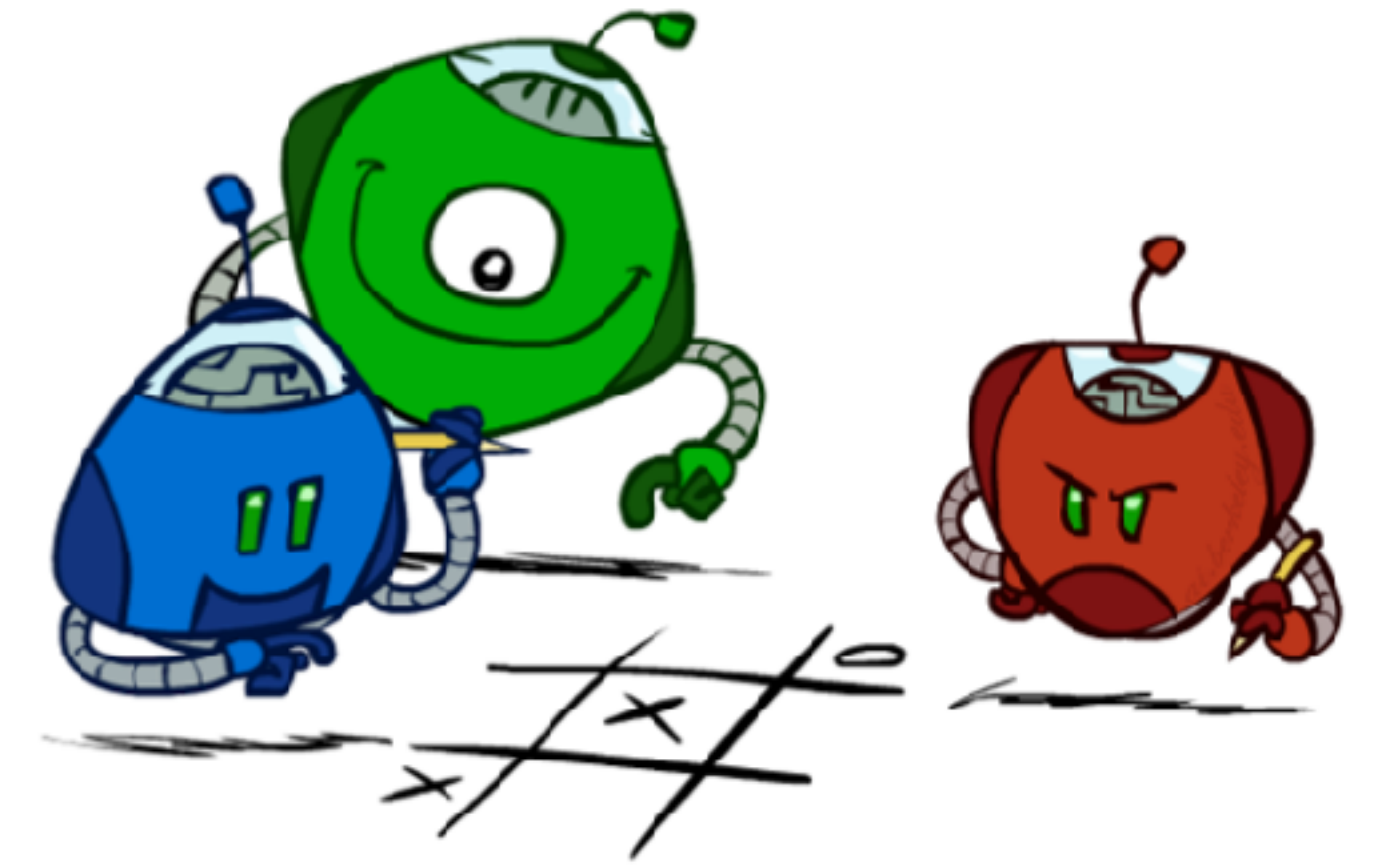
# Mixed Layer Types

- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent
  - Each node computes the appropriate combination of its children



# 83 Multi-player Non-Zero-Sum Games

- What if the game is not zero-sum, or has multiple players?
- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically...



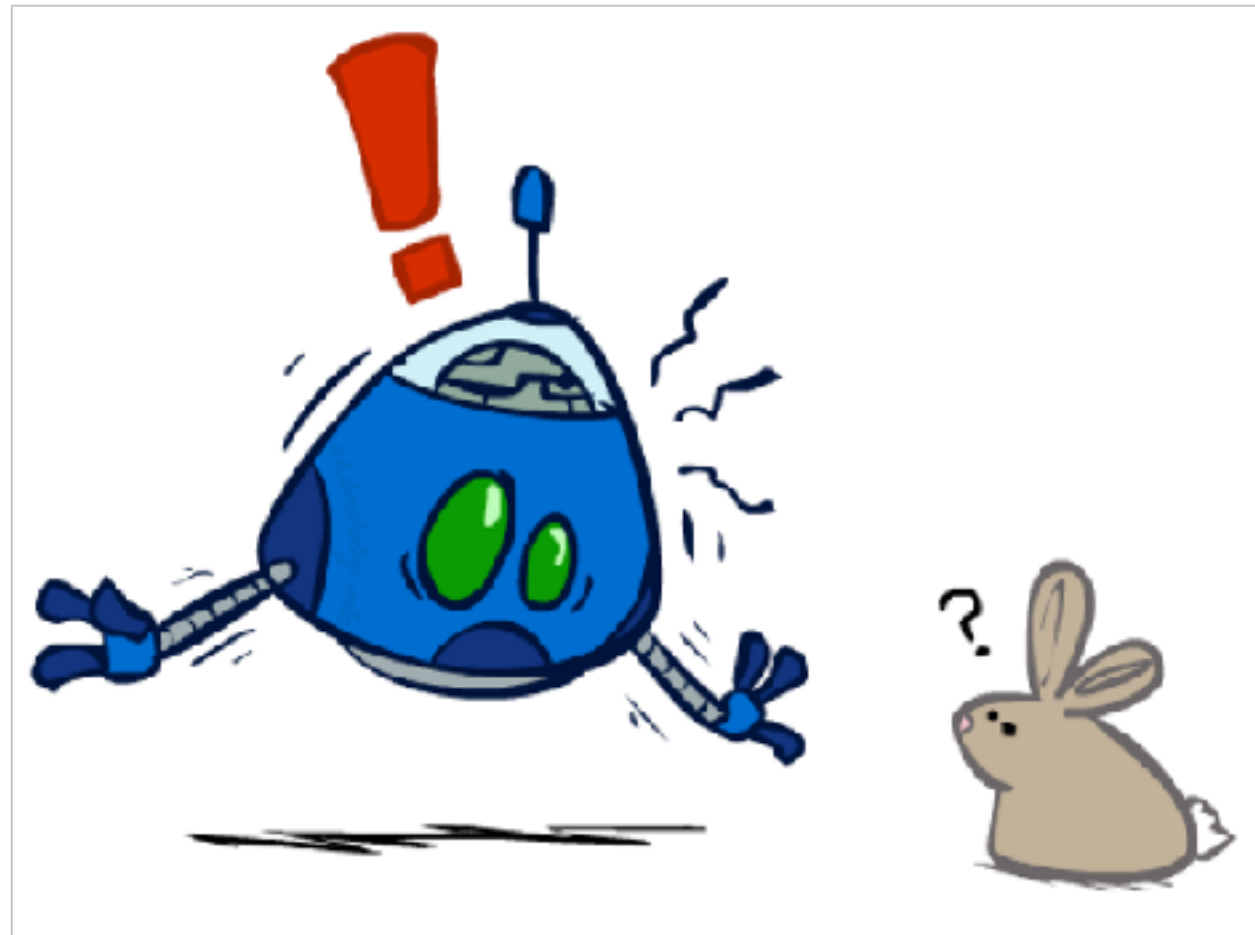
# Utilities

- **Utilities are functions from outcomes (states of the world) to real numbers that describe an agent's preferences**
- Where do utilities come from?
  - In a game, may be simple (+1/-1)
  - Utilities summarize the agent's goals
  - Theorem: any “rational” preferences can be summarized as a utility function
- We hard-wire utilities and let behaviors emerge
  - **Why don't we let agents pick utilities?**
  - **Why don't we prescribe behaviors?**

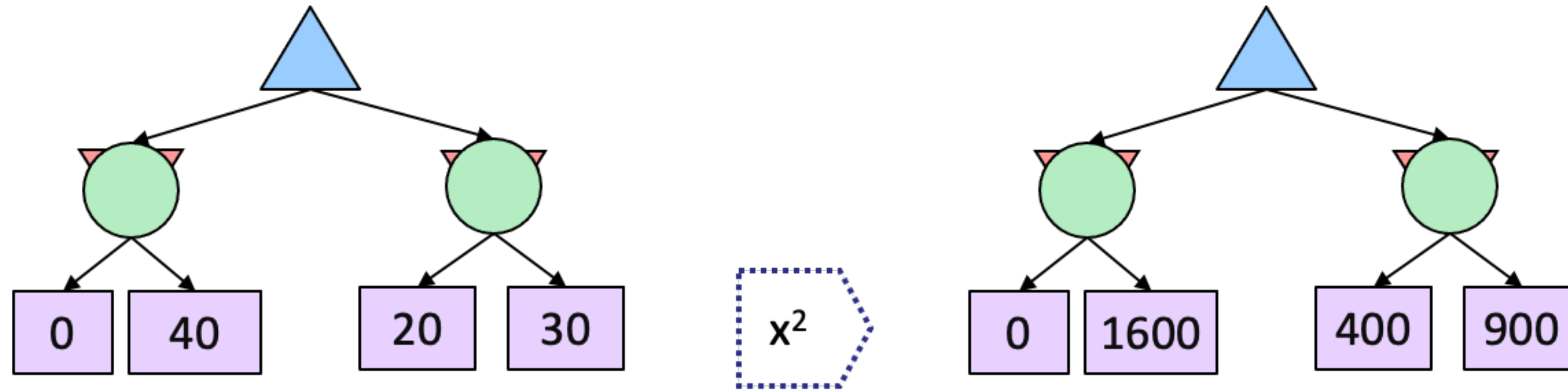


# Maximum Expected Utilities

- ⦿ Why should we average utilities? Why not minimax?
- ⦿ Principle of maximum expected utility:
  - A rational agent should choose the action that **maximizes its expected utility, given its knowledge**



# What Utilities to Use?



- ⦿ For worst-case minimax reasoning, terminal function scale doesn't matter
  - We just want better states to have higher evaluations (get the ordering right)
  - We call this **insensitivity to monotonic transformations**
- ⦿ For average-case expectimax reasoning, we need magnitudes to be meaningful (**we'll talk more about utilities in the future**)

## 87 Summary

- In two-player, discrete, deterministic, turn-taking zero-sum games with perfect information, the **minimax algorithm** can select optimal moves by a depth-first enumeration of the game tree.
- The **alpha–beta search algorithm** computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
- Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we need to **cut the search off** at some point and apply a heuristic evaluation function that estimates the utility of a state.
- An alternative called **Monte Carlo tree search (MCTS)** evaluates states not by applying a heuristic function, but by playing out the game all the way to the end and using the rules of the game to see who won. Since the moves chosen during the playout may not have been optimal moves, the process is repeated multiple times and the evaluation is an average of the results.
- Games of chance can be handled by **expectiminimax**, an extension to the minimax algorithm that evaluates a chance node by taking the average utility of all its children, weighted by the probability of each child.

# Thanks! Q&A