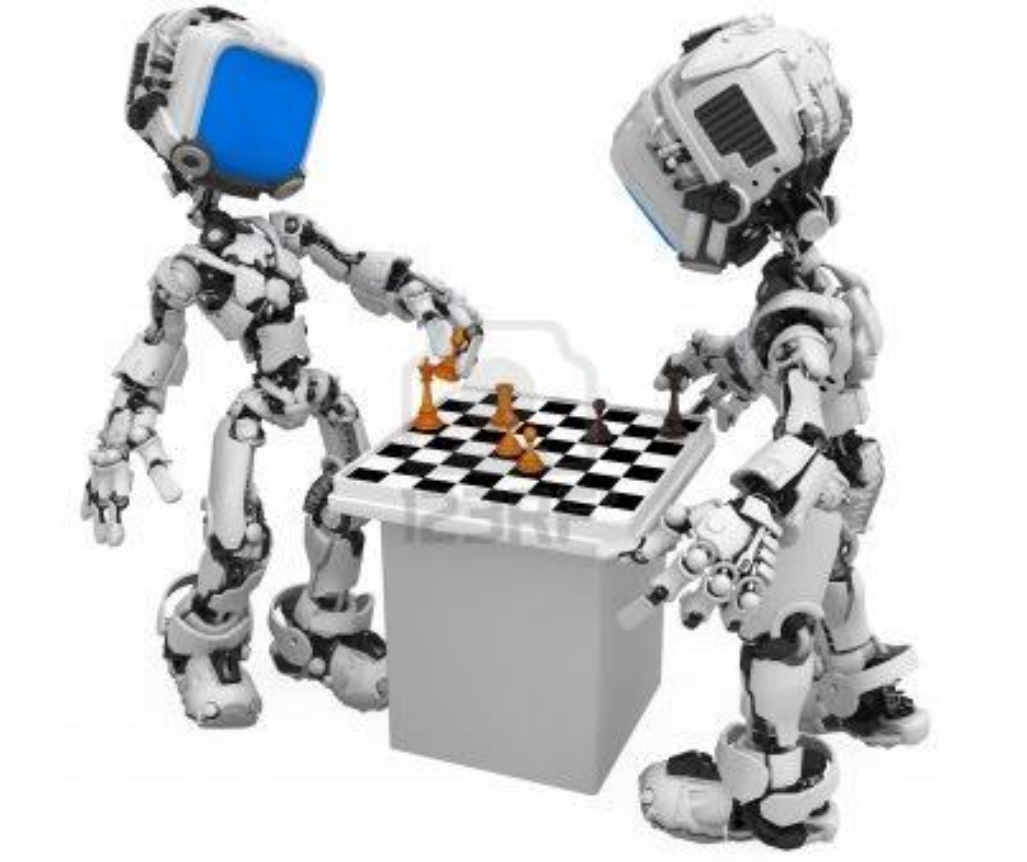
Adversarial Search

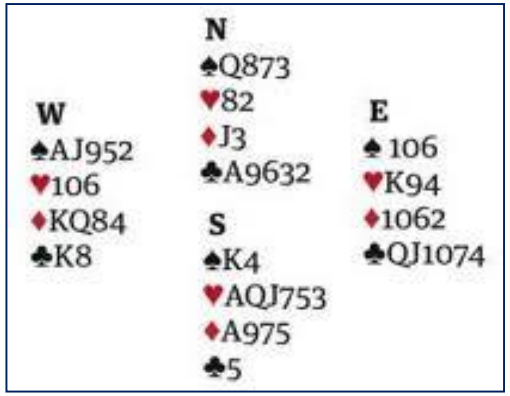
- Multi-player games: Competitive multi-agent environment

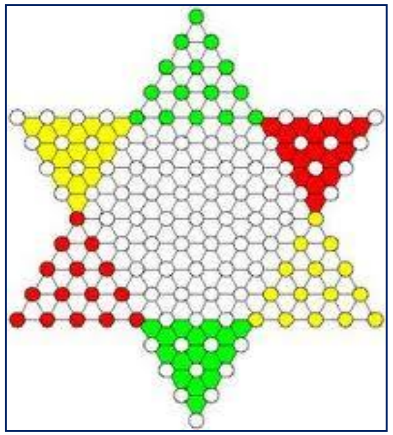


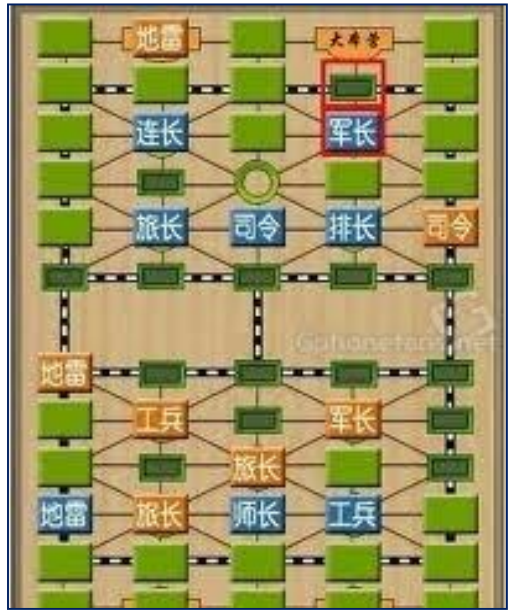
Types of (Turn-Taking) Games

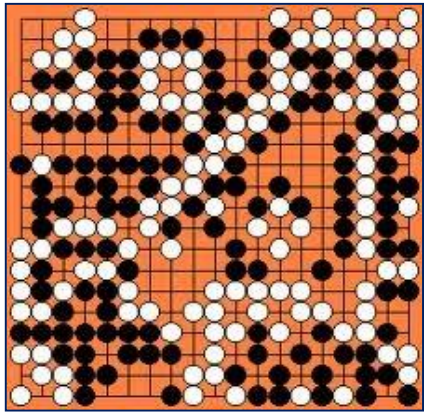
Two factors to consider:

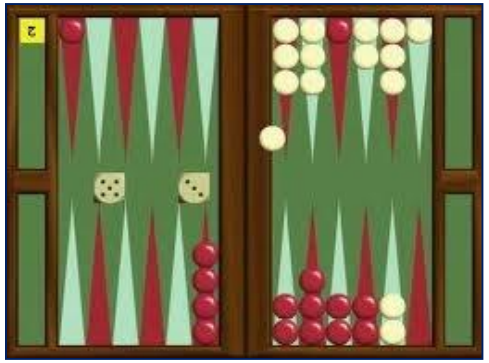
- Deterministic or stochastic (chance)
- Perfect or imperfect information

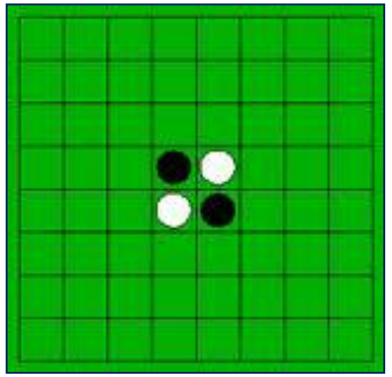












Adversarial Search

We will discuss

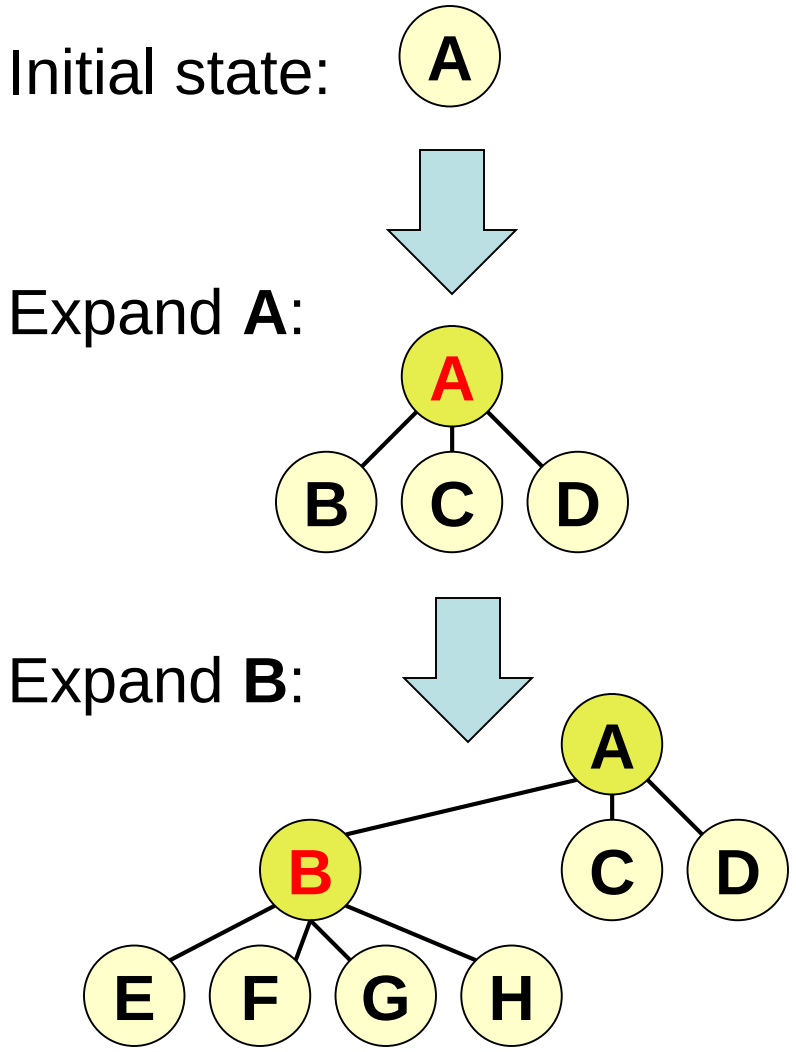
- Two-player, turn-taking, deterministic, zero-sum games with perfect information
- Handling resource limits
- Games with a factor of chance (stochastic games)
- Games with imperfect information
- More recent developments: MCTS, AlphaGo, etc.

We will not discuss these games in this set of slides as they are games that do not involve opponents/competitors, although some concepts are still applicable:

How about these Games?

These are not turn-taking games. What are their different challenges?

Review: Search Tree



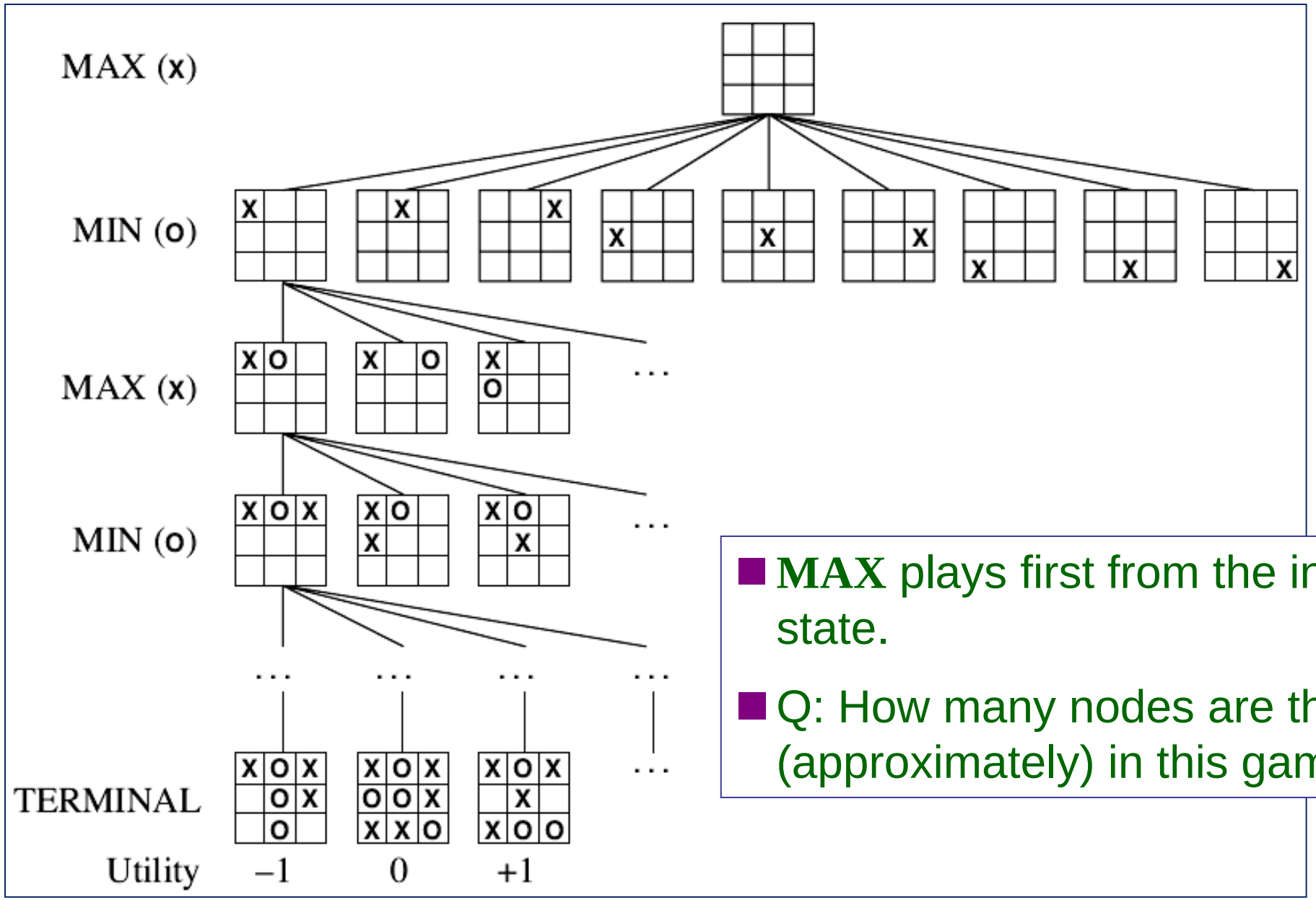
This is because games are modeled as trees:

- Each node corresponds to a state of the environment.
- The tree grows as the search goes on.
- In each iteration, a leaf node is selected for expansion.
  - Searching methods differ by how the node is selected.
  - Children are added for all valid actions from the expanded node.
- In game search, the root is the current state.

Game (Tree) Definition

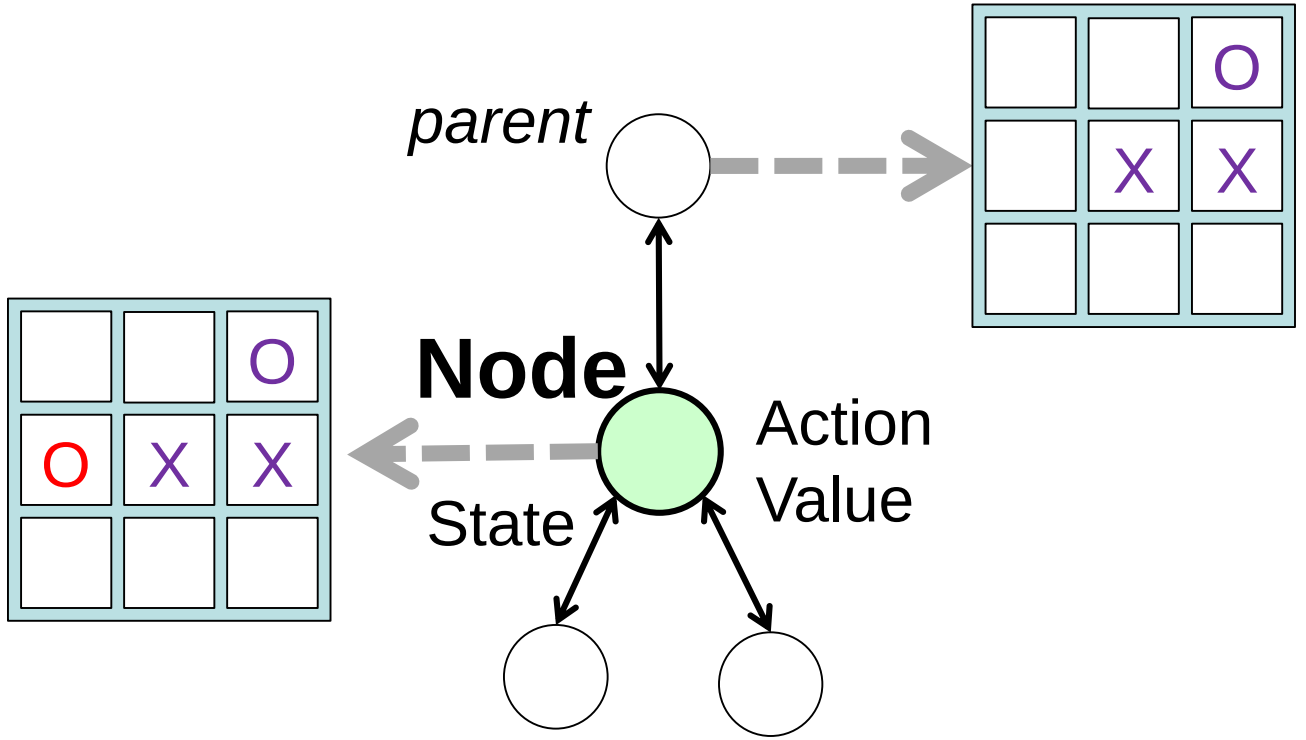
- Initial State S_0: How the game is set up in the beginning.
- **Player(s)**: Which player has the move in state s. (Only for multi-player games.)
- Actions(s) is the set of legal actions in state s.
- Transition model: Result(s,a) is the state reached from doing action a in state s. (This is probabilistic for stochastic games.)
- **Terminal test**: To determine when the game is over.
- **Utility(s,p)** : The utility / objective / payoff function for player p at a terminal state s.
  - **Zero-sum games**: The total utility for all players is a constant.

Game-Tree Example: Tic-Tac-Toe



- **MAX** plays first from the initial state.
- Q: How many nodes are there (approximately) in this game tree?

Game Tree: Information Kept by Nodes



In a game tree, the ***parent*** pointer is kept because the information collected has to be passed back to the root.

Move Selection

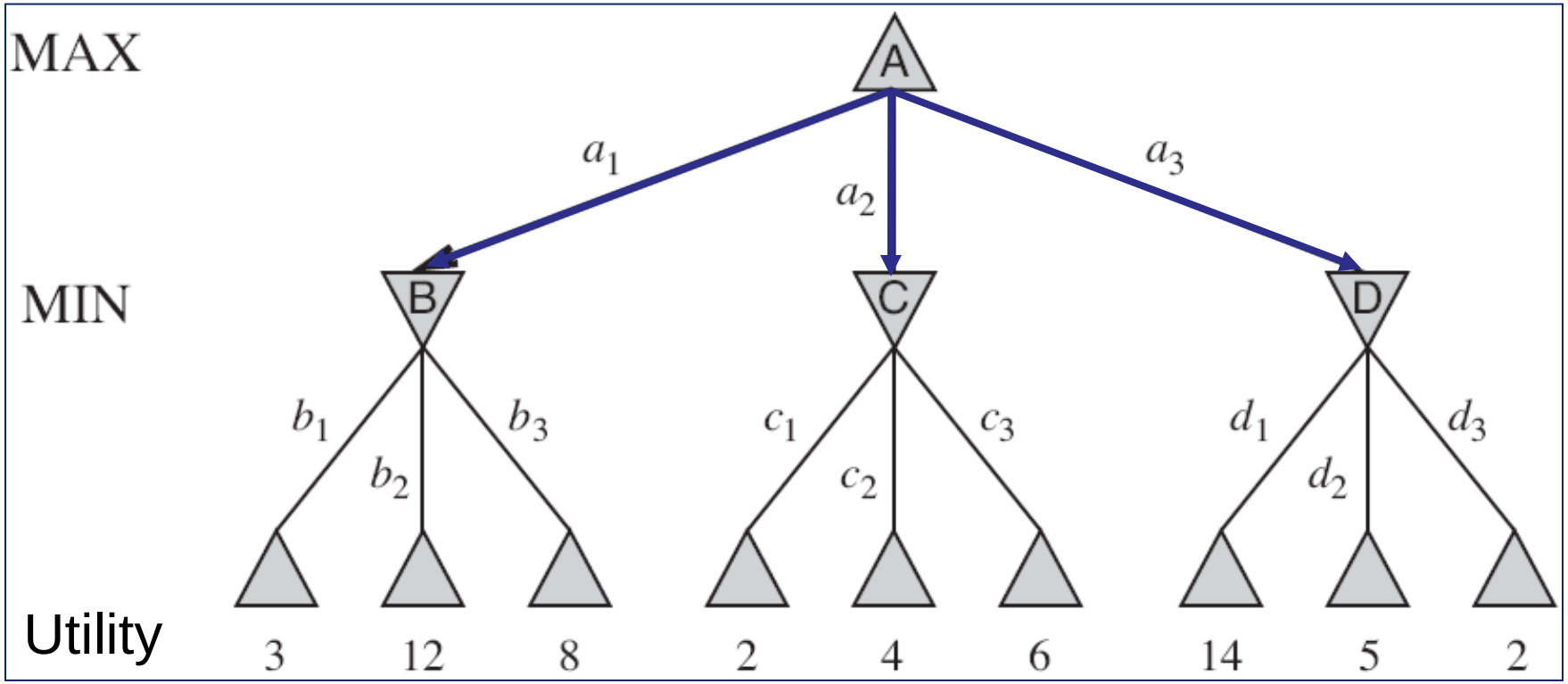
For almost all games, it is impractical to build the whole game tree.

- This is mostly due to a large branching factor b (many legal actions from any given state).
  - Example for chess: $b \approx 35$ and $m \approx 100$ (typical depth to a terminal state): about 35^{100} nodes.
- If the whole game tree can be built (such as in Tic-Tac-Toe), the game is completely "*solved*" because we know the optimal move from any state.
- Without the whole game tree, a new search has to be executed for the selection of each move. The search starts from the current state.

Move Selection

Let's start with two-player turn-taking deterministic games with perfect information.

Q: Consider the 2-ply (2-move) game tree below. The current game state is node A. Which move should **MAX** select?



Computing Node Values: Minimax

Optimal play against a perfect opponent:

- **MAX** nodes (my move): Select the child node with the maximum value.
- **MIN** nodes (the opponent's move): Select the child node with the minimum value "for me". ➔ This assumes that the opponent will choose the play to maximize his/her utility (thus minimizing my utility).

Therefore, we can sum up the concept of **Minimax** as

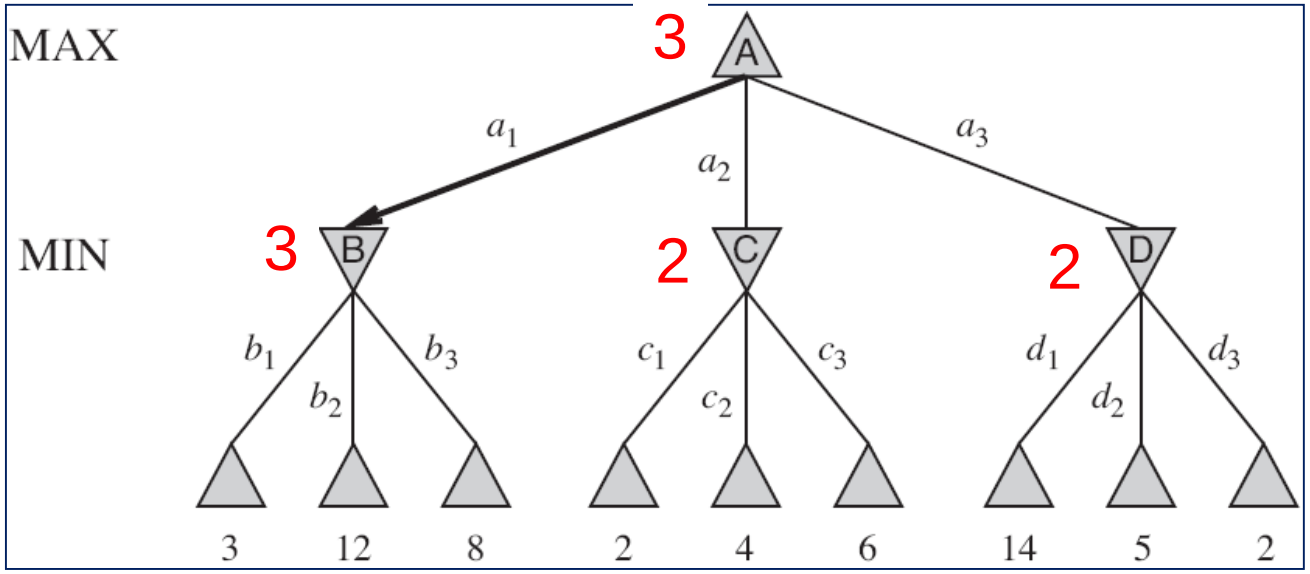
"Determine the worst possible outcome for each alternative, and select the one with *the best 'worst possible outcome'*".

Computing Node Values: Minimax

Values of the nodes are computed in a bottom-up order, and therefore best implemented recursively:

```
function Max-Value(s) returns a utility value
    if Terminal-Test(s) then return Utility(s)
    v ← −∞
    for each a in Actions(s) do
        v ← max(v, Min-Value(Result(s,a))
return v
```

```
function Min-Value(s) returns a utility value
    if Terminal-Test(s) then return Utility(s)
    v ← +∞
    for each a in Actions(s) do
        v ← min(v, Max-Value(Result(s,a))
return v
```



Finally, we select the move with leads to the largest value (B) from the current state (A).

α–β Pruning

- If we can examine the whole game tree, then the minimax method is optimal.
  - Time complexity is $O(b^m)$. Oops! ☹
  - However, do we need to check every path?

Consider the same 2-ply game tree as in the minimax example. We need to compute

max[min(3,12,8), min(2,4,6), min(14,5,2)]

The 9 leaf nodes are examined left to right. The result (my selected move) is the same as

max[min(3,12,8), min(2,?,?), min(14,5,2)]

Why?

α–β Pruning

- Idea: If a partially explored subtree is sure to be worse than the currently known best path, then we do not need to explore that subtree anymore.
  - The resulting choice of action is the same as the minimax method. ☺
- α = the value of the best (highest-valued) choice for **MAX** we have found so far along the path.
- β = the value of the best (lowest-valued) choice for **MIN** we have found so far along the path.
- α and β are passed down to cause pruning. (They are information gathered when exploring previous subtrees.)
- Computed minimax values of the nodes are passed up (as in the original minimax search).

α–β Pruning

```
function Max-Value(s,α,β) returns a utility value
    if Terminal-Test(s) then return Utility(s)
    v ← −∞
    for each a in Actions(s) do
        v ← max(v, Min-Value(Result(s,a),α,β)
        if v ≥ β then return v
        α ← max(α,v)
    return v
```

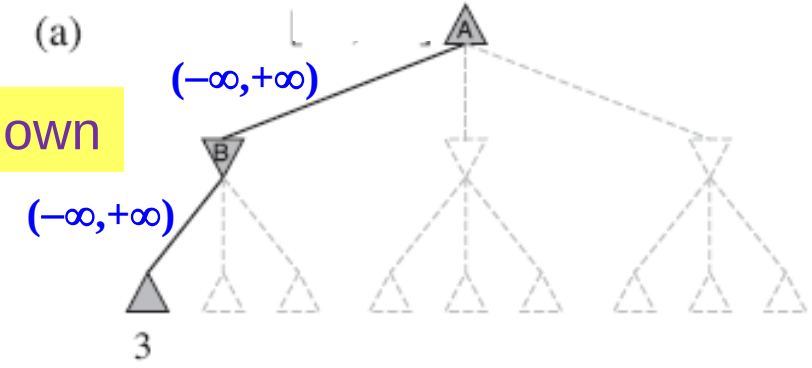
```
function Min-Value(s,α,β) returns a utility value
    if Terminal-Test(s) then return Utility(s)
    v ← +∞
    for each a in Actions(s) do
        v ← min(v, Max-Value(Result(s,a),α,β)
        if v ≤ α then return v
        β ← min(β,v)
    return v
```

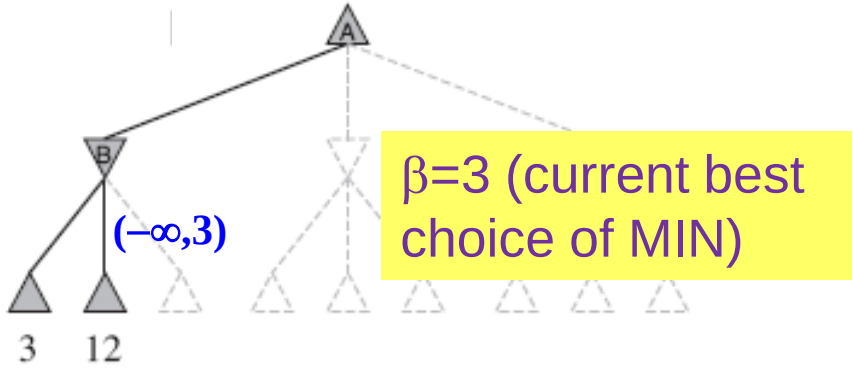
(Next slide)
Let us trace the process starting with the initial call

Max-Value(A,−∞,+∞)

α–β Pruning

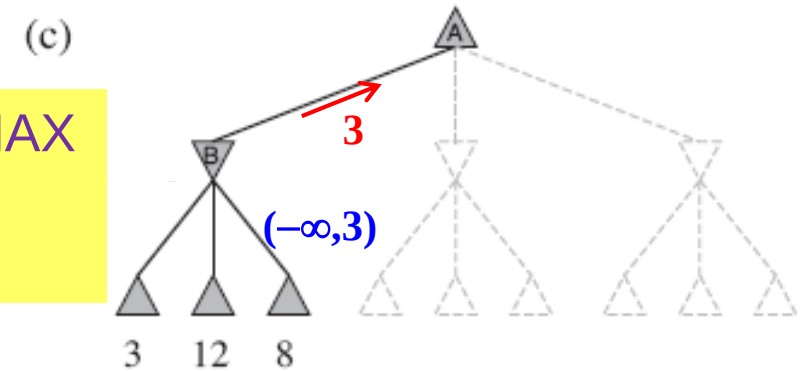
First path down

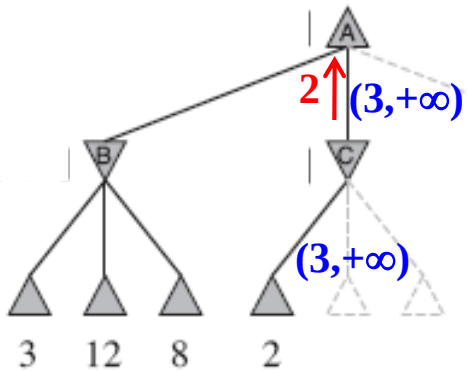




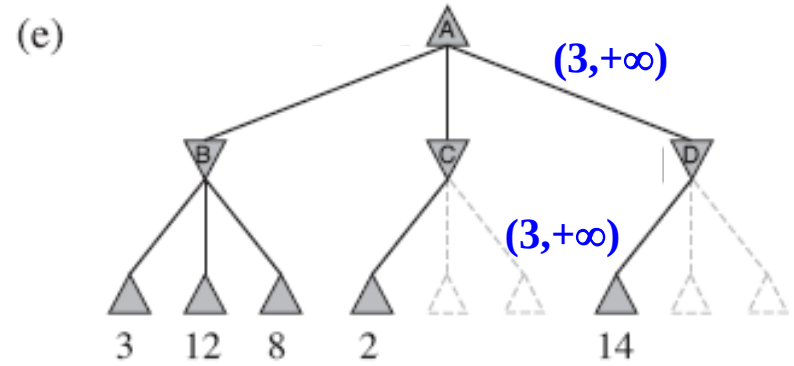
β=3 (current best choice of MIN)

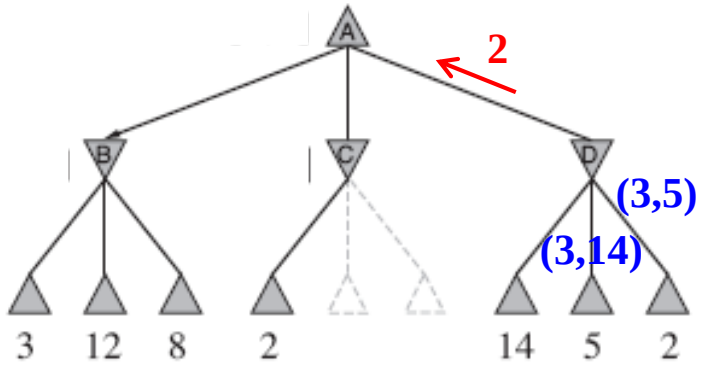
"3" passed to MAX as the current minimax value





2<3 found, so C is definitely worse than B for MAX; abandon this branch.





Move Ordering for α–β Pruning

- Comparing the expansions of nodes C and D, we can see that for a **MIN** node, it is best to expand the lowest-valued successor first (and for a **MAX** node, the highest-valued successor first).
- In the optimal case, the time complexity of minimax with α–β pruning becomes $O(b^{m/2})$. This means that the search can go twice as deep in the same amount of time.
- However, the exact minimax value of a node is not known until the search below that node is complete. ➔ Node ordering can only be done with some "evaluation function" that can be computed from a node's state without search, and is therefore approximate.

Resource Limit

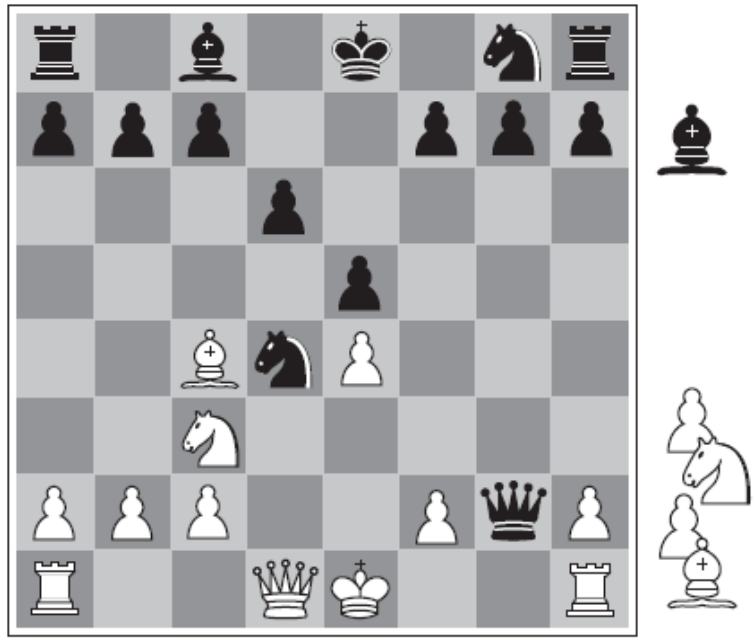
- As we can not search to the terminal states of the game tree, the standard solutions are:
  - **Cutoff test**: This replaces terminal test for determining whether to terminate the search along a path. The most common form is a depth limit. Another option is time limit.
  - **Evaluation function**: The "goodness" of a state estimated without searching the game tree.

Evaluation Function

- Evaluation functions are not part of the game rules. They usually come from expert knowledge and past experiences (i.e., learning).
- The minimax search with cutoff test uses the evaluation functions at the leaf nodes in place of the utility functions.
  - Example: For chess, if we can examine 10^6 nodes per move (about 35^4), then minimax with α–β pruning can reach depth 8 in the tree ➔ pretty good chess program.
- Many (classical) game AI programs combine minimax with α–β pruning with sophisticated evaluation functions for good results.
- Ideally, the evaluation function of a state should be strongly correlated to the "chance of winning" from that state.

Evaluation Function: Example

A simple way to build evaluation functions is by linearly combining features. Example: For chess, a beginner's evaluation function (for **White**) is



[#pawns(W) - #pawns(B)] x 1

[#bishops(W) - #bishops(B)] x 3

[#knights(W) - #knights(B)] x 3

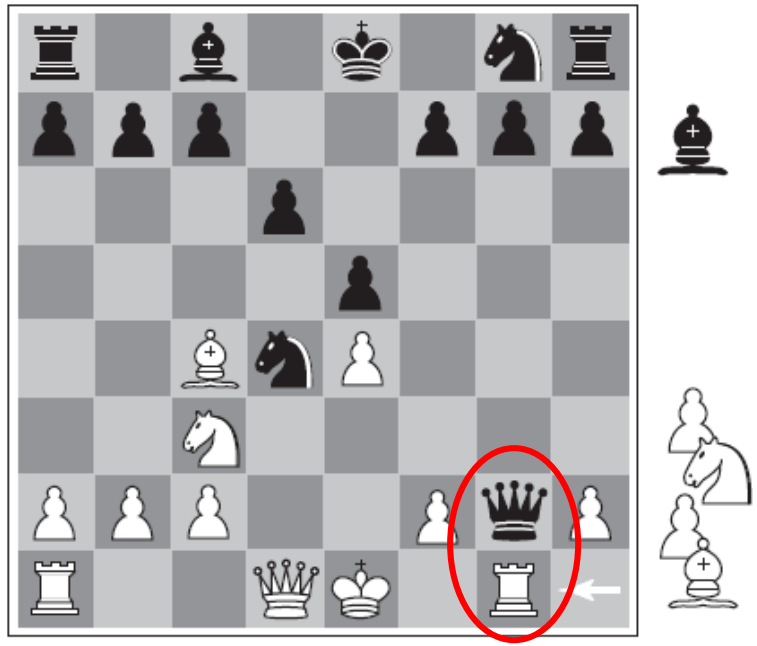
[#rooks(W) - #rooks(B)] x 5

[#queens(W) - #queens(B)] x 9

Now can you compute the evaluation function of this state for **white**? Does it favor **black** or **white**?

A Problem with Depth-Limited Search

This state has the same evaluation function as the previous one. However, if the search depth is extended by one ply, we will see that the white rook can capture the black queen, and the resulting state favors **White**.



- **Quiescent search**: Extend the cutoff search along a path until a quiescent state, where immediate additional moves will not significantly change the evaluation function.
  - Example: When the next move is a capture, the line of search should be extended.

Improving Depth-Limited Minimax Search

- **Singular extension**: Consider a potential move that is "clearly better than others" beyond the normal depth limit.
- **Forward pruning**: Follow fewer paths (selected using rules or evaluation functions) so that we can go deeper.
- **Transposition table**: Checking repeated states (similar to the explored list in graph-search) and reuse the computed evaluation functions / minimax values.
  - Transpositions are usually caused by different orderings of the same set of moves.
  - It can happen that there are too many visited states to be stored in practice, and only a subset can be kept.

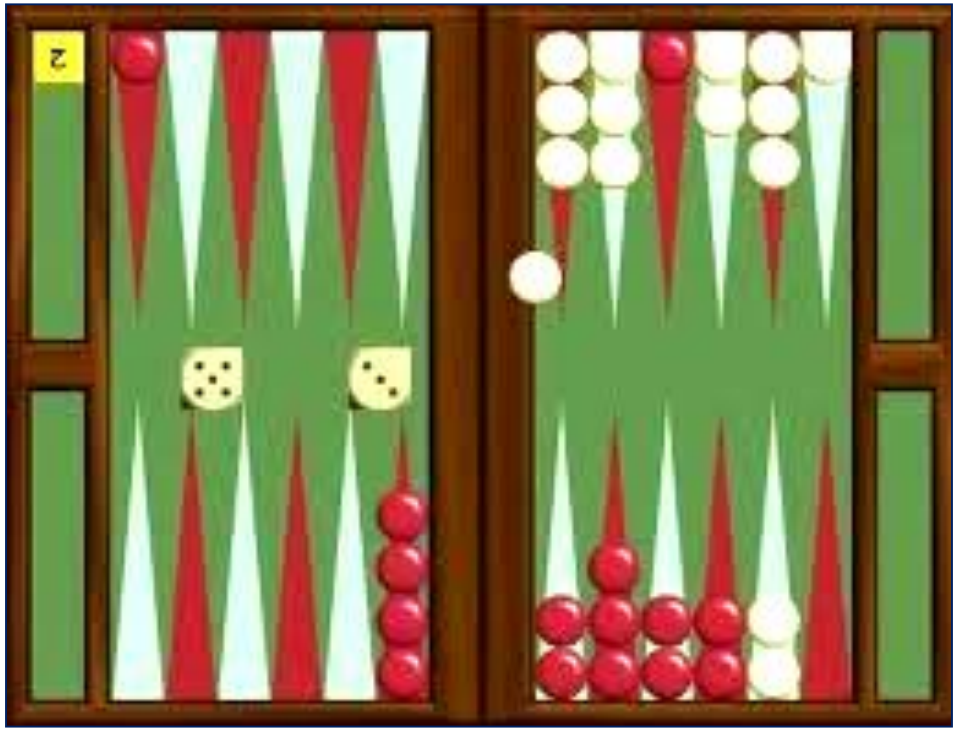
Lookup vs. Search

Lookup tables are useful for speeding up the decision making for opening and endgames.

- Opening: Usually from human experts or past experiences.
- Endgame: Mostly can be solved exactly using minimax search (computers are better than human).
  - If all the endgame states (e.g., with 5 or fewer remaining pieces in chess) are solved exactly, then the minimax search only needs to reach these endgame states (instead of the actual end of game) to be exact.

Stochastic Games

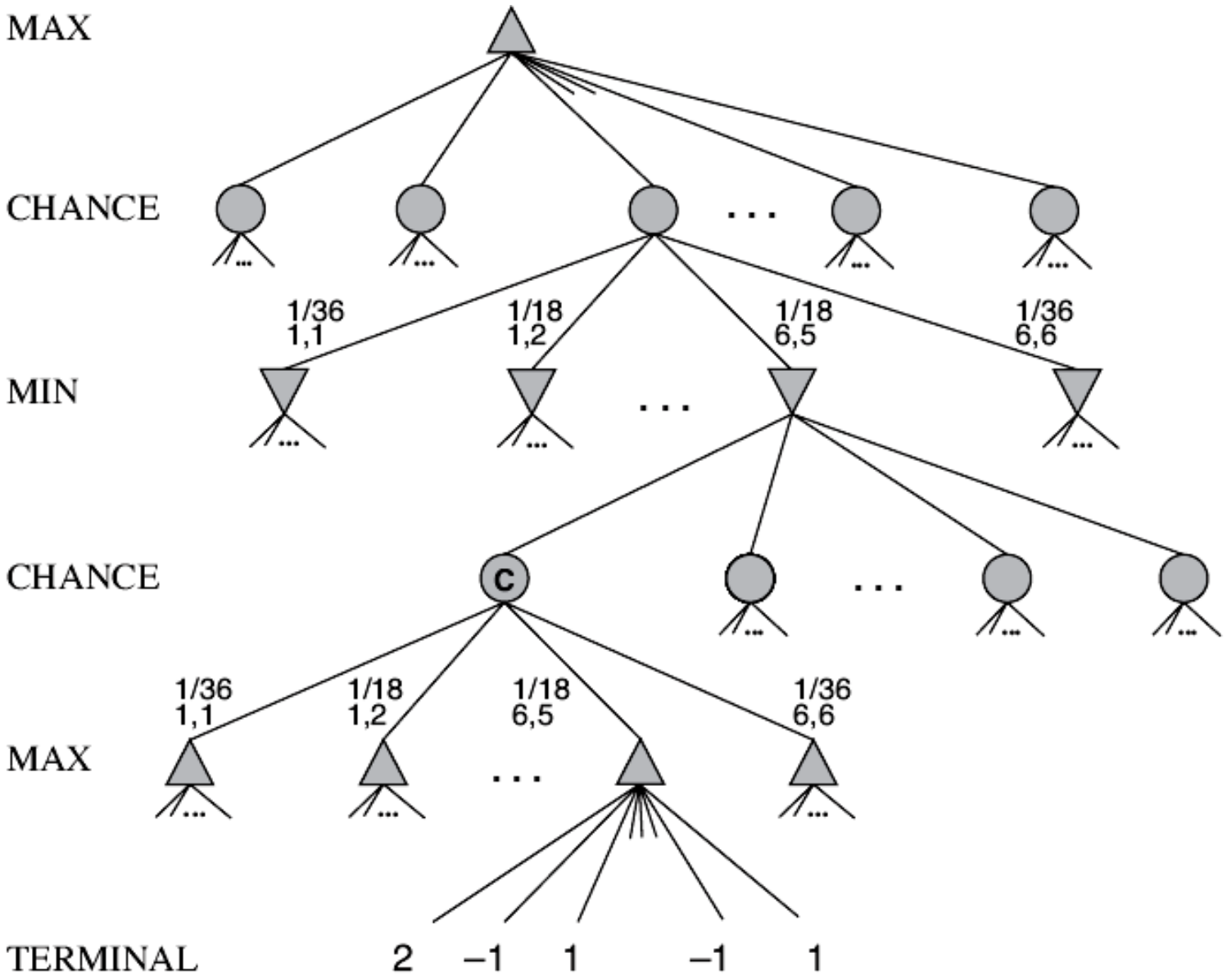
Example game: **backgammon**. When we are to decide on a move, we do not know the set of the opponent's legal moves because the opponent has not rolled the dice yet.



How do we do the game tree search in this case?

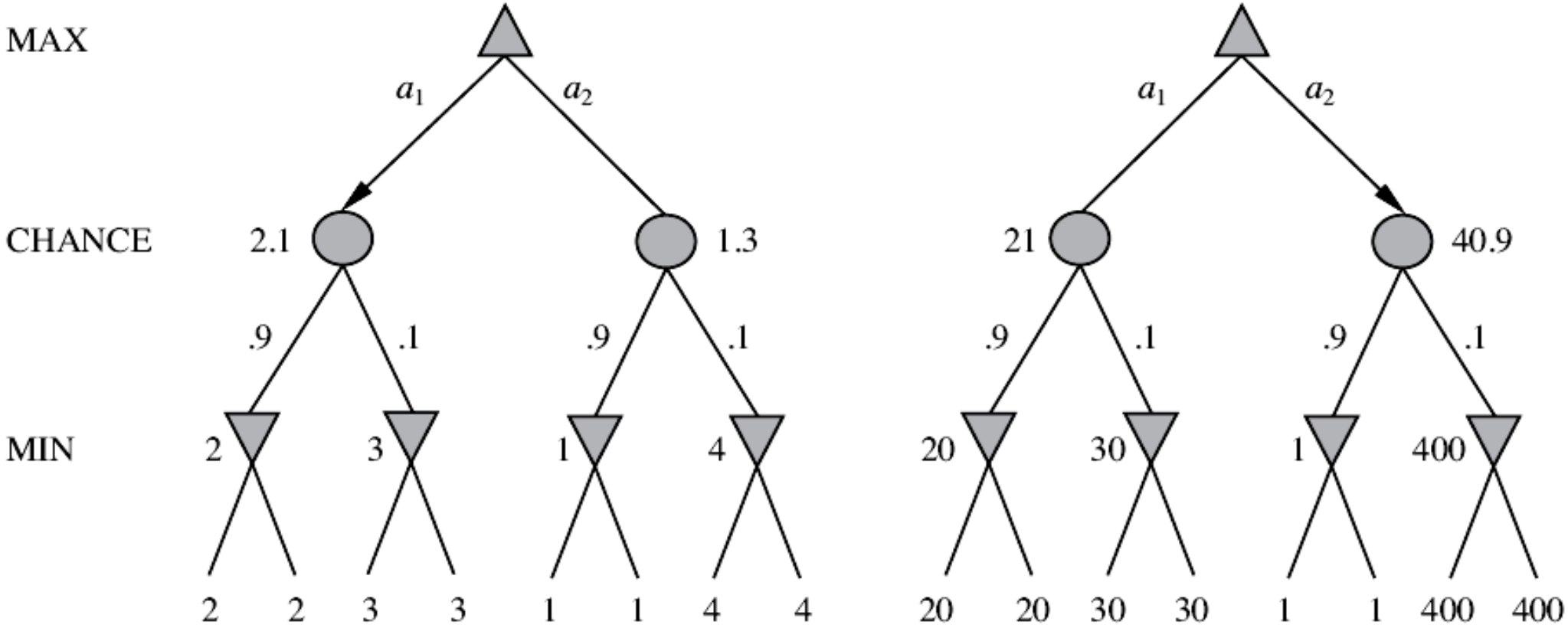
Chance Nodes

Each successor of a chance node corresponds to a distinct dice roll. A chance node computes an "expected" minimax value.



Chance Nodes and Evaluation Functions

- Without chance nodes, the evaluation functions of different states only need to have the correct ordering.
- With chance nodes, the evaluation function of a state should be proportional to its **expected utility** (illustrated below).



Complexity in Stochastic Games

- Let n be the branching factor of a chance node. The complexity is $O(b^m n^m)$ compared to $O(b^m)$ for standard minimax.
  - Example: $n = 21$ for backgammon.
- It is difficult to go deep into the game tree:
  - Complexity
  - A state that are several levels down the tree has little chance to actually occur in the game.
- Example for the backgammon program TD-Gammon:
  - Search is only 2-ply to 3-ply.
  - Very good evaluation function learned from playing.
  - Plays at the level of human world champion.

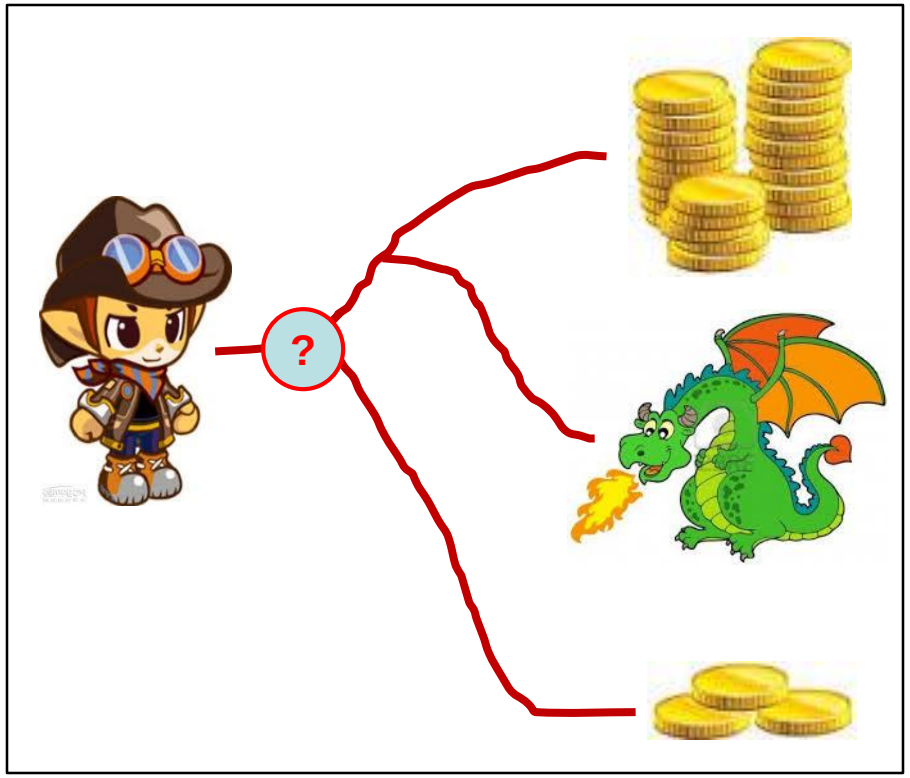
Games with Imperfect Information

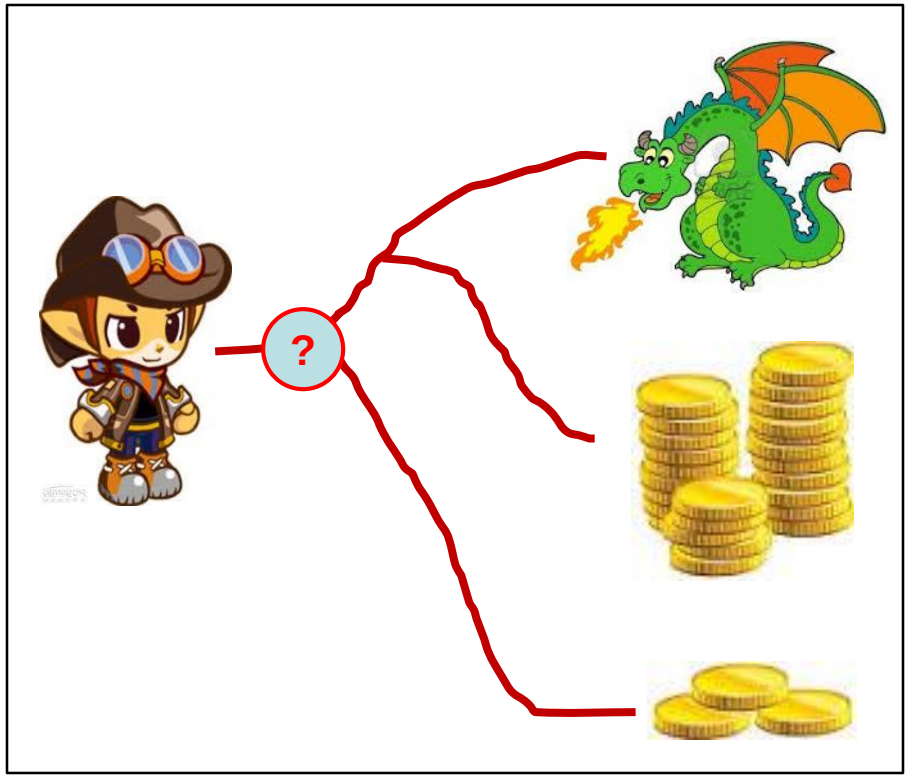
- The most common examples are card games (such as Bridge), where the other players' hands are unknown in the beginning.
- A common way to evaluate an action is by sampling:
  - Generate many exemplars for the missing information.
  - Determine the minimax values for the action in all the fully observable exemplar games.
  - Use the average as the minimax value of the action.
  - However, this sometimes lead to plays that are not good…

Games with Imperfect Information

This is a commonsense example illustrating "the value of an action with missing information is not the same as the average of its possible values if the missing information is known".

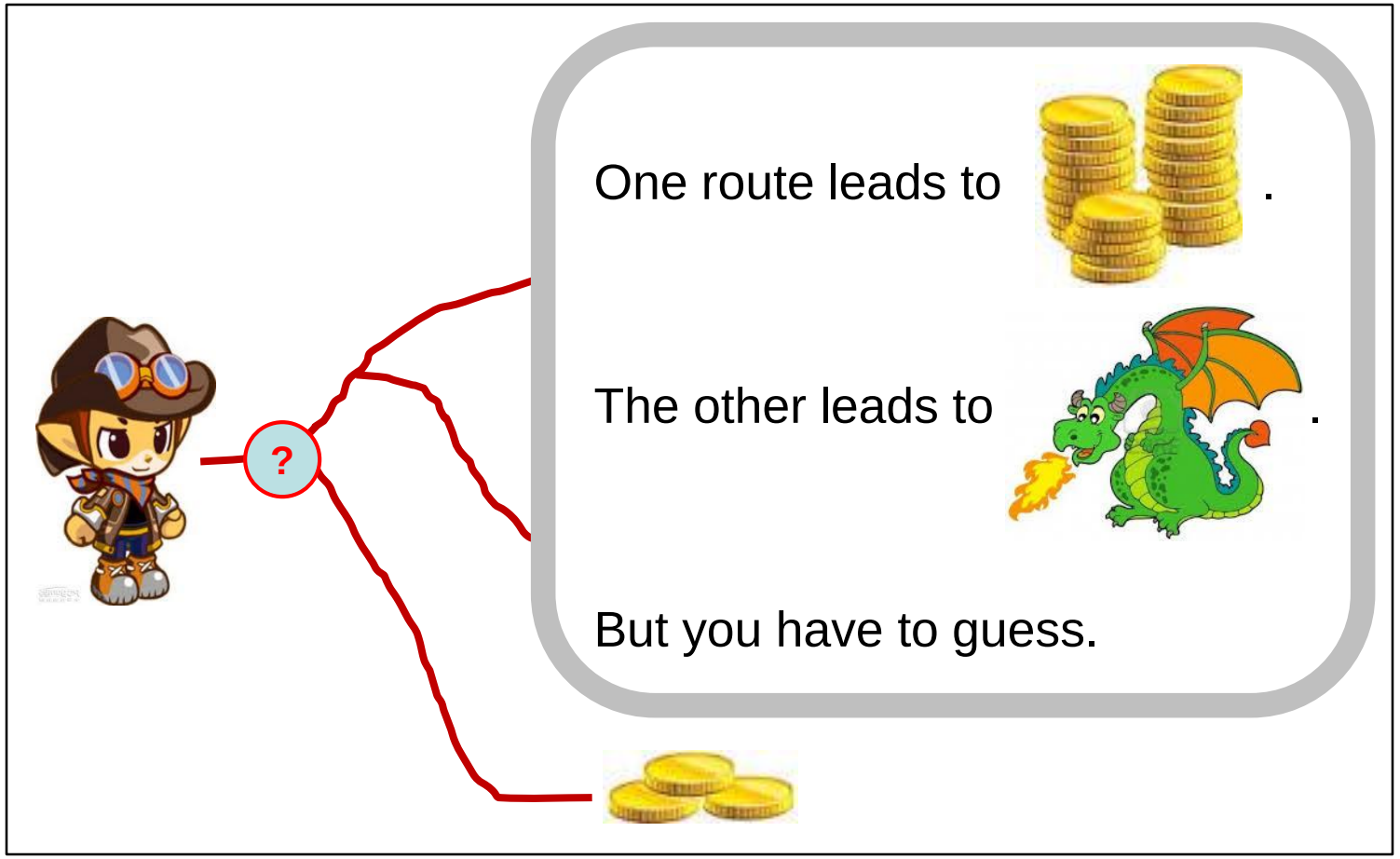
Q: In these two cases, which route will you choose at the first fork?





Games with Imperfect Information

Q: Now consider the case below. Which route will you choose at the first fork?



Games with Imperfect Information

- Problem with the simple method of averaging: The intuition that the value of an action is the average of its values in all the actual states is wrong.
- When playing games with imperfect information, the agent may have these rational behaviors:
  - Act to obtain information
  - Pass signals to partners (e.g., in Bridge)
  - Attempt to hide information or mislead opponents

State-of-the-Art, 25+ Years Ago

- **Deep Blue** (1997) from IBM is probably the best known achievement of classical game playing, beating the world chess champion at that time.

- Computers were already better than human at simpler games like, say, Othello.

State-of-the-Art, 25+ Years Ago

- **Deep Blue** uses
  - Parallel processing to evaluate up to 2×10^8 nodes per second. Minimax with α–β pruning, 12/14-ply game trees, singular extension up to 40 steps.
  - Evaluation function of about 8000 features (many for special cases) and many tunable parameters (can be adjusted to match particular opponent styles).
  - Complete endgame database of 5 or fewer pieces.
  - A dedicated computer (not a program) containing 30 processors and ICs specially designed for playing chess.

Beyond Classical Game Tree Search

- Classical game tree search suffers from the fact that it attempts to be exhaustive (up to a certain depth) in order to be optimal (only an illusion).
- Without the full game tree, it can be optimal only if the evaluation function correlates with "probability of winning" perfectly, but this is way too difficult.
- As a result, there was little progress of computer Go for many years. (Can you estimate the complete game tree size?)
- Nearly 10 years after Deep Blue, a new search algorithm jumped into the scene and basically took over (next side); it got everyone's attention by giving computer Go a big boost of performance.

Monte-Carlo Tree Search (MCTS)

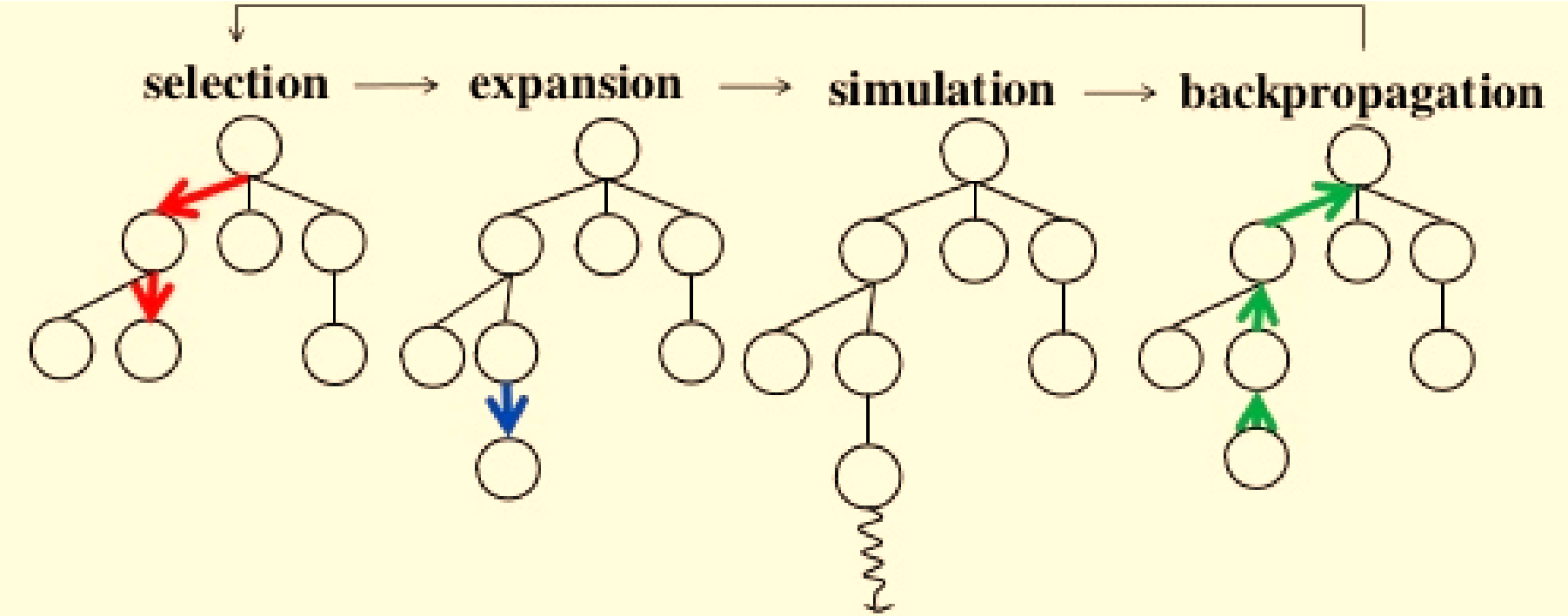
- This is more like a **best-first search** (the utility function will be explained later).
- The game tree is **sampled** (instead of traversed, as in Minimax search). The sampling gradually focuses on more promising paths based on past experiences (sampled search paths).

  ➔ This is why MCTS is considered a form of **reinforcement learning**.
- Information in each node: Count of visits, and Value.
  - "Count of visits" strongly correlates with the "goodness of node" because of the search strategy, which selects more promising paths more frequently.

MCTS Steps

- Four phases (an iteration in MCTS, for sampling a path in the game tree):
  - **Selection**: Move down the tree to select a leaf node. (At each node on the path, the selection of a child is based on the utility function.)
  - **Expansion**: If the selected leaf node has been sampled before, add its children to the tree, and select a child.
  - **Simulation** (rollout / play-out): Play a test game from the selected leaf node until the result (win/loss) is known.
  - **Back-propagation**: Use the result to update the tree nodes along the current path.

MCTS Steps



https://www.slideshare.net/ftgaic/application-of-monte-carlo-tree-search-in-a-fighting-game-ai-gcce2016

- For the opponent's nodes, child selection is based on the value from the opponent's perspective.
- This is an "any-time" algorithm: You can stop at any time and return the current best action.
- The best action returned is the one with the most visits.

Search: Exploitation vs. Exploration

- All non-exhaustive search methods require a balance of exploitation and exploration.
- **Exploitation**: Follow the best known path (you're more confident of its outcome).
- **Exploration**: Try unfamiliar, less traveled, paths (you may get huge rewards, or you may lose badly).
- The typical strategy is to do more exploration in the beginning (to gather information), and more exploitation in later stages of the search.

What other search algorithms (or optimization algorithms) have mechanisms that handle the exploration-exploitation tradeoff?

Search: Exploitation vs. Exploration

- The typical utility function used in MCTS, the UCB (upper confidence bound), originates from studies on the multi-armed bandit problem:
  - Many slot machines, each with different (and unknown) chances of rewards.
  - How to maximize the total reward given fixed initial money?
- This is the utility function:

$$UCB(q) = X(q) + c\sqrt{\frac{\log_{10} n}{n(q)}}$$

exploitation

exploration

q: a choice (arm)

$X(q)$: current value (reward ratio) of q

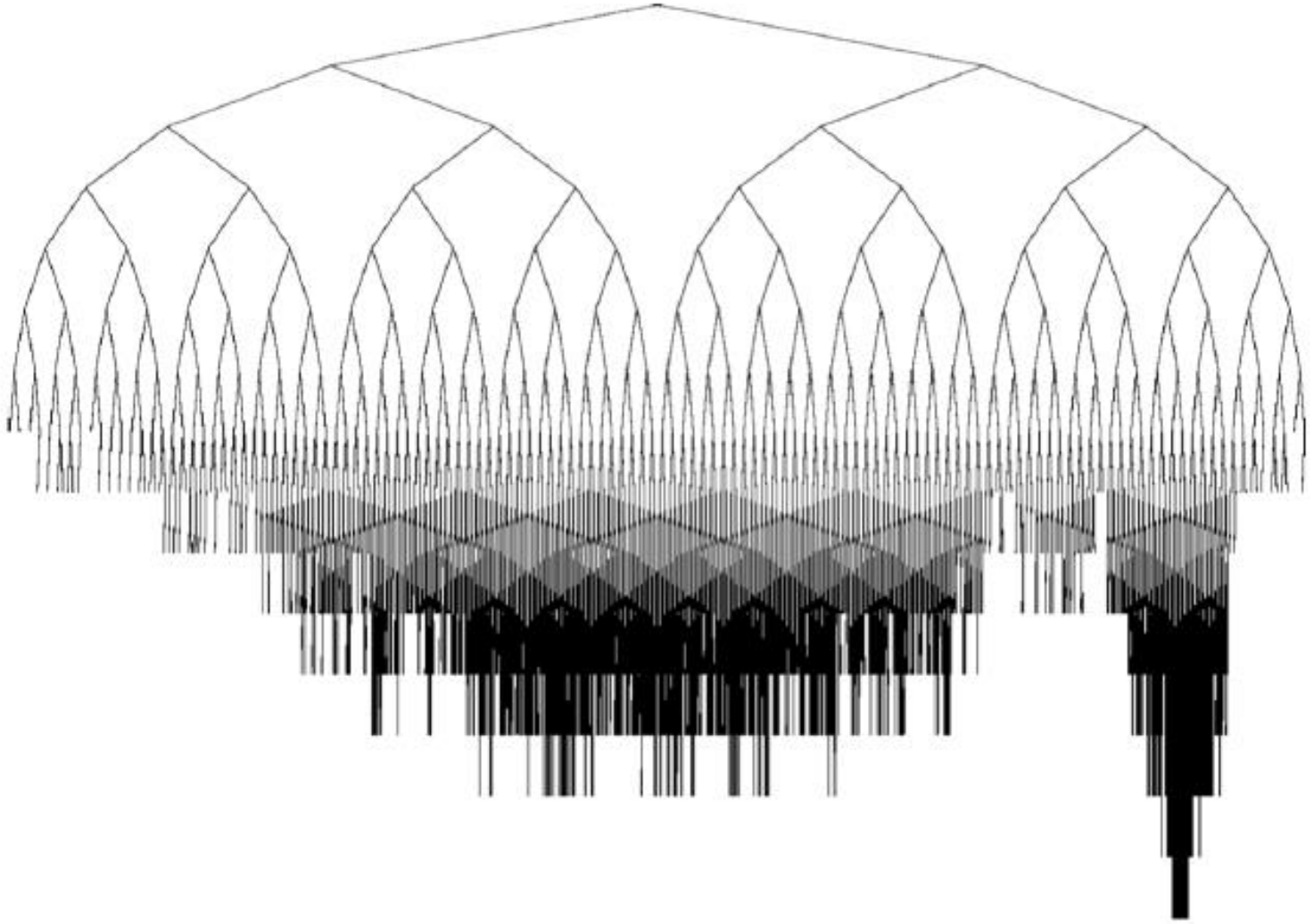
$n(q)$: times of playing q

n: times of playing all choices

c: a weighting constant

MCTS Tree Example

The effect of exploration and exploitation is visible here.



https://ai.stackexchange.com/questions/23811/how-does-the-mcts-tree-look-like

AlphaGo Lee (2016): The Ideas

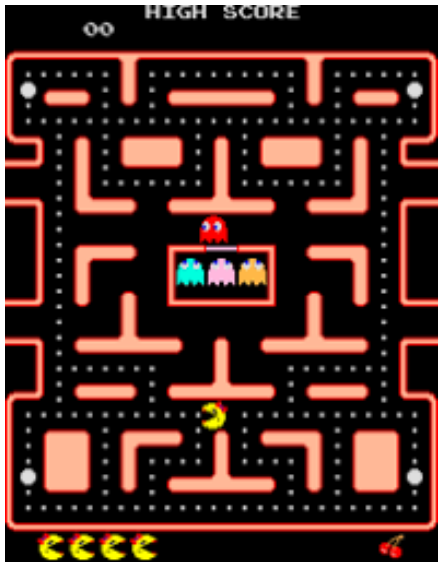
- MCTS game tree with two neural networks:
  - **Value network**: For estimating the "value" of nodes.
  - **Policy network**: For move selection.
  - The networks are trained with human expert data and then self-play results (**reinforcement learning**).
- MCTS steps:
  - Selection: "action probabilities" + "exploration term"
  - Expansion: Children are given "action probabilities" (policy network) and values (value network).
  - Simulation: A play-out to the end (policy network).
  - Back-propagation: The node value passed back is a combination of the given value (value network) and the simulation result.

General Game Playing / Game AI

- General Game AIs: Given only the rules, learn all the strategies automatically. (So a single AI agent can learn to play multiple games.)
- Alpha Go Zero (2017):
  - Start with only the game rules.
  - MCTS with a single network (value network) and NO play-out (simulations replaced by estimations by the value network).
  - Learn purely by playing against many versions of itself.
  - Learning curve: 3 hrs (beginner), 19 hrs (somewhat skilled human player), 3 days (level of Alpha Go Lee), 21 days (level of Alpha Go Master)
  - Mastering other board games (chess, shogi, Chinese chess, etc.) even faster.

Beyond Board Games

- Many other types of games have been tried (and some "conquered") by AI.
- Examples:
  - Trivia games (e.g., Jeopardy, IBM, 2011)
  - Arcade games (e.g., Ms. Pacman, Microsoft, 2017)
  - Texas Poker (2017)
  - etc.



Beyond Board Games

More recent examples of larger-scale games:

- DeepMind AlphaStar for StarCraft II (2017-2019)
  - Combined neural networks and evolutionary computation
  - Initial training using human-vs-human data (more like AlphaGo Lee than AlphaGo Zero)
  - Plays the 1v1 mode of the game.
  - An earlier version (2018) was considered "unfair" when beating top human players; a more realistic version was then developed.
  - In 2019, it reaches the "grandmaster" level of human players.

Beyond Board Games

More recent examples of larger-scale games:

- OpenAI Five for DOTA 2 (2016-2019)
  - Based on LSTM (a type of NNs for long-range sequential data)
  - Reinforcement learning
  - Use inputs from API (not screen based)
  - Real-time actions learned by playing against itself (180 years worth of experience)
  - In 2019, it reaches a level beyond top human players.