ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

IT3160E
Introduction to Artificial Intelligence

Chapter 3 – Problem solving

Part 4: Adversarial search

Lecturer:

Muriel VISANI

Acknowledgements:

Le Thanh Huong

Tran Duc Khanh

Department of Information Systems

School of Information and Communication Technology - HUST

Content of the course

- Chapter 1: Introduction
- Chapter 2: Intelligent agents
- Chapter 3: Problem Solving
  - Search algorithms, adversarial search
  - Constraint Satisfaction Problems
- Chapter 4: Knowledge and Inference
  - Knowledge representation
  - Propositional and first-order logic
- Chapter 5: Uncertain knowledge and reasoning
- Chapter 6: Advanced topics
  - Machine learning
  - Computer Vision

Outline

- Chapter 3 – part 1: un-informed (basic) algorithms
- Chapter3 - part 2: informed search strategies in graphs
- Chapter 3 – part 3: advanced search strategies
- Chapter 3 – part 4: adversarial search
  - Introduction and applications to games
  - 2-player game problem formulation
  - Minimax algorithm
  - α-β pruning
    - Definition
    - Properties and limitations
    - How to twist it to make imperfect, real-time decisions
  - Some insights on non-deterministic games / games with imperfect info / games with imperfect opponent
  - Summary
  - Exercises / homework

Goal of this Lecture

Goal	Description of the goal or output requirement	Output division/ Level (I/T/U)
M1	Understand basic concepts and techniques of AI	1.2

Introduction and applications to games

Introduction

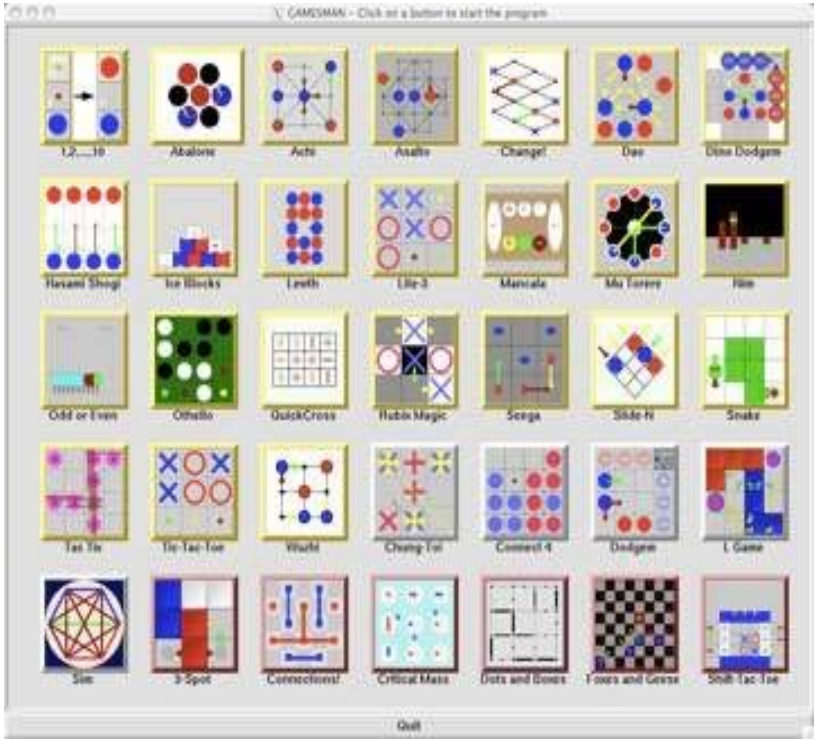
- Adversarial search is useful for **multi-agent**, **competitive** environments
- **Multi-agent** environment:
  - Any given agent needs to consider the actions of other agents (past, current and / or future) to select their next action
  - Introduce possible contingencies into the agent's problem-solving process
- For **adversarial** search, the environment is **competitive** (and not collaborative)
  - Agents have conflicting goals

Introduction

- Hence, natural application to **games** !

Games *v.s.* search problems

- **"Usual" search** (parts 1-3): no adversary
  - Solution: (heuristic) method for reaching goal
  - Heuristics and CSP techniques (see next lecture) can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities



[https://www.cpp.edu/~ftang/courses/CS420/notes/adversarial%20search.pdf]

- **Games**: adversarial search
  - "Unpredictable" opponent
  - Solution is a **strategy**
    - Strategy specifies the "best" move (action) for every possible opponent reply
  - Time limits often force the search for an *approximate* solution
  - Evaluation function: evaluate "goodness" of game actions

Major assumptions about adversarial search

- Majors differences with other types of search: in adversarial search:
  - Only an agent's actions change the world
    - No other factor of change, beyond the players (and possibly chance)
    - The problem is usually very well defined
  - Turn-taking: player A plays, then player B, etc
  - The final reward (payoff) can only be calculated at the end
    - Often the payoff is just 1=win or 0=lose
  - Mostly, resource intensive (except for the simplest games)

Why study games?

- There is a whole field of AI which is called “games theory”
  - There are a lot of algorithm, research, etc. in that field
- Why study games?
  - Fun; historically entertaining
  - Interesting subject of study because they are hard
  - Easy to represent and agents restricted to small number of actions
    - Board games are deterministic

Different types of games

- **Different types of games**

	deterministic	stochastic
Fully observable	chess, checkers, go, othello	backgammon monopoly
Partially observable	battleships, blind tictactoe	bridge, poker, scrabble

- **Recall**:
  - Fully observable (vs. partially observable): each agent "knows" the complete state/events of the environment at each point in time (including the previous moves from both players)
    - Some authors use the terms **perfect info / imperfect info**
  - Deterministic (vs. stochastic): The next state of the environment is completely determined by the current state, and the action executed by the agent who's playing
    - Some authors use the terms **deterministic / chance**

Different types of games

- **Zero-sum** games: participant A's gain or loss is exactly balanced by the losses/ gains of participant B
  - Examples: Chess, Rock Paper Scissors game
- **Non zero-sum** games:
  - Example: Monopoly
- **Constant-sum** games: the total payoff to all players is the same for every instance of the game
  - Example: poker

Different types of games

- Another important characteristic of games: Is it small or big?
  - Relative to the # of configurations to examine at each step (branching factor & depth of the search tree)
  - Only in small games can we do an exhaustive search for best move…
  - What's small or big? Difficult to say, depends on your computing capacity, but authors usually consider:
    - Tic-tac-toe: small
    - Most card games are big
    - Chess, Monopoly: very big
    - Go game: extremely big
- Related to two important definitions: very formally, in the computing field:
  - A "ply" is the name for **one turn by one player**
  - A "move" is a sequence of plies in which each player has one turn
  - But, in many games (including chess) a ply = a move
    - In this lecture, we will use indisctinctly « ply » and « move » (as most authors do)
- Example: chess
  - About 35 moves from any board configuration
  - A game may be 100 ply deep
  - So the full search tree will contain about 35^100 nodes which is about 10^154 nodes!!!

Majors assumptions in this lecture

- In this lecture, we will focus on
  - Fully observable, deterministic games
  - 2-player games
  - Zero-sum games

	deterministic	stochastic
Fully observable	chess, checkers, go, othello	backgammon monopoly
Partially observable	battleships, blind tictactoe	bridge, poker, scrabble

Deterministic games in practice

- **1994**, **checkers**: Chinook defeated the 40-year-long human world champion Marion Tinsley
  - Chinook used a precomputed endgame database defining perfect play for all positions involving <=8 pieces on the board
    - In total, 444 billion positions!!!
    - Basically, the computer “knew” in advance how to finish the game, whatever the actions from the human player!

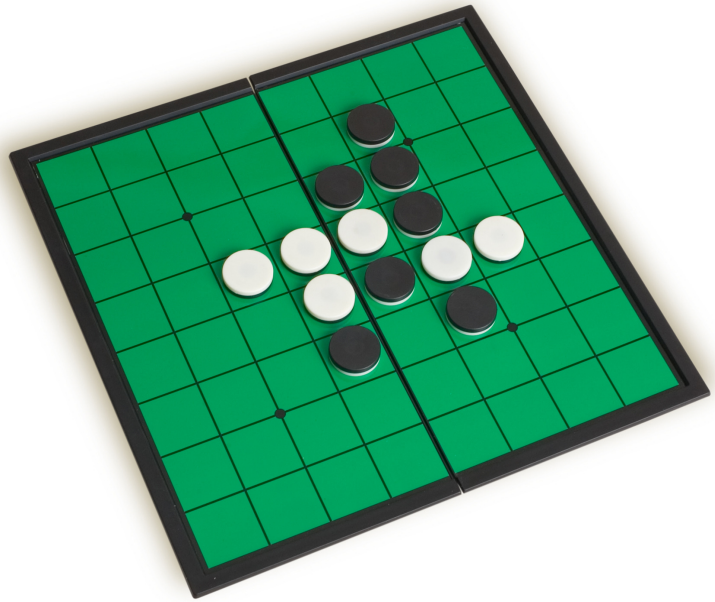
Deterministic games in practice

- **1997, Chess**: Deep Blue defeated human world champion Garry Kasparov in a six-game match.
  - Deep Blue searched 200 million positions per second, used very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply

May 1997
Deep Blue - Garry Kasparov
3.5 - 2.5

Deterministic games in practice

- **1997, Othello (*a.k.a.* reversi)*:***
  - The world champion lost a six-game series **against** Logistello
  - Since then, most human champions refuse to compete at Othello against computers, because computers are too good ☺



Deterministic games in practice

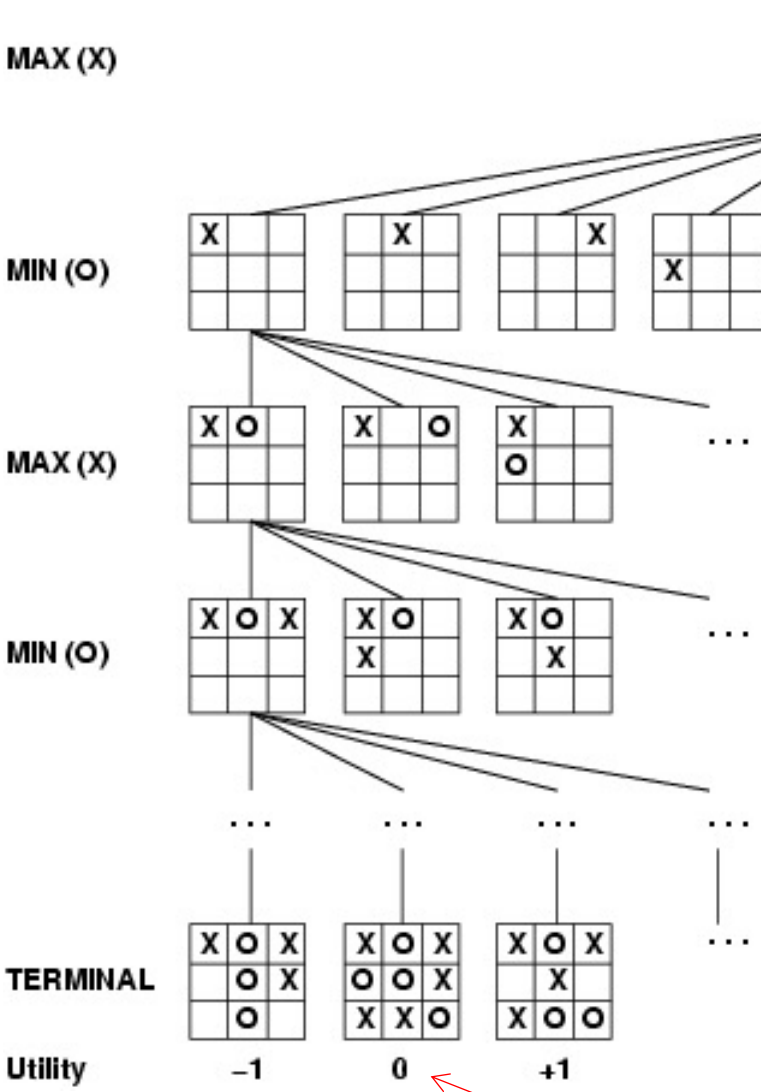
- **Go game: much more complex game** *(b > 300)*
  - For a long time, human champions refused to compete against computers, who were too bad!
  - But... In **2015**, AlphaGo wins over the Europe champion Fan Hui
    - 5 games vs 0
  - In **2016**, AlphaGo defeats Lee Sedol, one of the best world players

2-player game problem formulation

Problem formulation: minimax

- A game with 2 players: **MAX** and **MIN**
- **MAX** moves first then **MIN**, etc (turn-taking), until the game is over
- Winner gets award (+1), looser gets penalty (-1)
- This problem can be defined as a search problem with:
  - initial state: initial position(s) (fixed in most board games)
  - player: player to move (*a.k.a* to take an action)
  - successor function: a list of legal (move, state) pairs
  - goal test: is the game over? terminal states
  - utility function: gives a numeric value for the terminal states
    - E.g. win (+1), loose (-1) and draw (0) in tic-tac-toe
    - In minimax, we use a "utility function" rather than an "evaluation function"
- **Game tree** = initial state + legal moves
  - MAX uses the search tree to determine its next move
  - Perfect for playing deterministic games

Minimax: example with tic-tac-toe

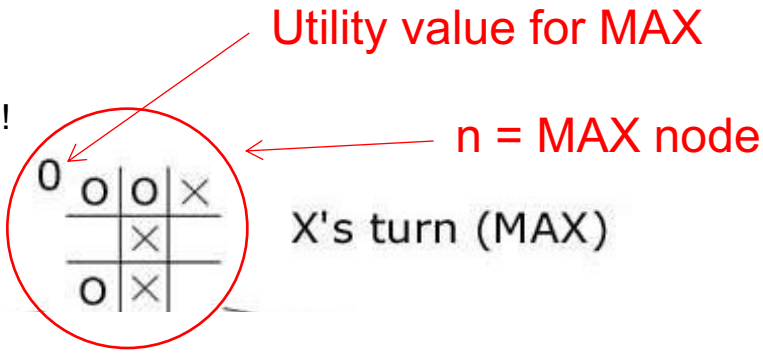


- From among the moves available to MAX, MAX must choose the best one
  - Taking into account possible MIN's moves
- The best move is determined by a search using the **MiniMax** strategy

Utility values of the terminal states

Minimax strategy

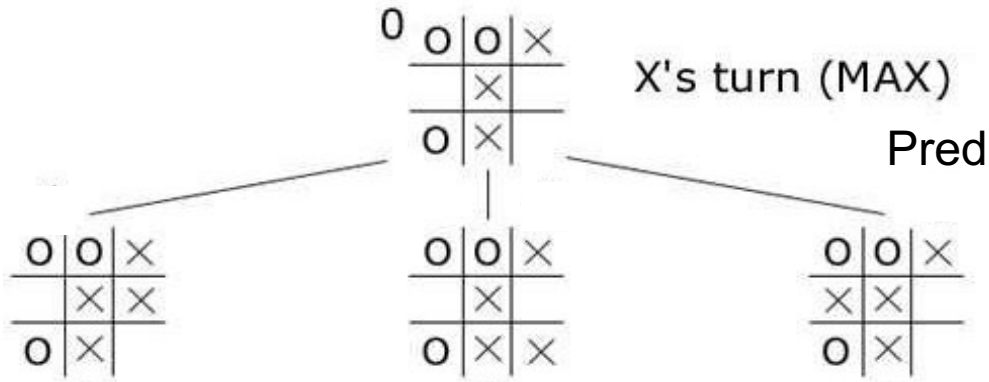
- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



Example by Yosen Lin

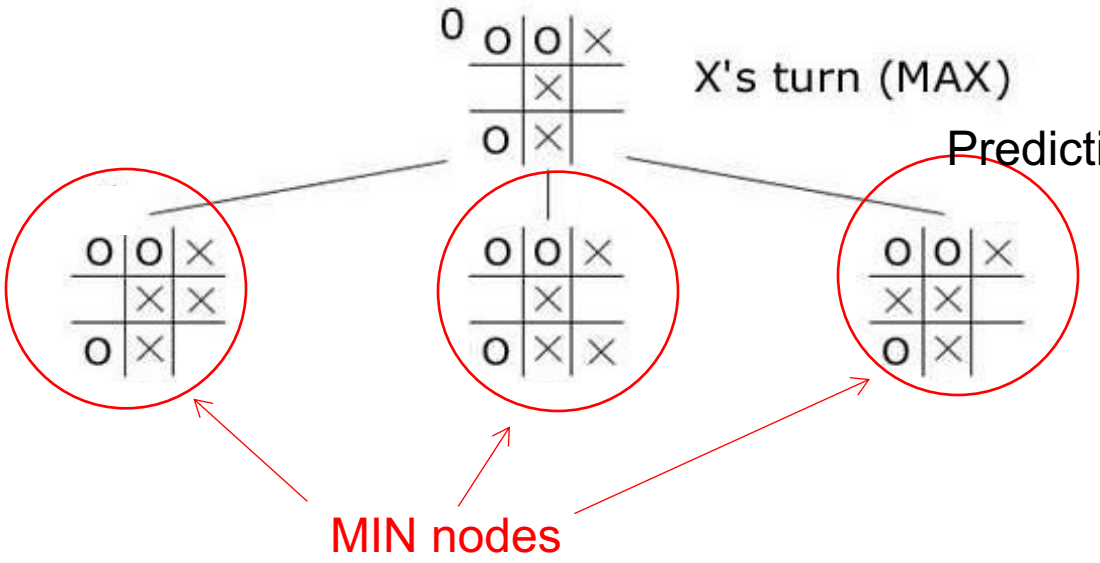
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



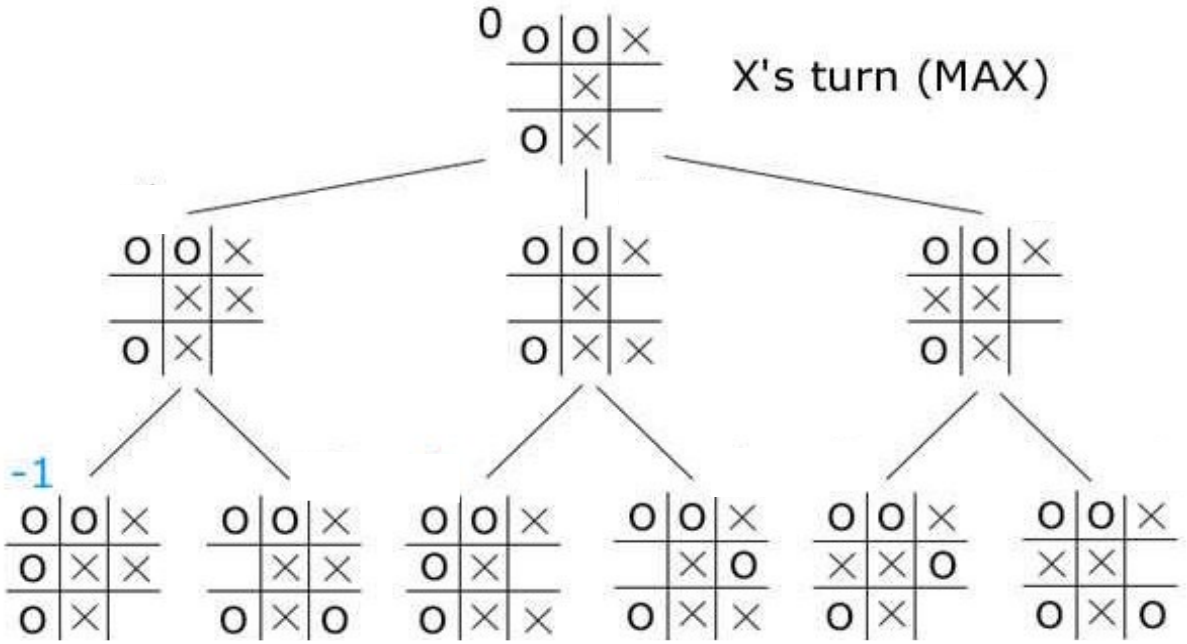
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



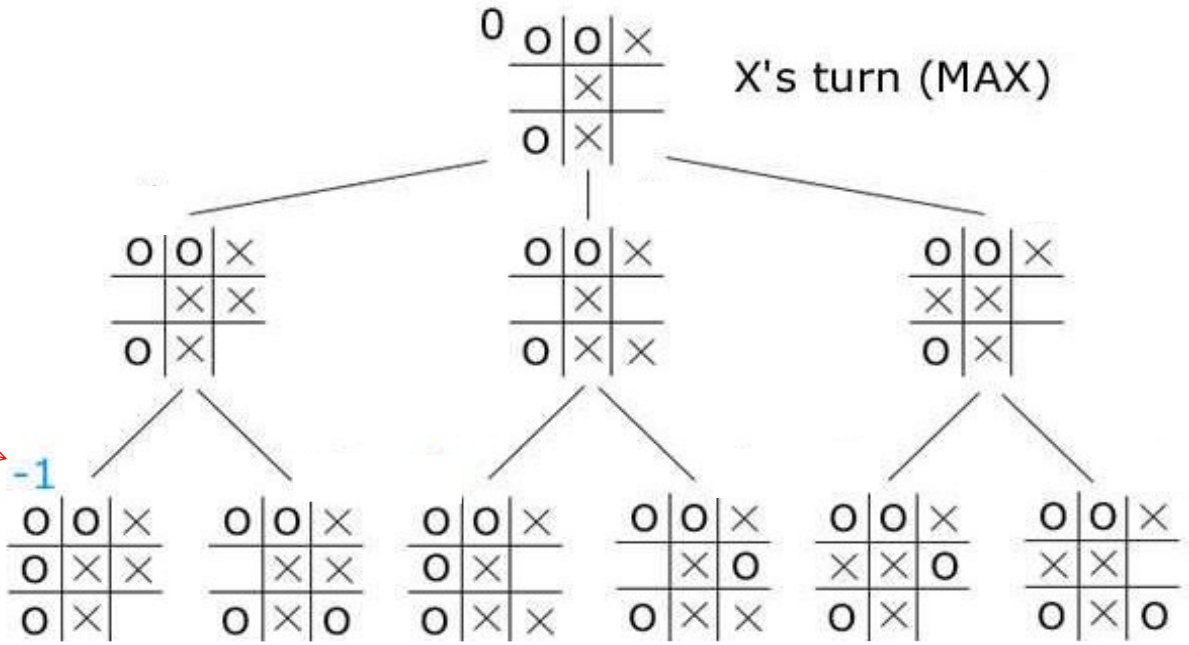
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



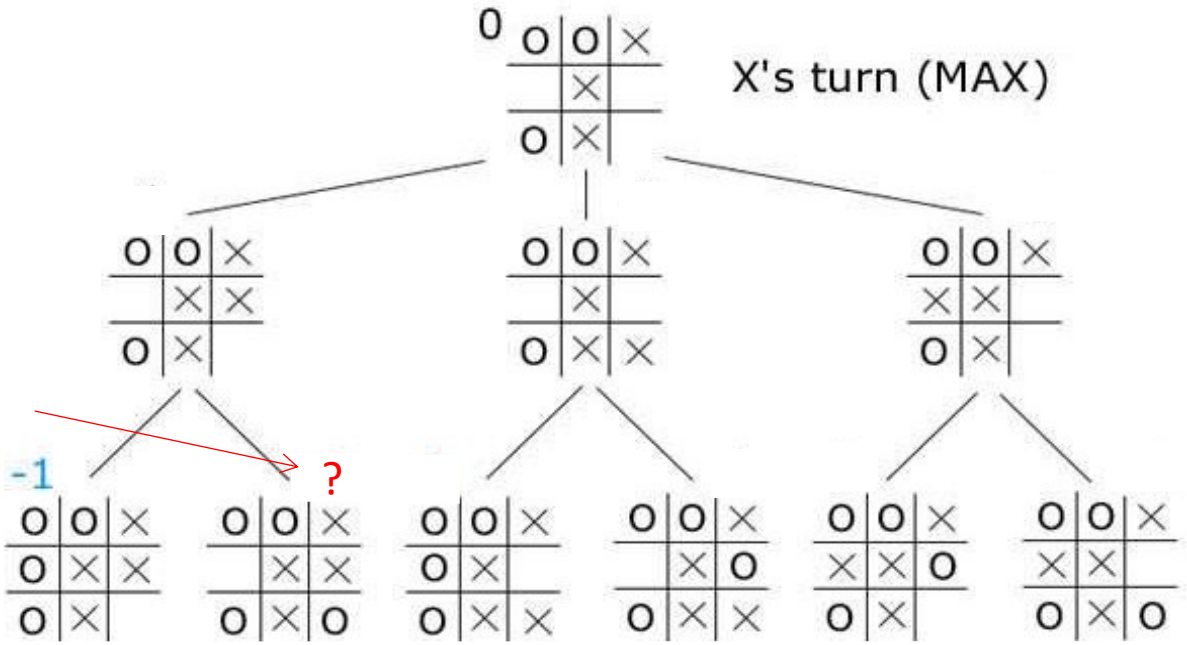
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



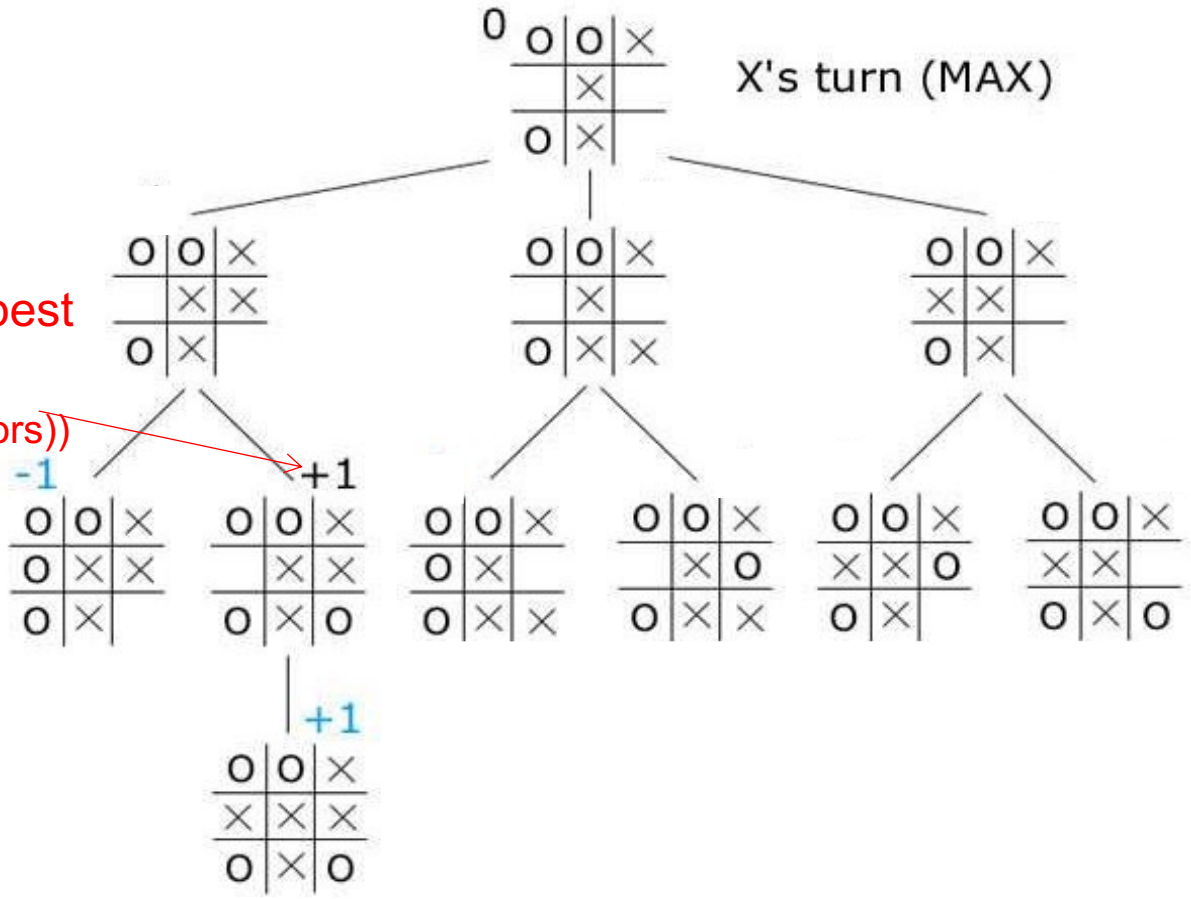
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!

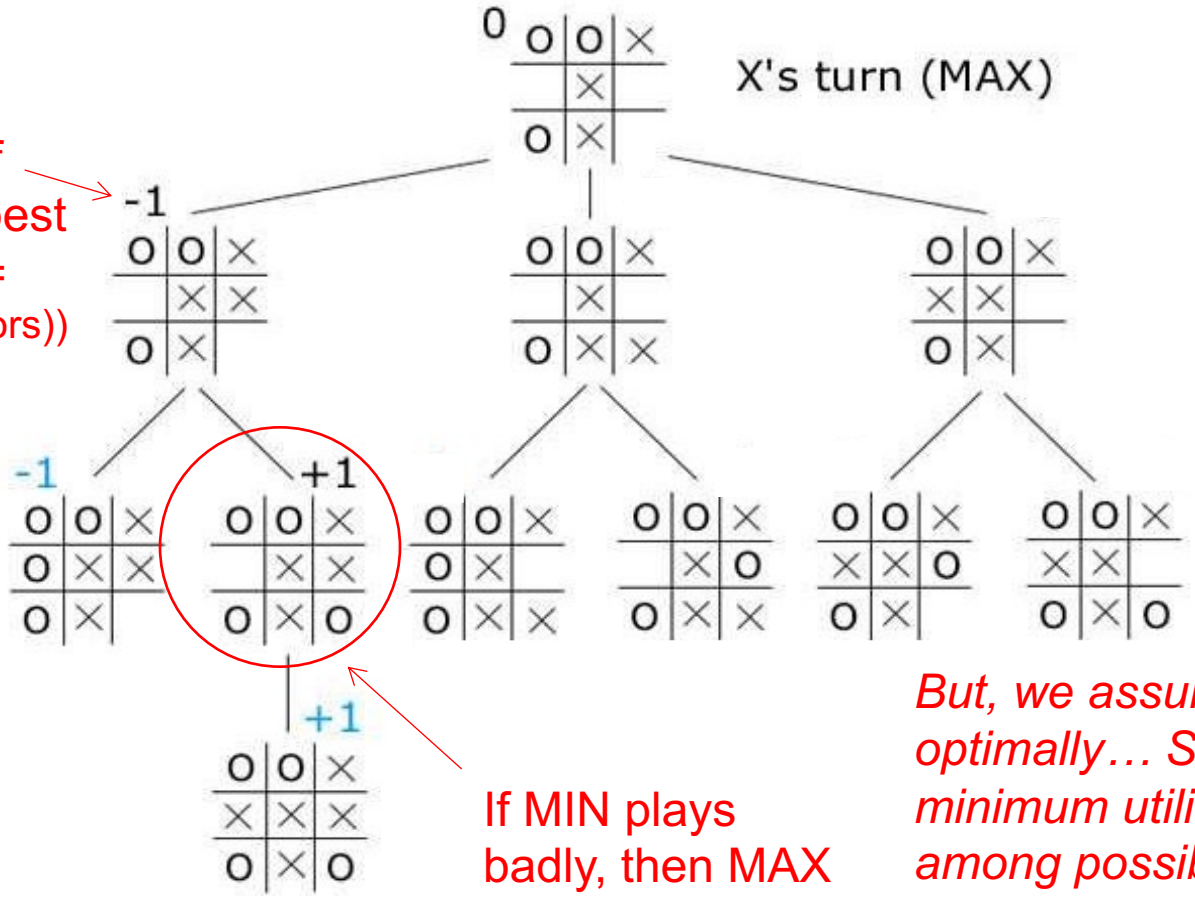


Utility for MAX if MAX makes its best move

max(utility(successors))

Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



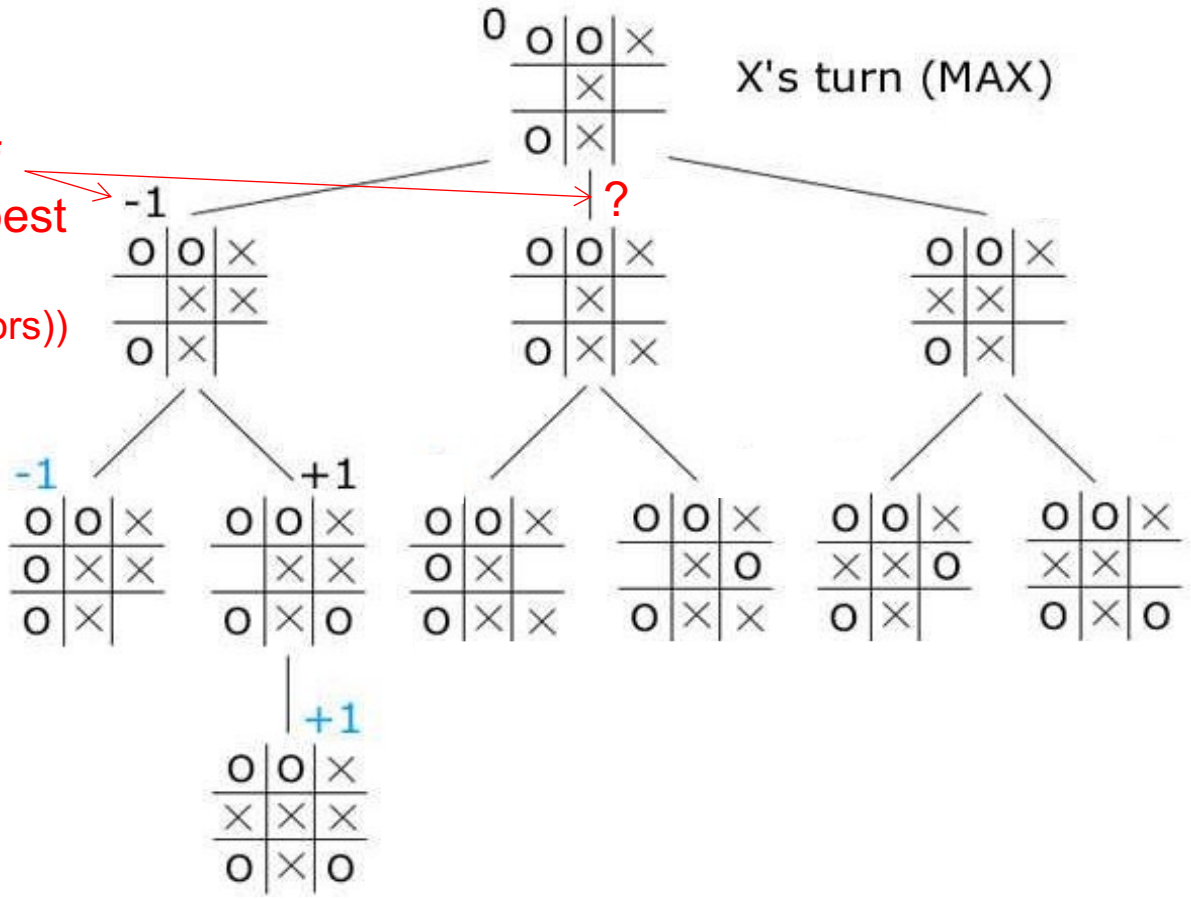
Utility for **MAX** if MIN makes its best possible move = min(utility(successors))

If MIN plays badly, then MAX can win!!!

*But, we assume MIN plays optimally… So we take the minimum utility value for **MAX** among possible MIN's moves consequences*

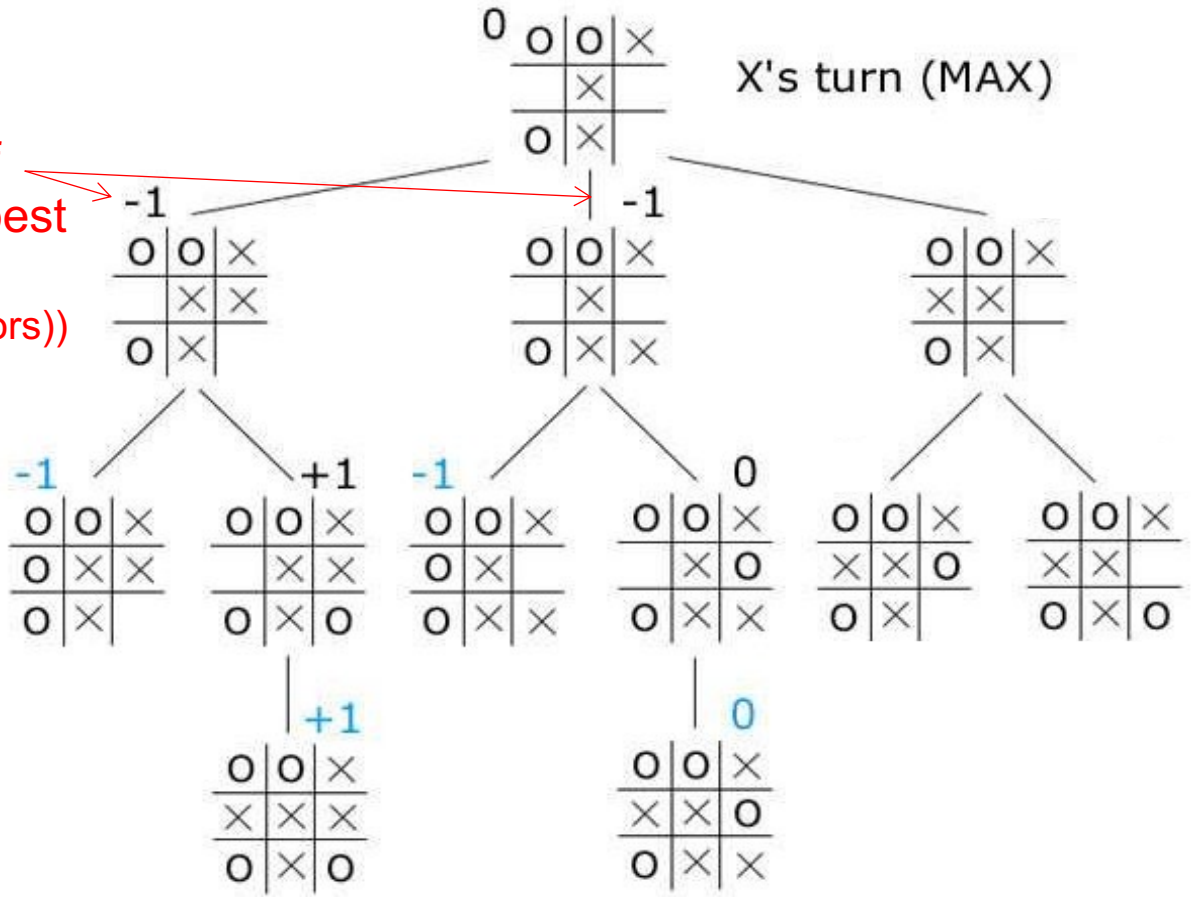
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



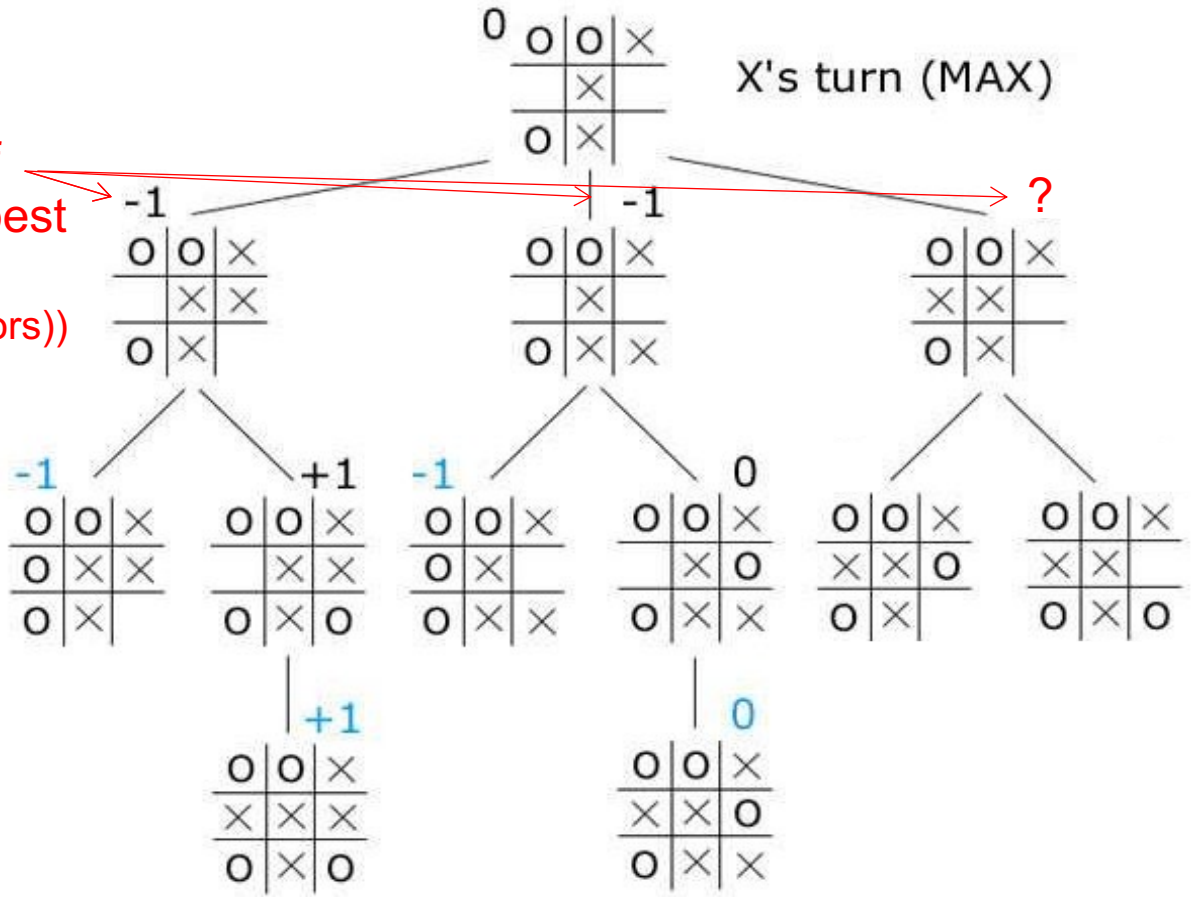
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



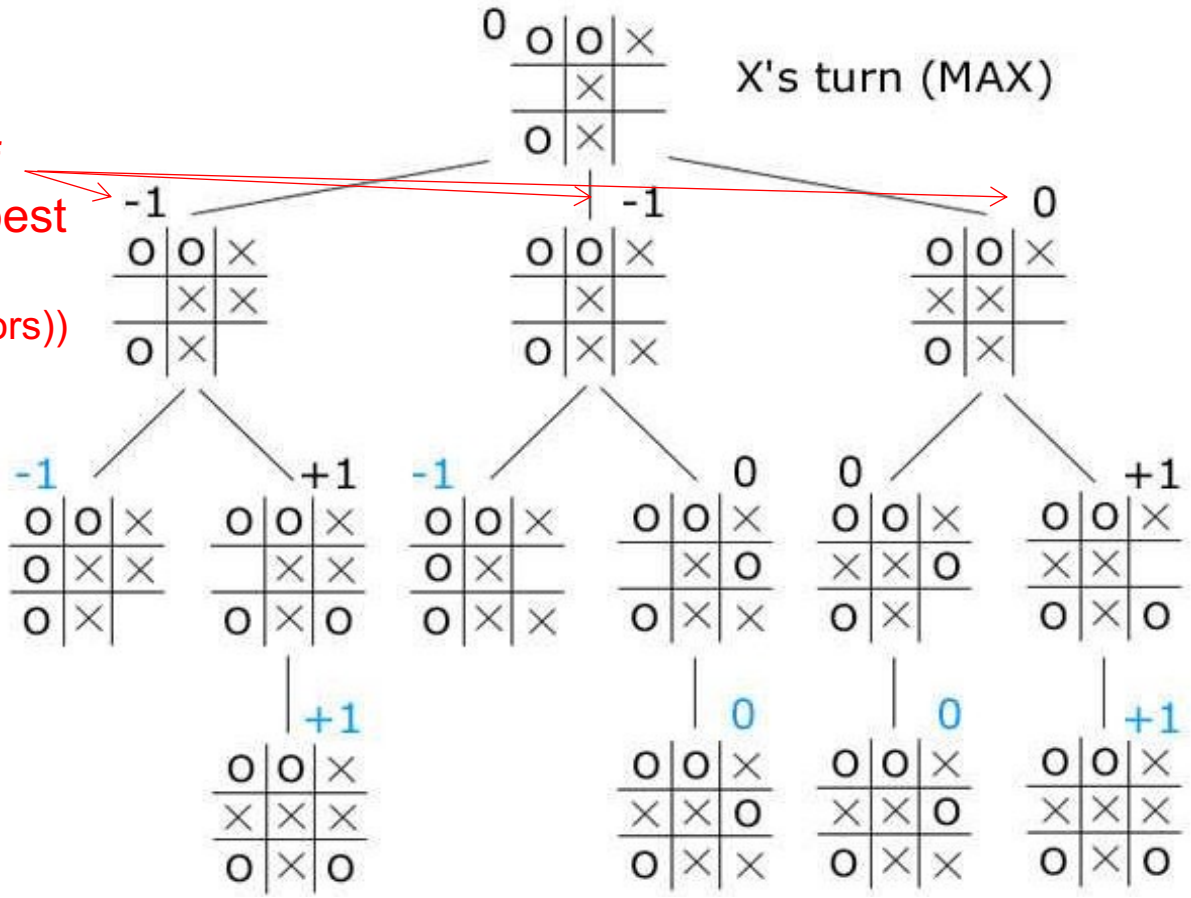
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



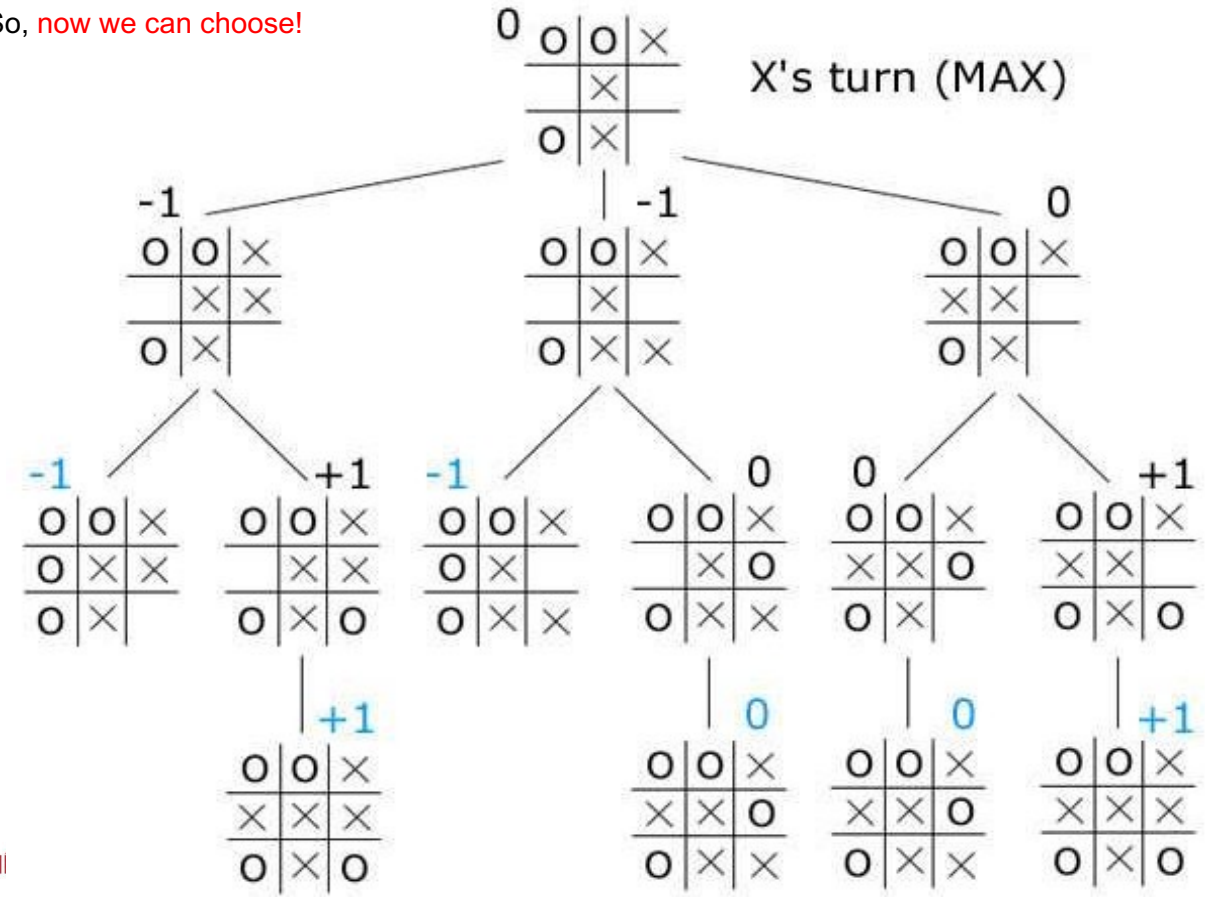
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!



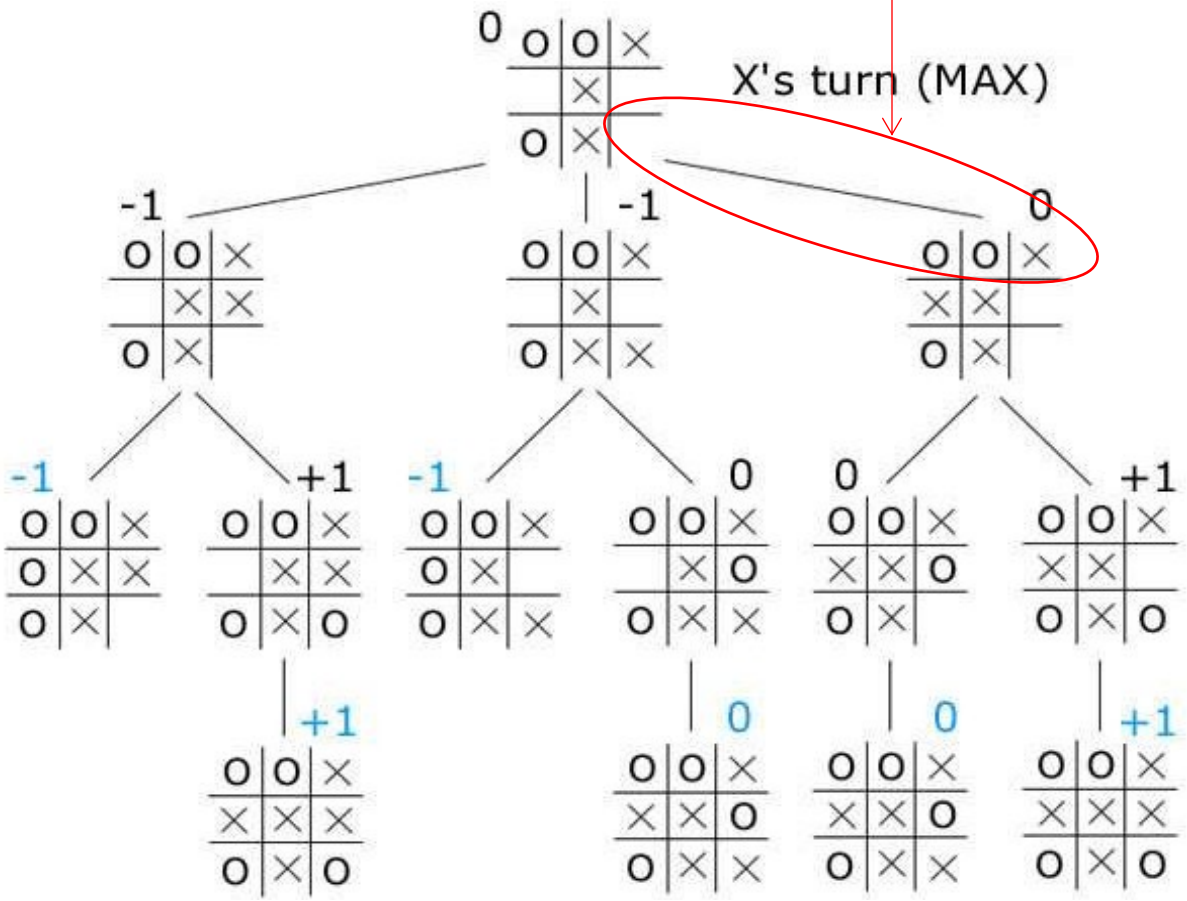
Minimax strategy

- **Important**: In Minimax strategy, both players are **supposed to play perfectly**!
- **Example:** partial tic-tac-toe tree (game already started)
  - **It is MAX's move:**
    - Which action should it choose?
    - Depends on MIN&MAXs next actions!
      - Now we know the possible final utilities
      - So, now we can choose!



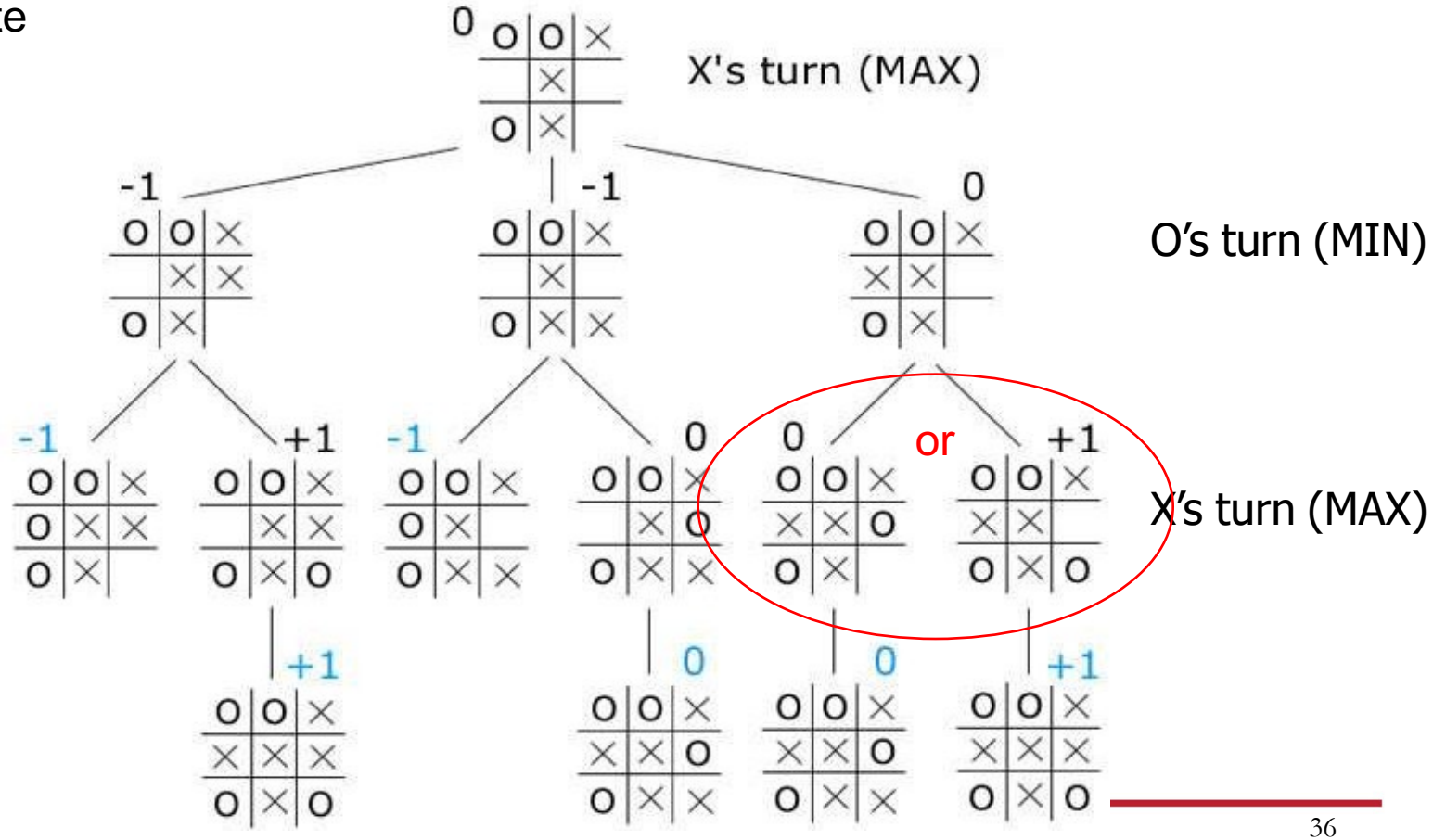
Minimax strategy

- Objective of Minimax strategy: choose this action
  - = the only one that guarantees either a draw or a win (if MIN & MAX are both playing “perfectly”)



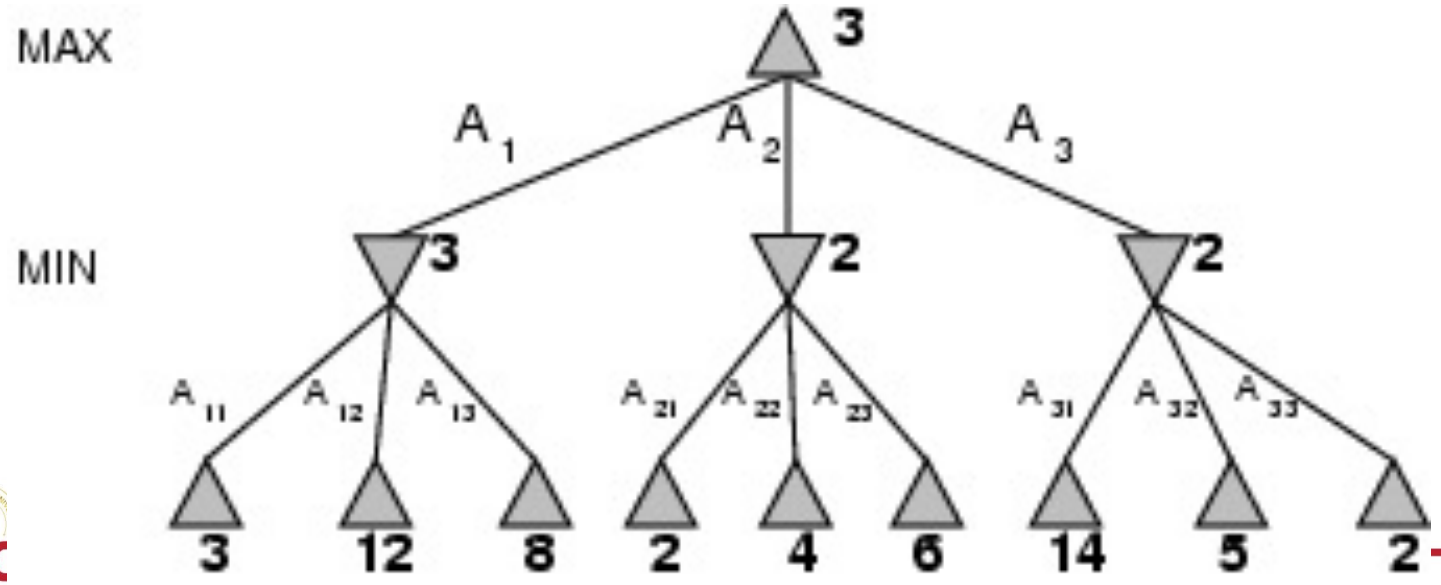
Minimax strategy

- Now we have to:
  - apply minimax strategy to the next initial state (depending on MIN's action)
  - take the best action for MAX by applying minimax strategy
  - iterate



Minimax: exercise 1

- **Exercise**: A two-ply game tree
  - The △ nodes are "MAX nodes," in which it is MAX's turn to move
  - The ▽ nodes are "MIN nodes."
  - The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values
    - The exercise is very simple, because the utility values for MAX are given to you!
  - **Questions**:
    - What is MAX's best move at the root?
    - What is MIN's best move during the next move (assuming MAX did the best move)?



Minimax algorithm

If you still did not understand minimax

- Here is a very simple example for understanding the principle
  - https://www.youtube.com/watch?v=KU9Ch59-4vw
  - In this video,
    - MAX = R; MIN=B
    - A RED node is a node leading to R winning (if both players play perfectly)
    - A BLUE node is a node leading to B winning (if both players play perfectly)
    - A GREY node leads to a draw between R and B (if both players play perfectly)

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

Properties of minimax

- In 2-player, deterministic, fully observable, zero-sum games, the minimax algorithm can select optimal moves by a depth-first search in the game tree

- Complete? Yes (if the search tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)
  - -> do not keep all expanded nodes in main memory

- For chess, b ≈ 35, d ≈100 for "reasonable" games
  → optimal solution untractable (if it is played with a clock)
  - And it's even worse for the Go game!!!

→ Most often, for big games, we use a **depth-limited** version of minimax algorithm (depth limit l= number of plies to study ahead)

Minimax: exercise 2

- Prove that: for every game tree, the utility obtained by MAX using minimax decisions against a suboptimal MIN will always be >= to the utility obtained playing against an optimal MIN
- Answer:

Problem of minimax search

- Number of games states is exponential to the number of moves.
  - Solution: Do not examine every node

$\Rightarrow$ α-β pruning:

- Remove branches that **will not influence** the final decision
- In short, try to prune the node "smartly"

α-β pruning

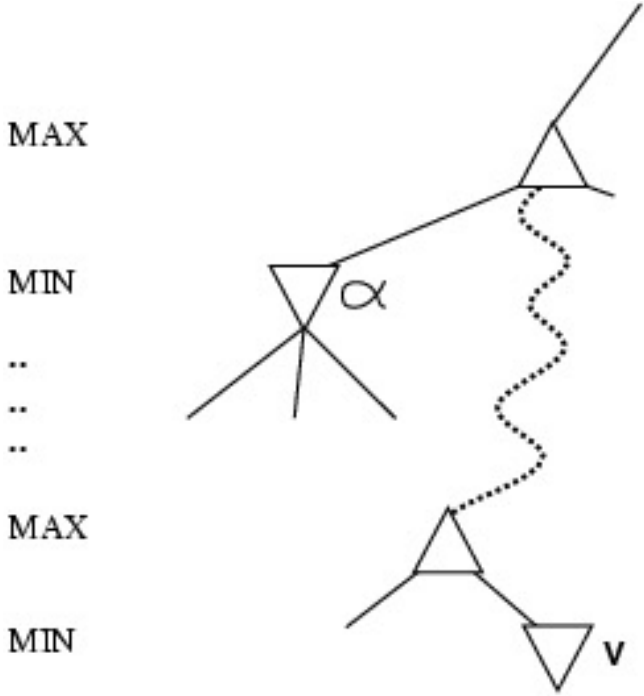
Definition

α-β pruning example (two-ply game tree)

- https://www.youtube.com/watch?v=l-hh51ncgDI
- In this video,
  - WHITE = MAX and BLACK = MIN
  - The values shown are the utility functions for WHITE (MAX)
    - BLACK wants to minimize it ; WHITE wants to minimize it
  - The whole video is only about **one** TURN of the game
    - Where MAX (WHITE) needs to decide about the best move

Why is it called α-β?

- The algorithm keeps track of two values when exploring nodes:
  - α (best utility value so far for MAX) – assuming MIN plays perfectly
    - MAX wants to **maximize** the utility function
  - β (best utility value so far for MIN) – assuming MAX plays perfectly
    - MIN wants to **minimize** the utility function
- So, if it's MAX's turn :
  - if MAX can infer that the utility value *v* of a node *n'* is < α, then MAX will prune the branch with root *n'*
- Similar for MIN's turn, with β



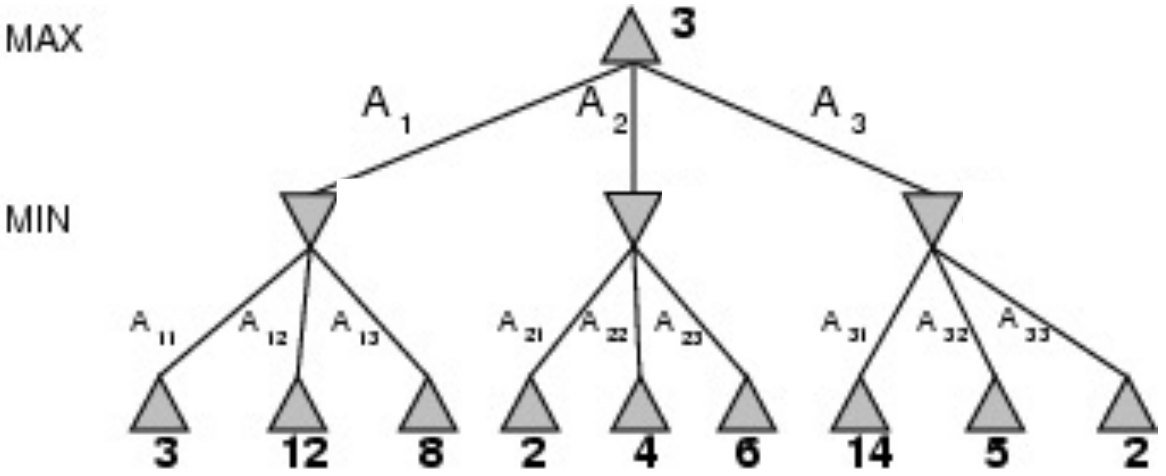
α-β search algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) 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(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

α-β pruning: exercise 3 (two-ply game tree)

- On this problem, by using α-β pruning, we can identify the minimax decision without never evaluating two of the leaf nodes…
- Which nodes do not need to be examined?
  - Answer :



α-β pruning

Properties and limitations

Properties of α-β

- Pruning does not affect the final result
- Entire sub-trees can be pruned
  - See the last example in the video that I put on slide 44
- Good move ordering improves effectiveness of pruning. With "perfect ordering"
  - In the exercise 3, "perfect ordering" = ascending order among siblings
  - time complexity = $O(b^{m/2})$

    →doubles depth of search, for a given time, compared to minimax

    ”α-β pruning can look twice as far as minimax in the same amount of time”
  - Effective branching b* factor becomes sqrt(b) !!
    - For instance, for chess, roughly 6 instead of roughly 35
- α-β pruning is a simple example of meta-reasoning

Limitations of α-β

- Even when using α-β pruning, repeated states are still possible
- Actually, repeated states are very frequent in games because of transpositions
  - different permutations of the move sequence lead to the same state (position)
    - *e.g.*, [a1, b1, a2, b2] vs. [a1, b2, a2, b1]
- It's worthwhile to store the evaluation of this position in a **hash table** the first time it is encountered
  - Such hash tables are called transposition tables
  - Similar to the "closed nodes" in graph search
- Tradeoff:
  - Transposition table can be too big
  - Which to keep and which to discard?

α-β pruning

How to twist it to make imperfect, real-time decisions

Imperfect, real-time decisions

- Minimax generates the entire game search space
  - α-β prunes large part of it
  - but still needs to search all the way to terminal states

- But, in many games, moves must be made in limited amount of time
  - Suppose we have 100 secs, and our computational power is to explore 10^4 nodes/sec
    → 10^6 nodes per move

- Standard solutions (Shannon, 1950):
  - turning non-terminal nodes into terminal leaves, by using either:
    1. CUTOFF-TEST: replaces TERMINAL-TEST in the α-β algo.
       - *e.g.:* depth limit
    2. Heuristic evaluation function EVAL: estimated desirability or utility of position
       - EVAL replace the utility function in α-β algorithm

1. Cut-off search

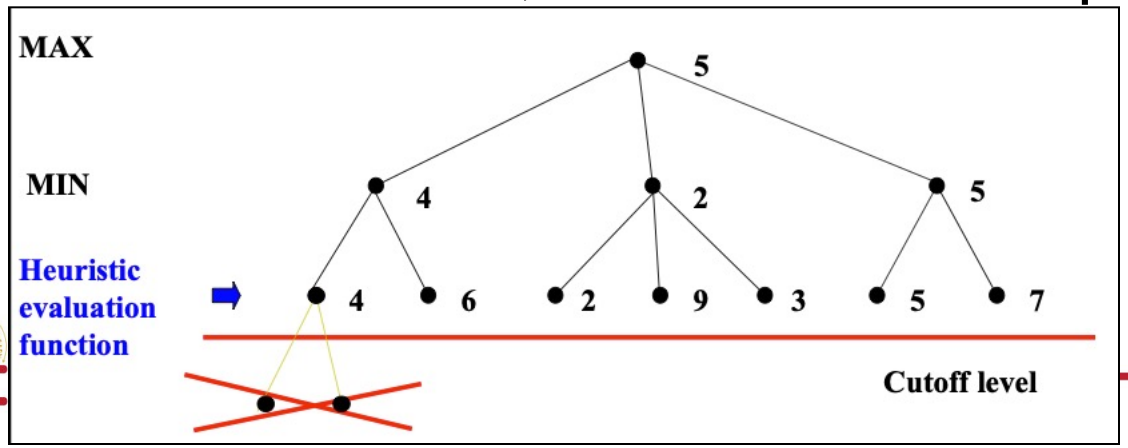
- In the α-β algorithm, replace:

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

  with: **if** CUTOFF-TEST(*state,depth*) **then return** EVAL(*state*)

- Introduces a fixed-depth limit *depth*
  - Is selected so that the amount of time will not exceed what the rules of the game allow

- When cut-off occurs, the evaluation is performed

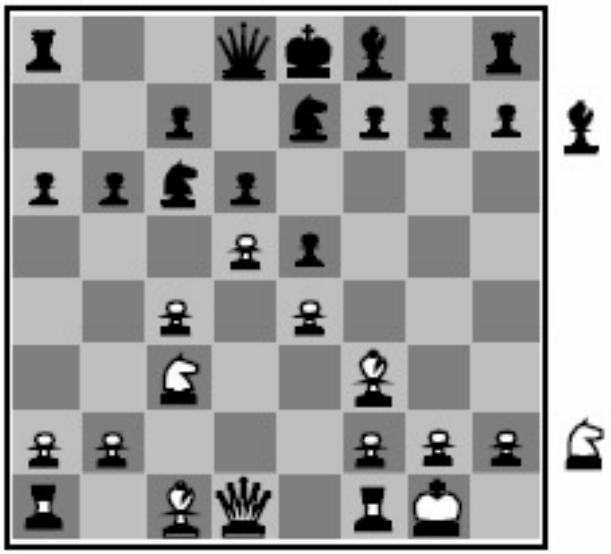


2. Heuristic evaluation (EVAL)

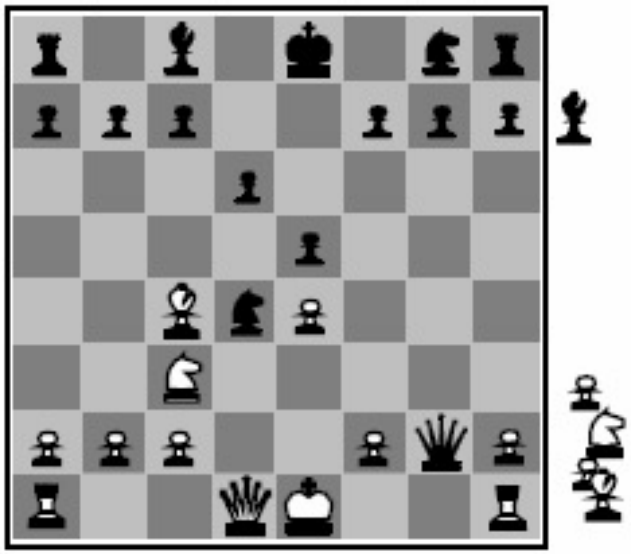
- Idea: produce an estimate of the expected utility of the game from a given position
- Performance depends on quality of EVAL
- Requirements:
  - EVAL should **order** terminal nodes in the same way as UTILITY
    - *E.g.* if the current node would be the second highest using UTILITY, then it should also be the second highest using EVAL
    - For alpha-beta pruning to work in the same way
  - Computation of EVAL should not take too long
  - For non-terminal states, the EVAL should be strongly correlated with the actual chance of winning
- Only useful for quiescent (no wild swings in value in near future) states

2. Heuristic evaluation (EVAL): example

- Example: even if we don't have the computational power to estimate all possible future actions, we can say that:



- Here, the EVAL value for whites should be slightly better than for blacks



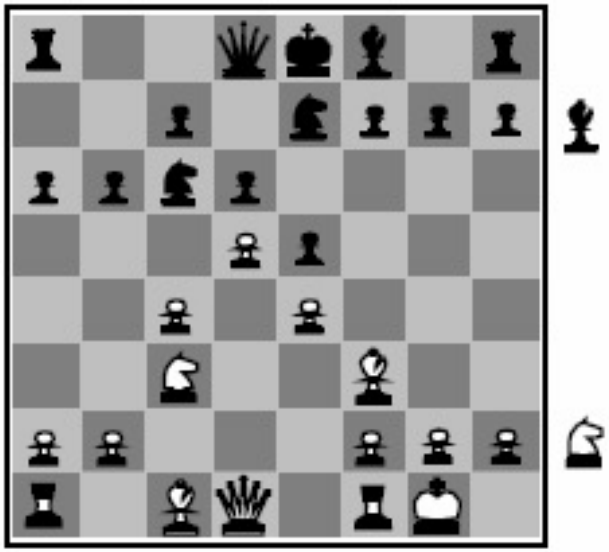
- Here, the EVAL value for whites should be much better than for blacks

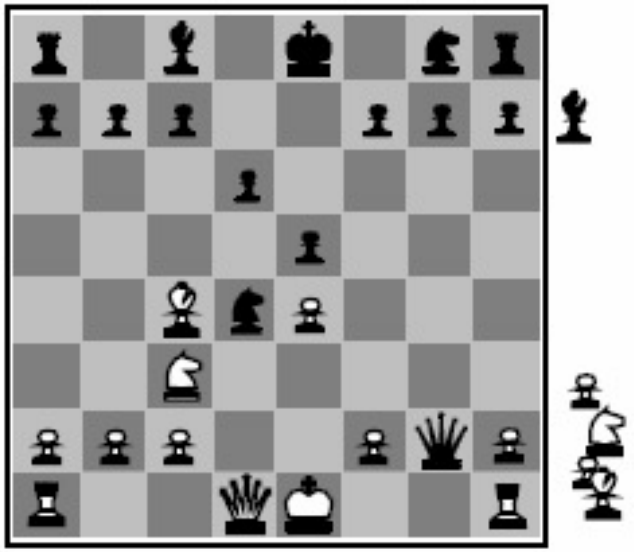
2. Heuristic evaluation (EVAL): example

- For chess, a typical example of EVAL function is a linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- *e.g.*, if we play WHITE, $w_1 = 9$ with s the current state and
  - $f_1(s)$ = (number of white queens) – (number of black queens), etc.
  - $f_2(s)$ = (number of white towers) – (number of black towers), etc.





A few words about chess complexity

- Let's say we play with a 3-min clock
- Let's say a regular PC can search 200 millions nodes / 3min.
- Branching factor: ~35
- Does it work in practice?
  - 4-ply ≈ human novice → hopeless chess player
  - 5-ply ≈ regular PC with minimax (as 35^5 ~ 50 millions nodes / mn)
  - 8-ply ≈ human master, regular PC using alpha-beta pruning
  - 12-ply ≈ Deep Blue, human grandmasters, *e.g.* Kasparov
- To reach grandmaster level, the machine needs:
  - An *extensively* tuned heuristic evaluation function
  - A large database of ***optimal opening and ending*** of the game
    - Pre-computed, possibly with their true utility functions

Some insights on non-deterministic games / games with imperfect info / games with imperfect opponent

Non-deterministic games

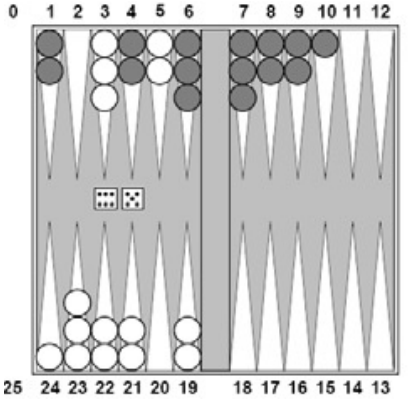
Nondeterministic games

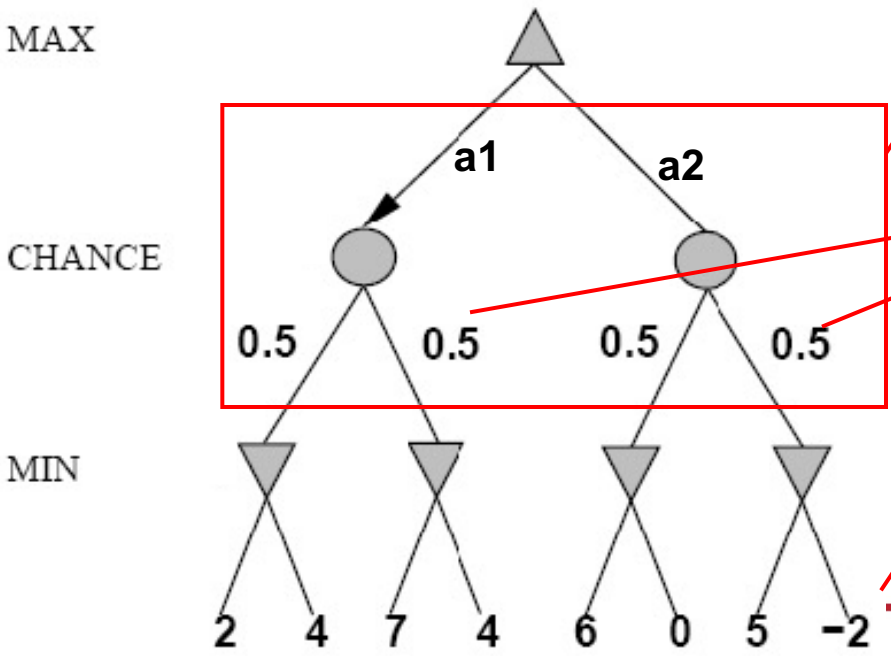
- Let's say that now I have enough computational power to compute the utility values of the leaves, but **chance** has an effect
  - Either because of the game in itself: roll a dice, pick a card…
  - … Or because the opponent is actually not playing perfectly

Nondeterministic games

- Example of a game with coin-flipping:
  - It is MAX's turn: MAX needs to choose an action (a1 or a2), then flip a coin which will determine the next state, and then it is MIN's move
  - N.B. I took the example of coin-flipping because it is easier (even if less realistic)

Simplified version of backgammon





chance nodes

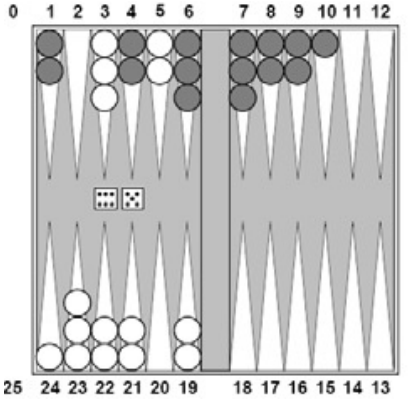
Probabilities for head / tail in coin flipping

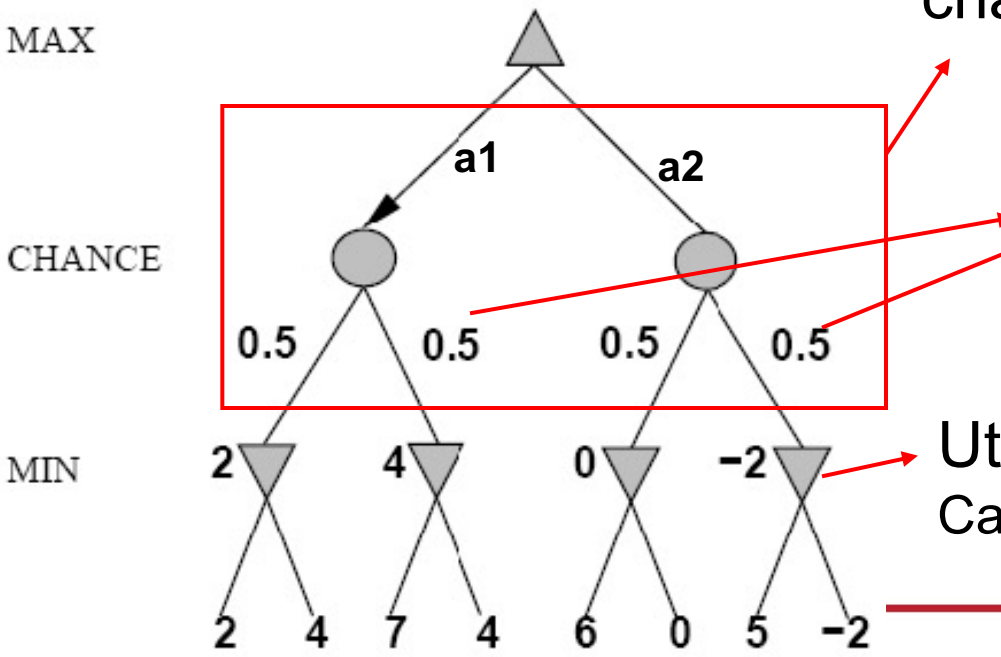
Utilities for MAX

Nondeterministic games

- Example of a game with coin-flipping:
  - It is MAX's turn: MAX needs to choose an action (a1 or a2), then flip a coin which will determine the next state, and then it is MIN's move
  - N.B. I took the example of coin-flipping because it is easier (even if less realistic)

Simplified version of backgammon





chance nodes

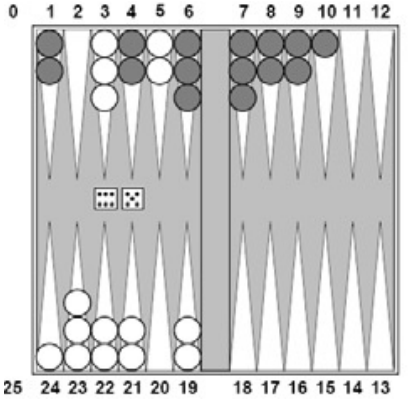
Probabilities for head / tail in coin flipping

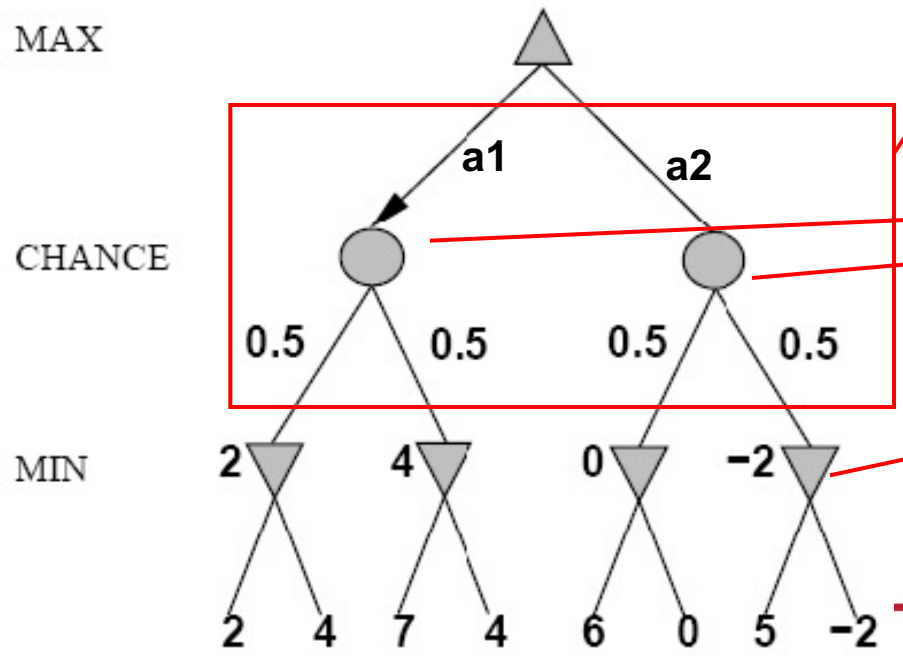
Utilities of nodes at the MIN level
Calculated as in minimax

Nondeterministic games

- Example of a game with coin-flipping:
  - It is MAX's turn: MAX needs to choose an action (a1 or a2), then flip a coin which will determine the next state, and then it is MIN's move
  - N.B. I took the example of coin-flipping because it is easier (even if less realistic)

Simplified version of backgammon





chance nodes

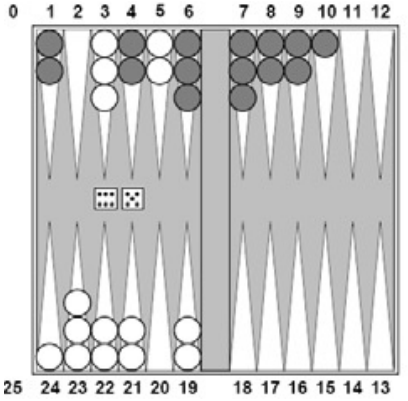
How to give a utility value to the chance nodes, so that MAX can choose?
-> expected minimax algorithm!

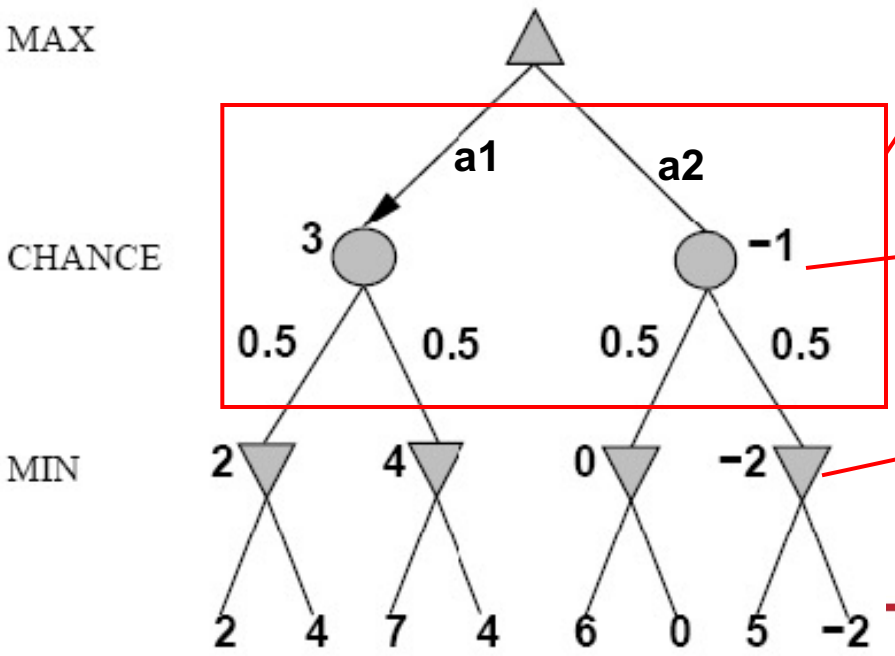
Utilities of nodes at the MIN level
Calculated as in minimax

Nondeterministic games

- Example of a game with coin-flipping:
  - It is MAX's turn: MAX needs to choose an action (a1 or a2), then flip a coin which will determine the next state, and then it is MIN's move
  - N.B. I took the example of coin-flipping because it is easier (even if less realistic)

Simplified version of backgammon





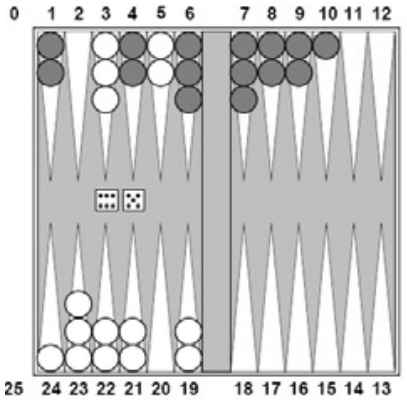
chance nodes

Output of expected minimax:
CHANCE gets the (weighted) average of the possible outcomes' utilities

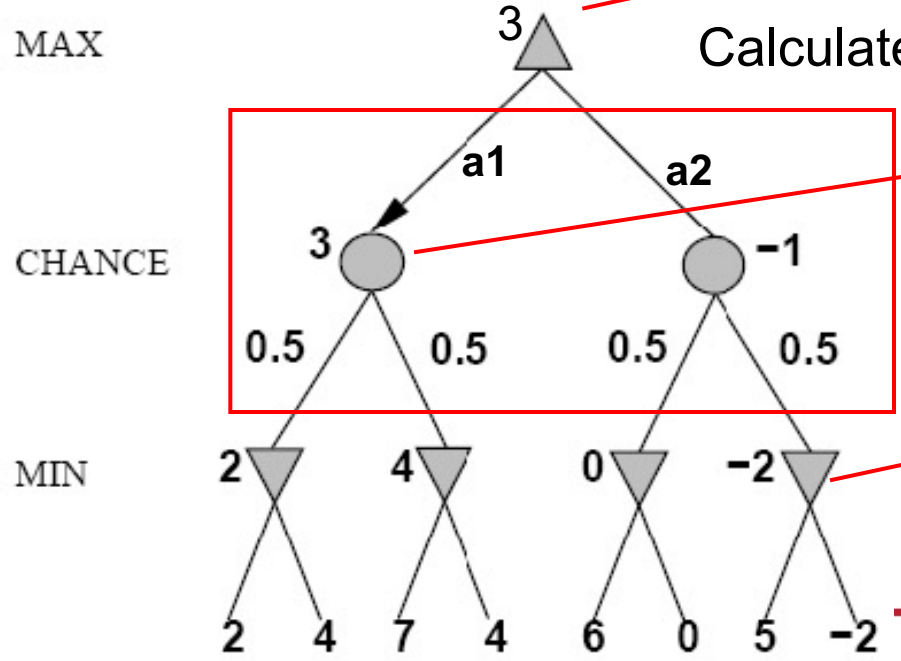
Utilities of nodes at the MIN level
Calculated as in minimax

Nondeterministic games

Simplified version of backgammon



- Example of a game with coin-flipping:
  - It is MAX's turn: MAX needs to choose an action (a1 or a2), then flip a coin which will determine the next state, and then it is MIN's move
  - N.B. I took the example of coin-flipping because it is easier (even if less realistic)



Utilities of nodes at the MAX level
Calculated as in minimax

MAX will choose this node

Utilities of nodes at the MIN level
Calculated as in minimax

Expected minimax value

```
...
if state is a MAX node then
        return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a MIN node then
        return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a chance node then
        return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
```

EXPECTED-MINIMAX-VALUE(*n*)=

UTILITY(*n*) If *n* is a terminal

$\max_{s \in successors(n)}$ EXPECTEDMINIMAX(*s*) If *n* is a MAX node

$\min_{s \in successors(n)}$ EXPECTEDMINIMAX(*s*) If *n* is a MIN node

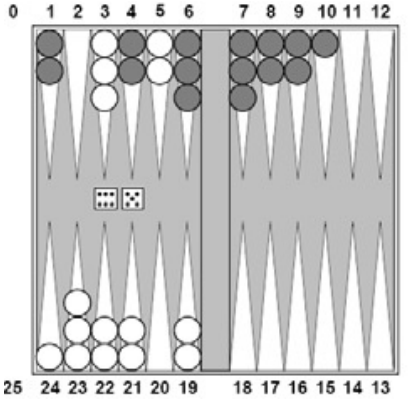
$\Sigma_{s \in successors(n)}$ *P*(*s*) .EXPECTEDMINIMAX(*s*) If *n* is a chance node

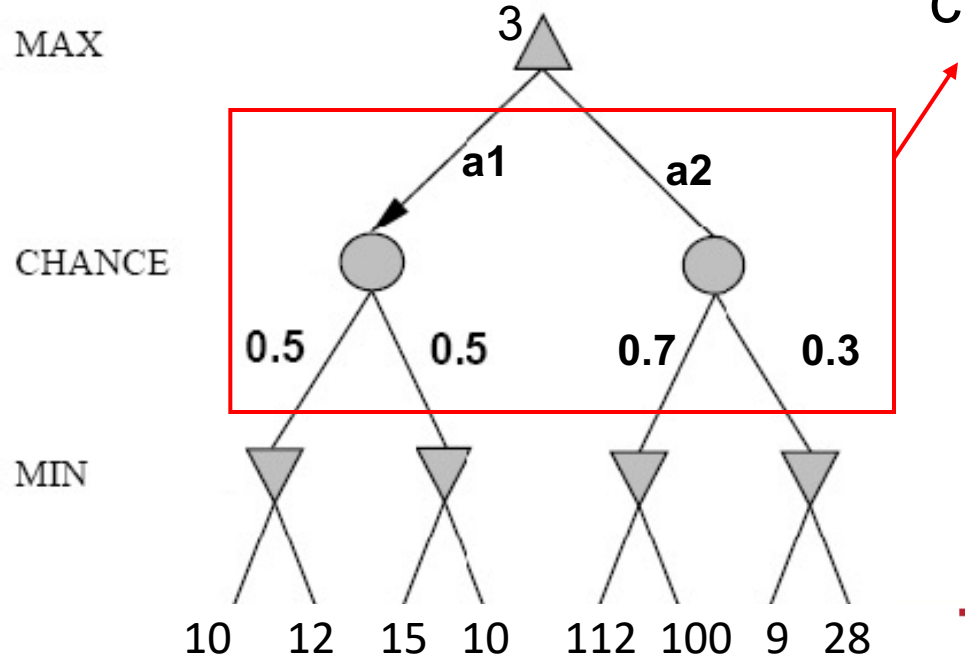
P(s) is the probability of occurrence of state s

Nondeterministic games

- Example of another game instance with coin-flipping:
  - Should the player MAX choose action a1 or a2?

Simplified version of backgammon



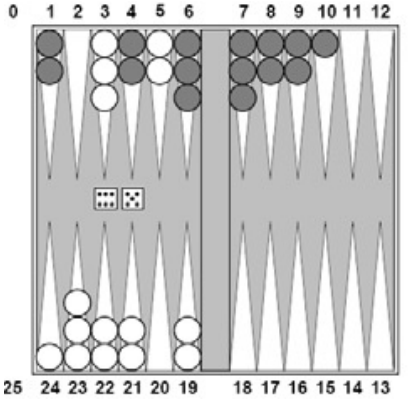


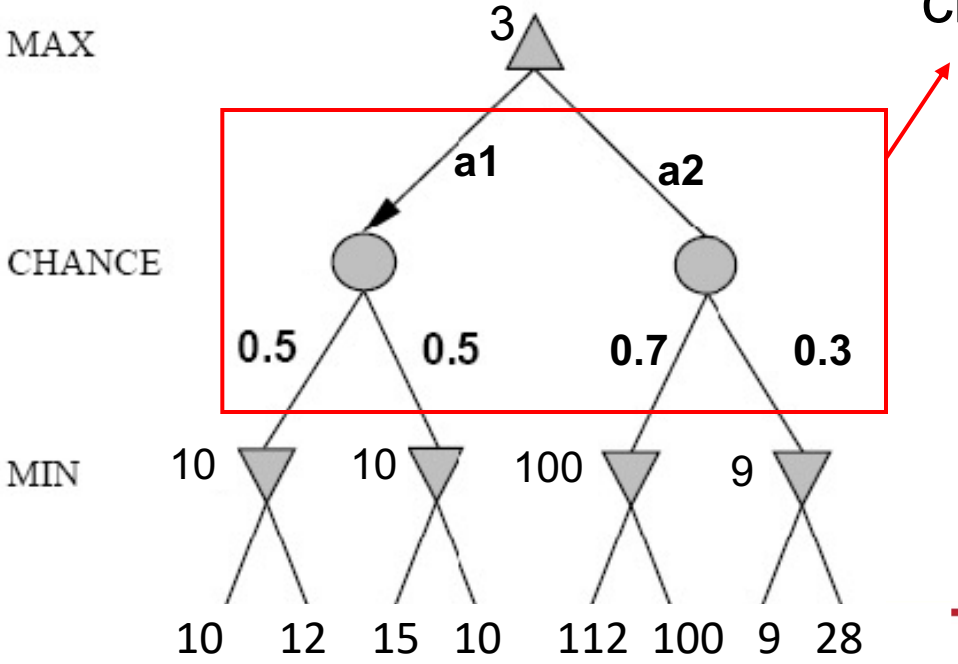
chance nodes

Nondeterministic games

- Example of another game instance with coin-flipping:
  - Should the player MAX choose action a1 or a2?

Simplified version of backgammon

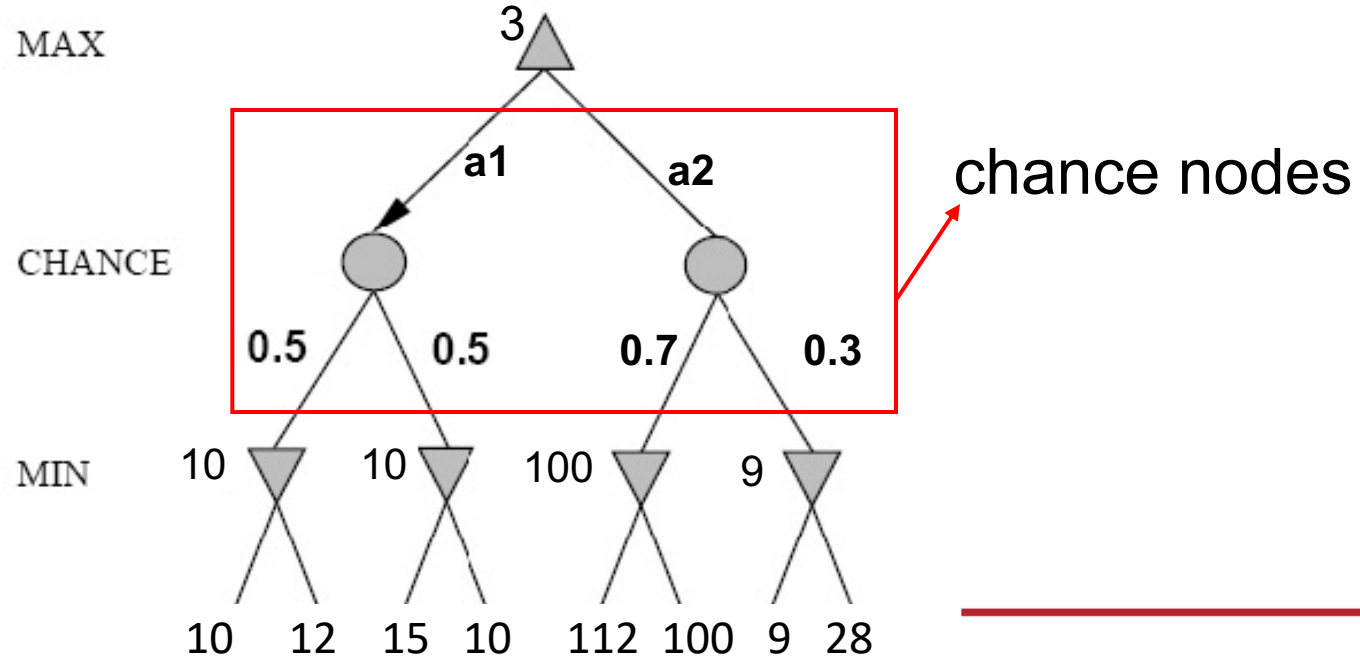




chance nodes

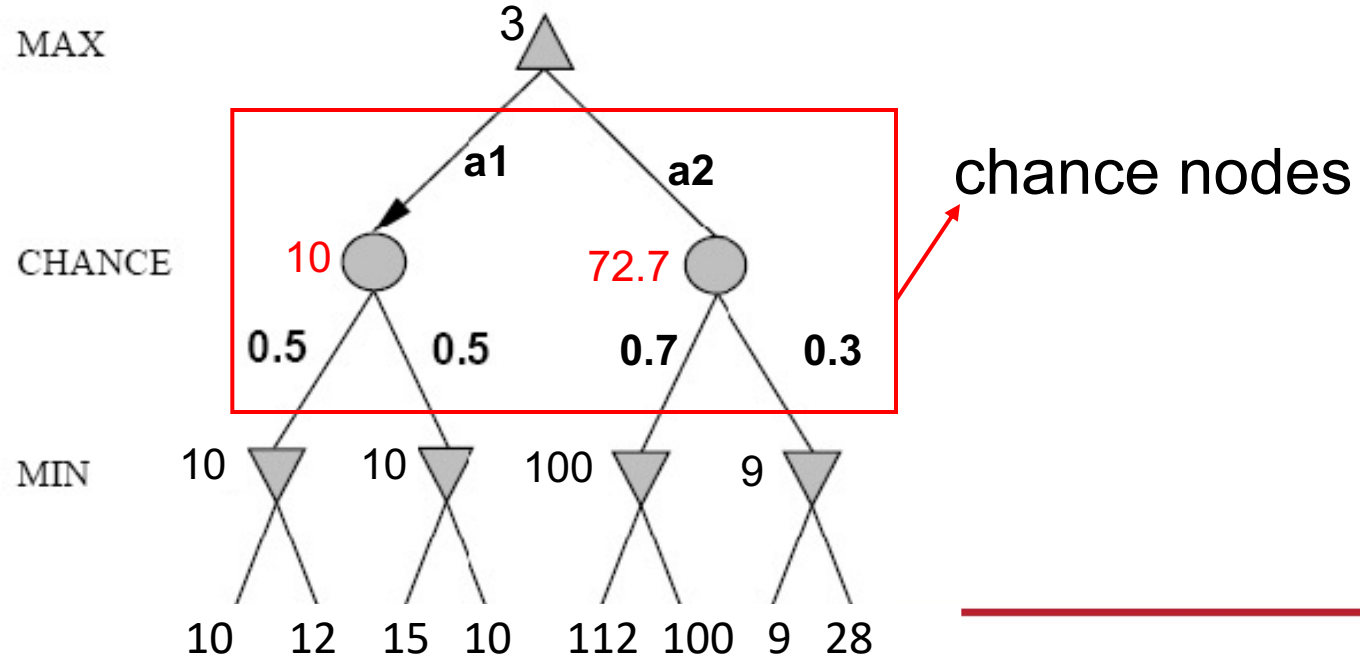
Nondeterministic games

- Example of another game instance with coin-flipping:
  - Should the player MAX choose action a1 or a2?
    - If he chooses a1, then it is guaranteed to get a 10
    - If he chooses a2, then it will get at least 9 and at most 100
  - The action **a1** is the “safest” move…
  - But with action **a2**, MAX has:
    - 30% chance to “lose” 1 point compared to a1
    - 70% chance to “win” 90 points compared to a1
    - -> Expected minimax will decide, based on the **expected utility value**



Nondeterministic games

- Example of another game instance with coin-flipping:
  - Should the player MAX choose action a1 or a2?
    - If he chooses a1, then it is guaranteed to get a 10
    - If he chooses a2, then it will get at least 9 and at most 100
  - The action **a1** is the "safest" move…
  - But with action **a2**, MAX has:
    - 30% chance to "lose" 1 point compared to a1
    - 70% chance to "win" 90 points compared to a1
    - -> Expected minimax will decide, based on the **expected utility value**
      - In this case, MAX will choose **a2** based on expected minimax



Some insights on non-deterministic games / games with imperfect info / games with imperfect opponent

Games with imperfect information

Note on games of imperfect information

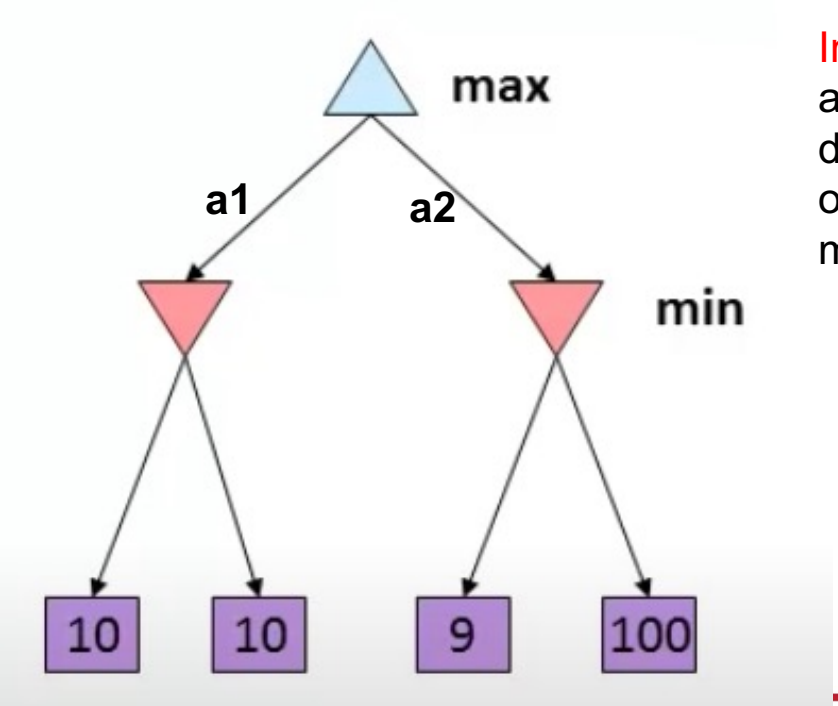
- **Note:**
  - Chance is not only due to flipping coins or rolling dice!
  - **Imperfect information** (partially observable environment) might also lead MAX to take a chance
    - *E.g.,* poker games, where opponent's initial cards are unknown
    - MAX could calculate a probability for each possible deal for MIN, and make decisions based on these probabilities
      - Seems just like having one big dice roll at the beginning of the game
- **How to deal with games of imperfect information?**
  - Compute the minimax value of each action in each deal, then choose the action with highest expected value over all deals
  - Special case: if an action is optimal for all deals, then it's optimal
  - GIB, one of the best bridge programs, approximates this idea by
    - ➢ generating 100 deals consistent with bidding information from the opponent
    - ➢ picking the action that wins most tricks on average

Some insights on non-deterministic games / games with imperfect info / games with imperfect opponent

Games with imperfect opponent

What about imperfect opponents?

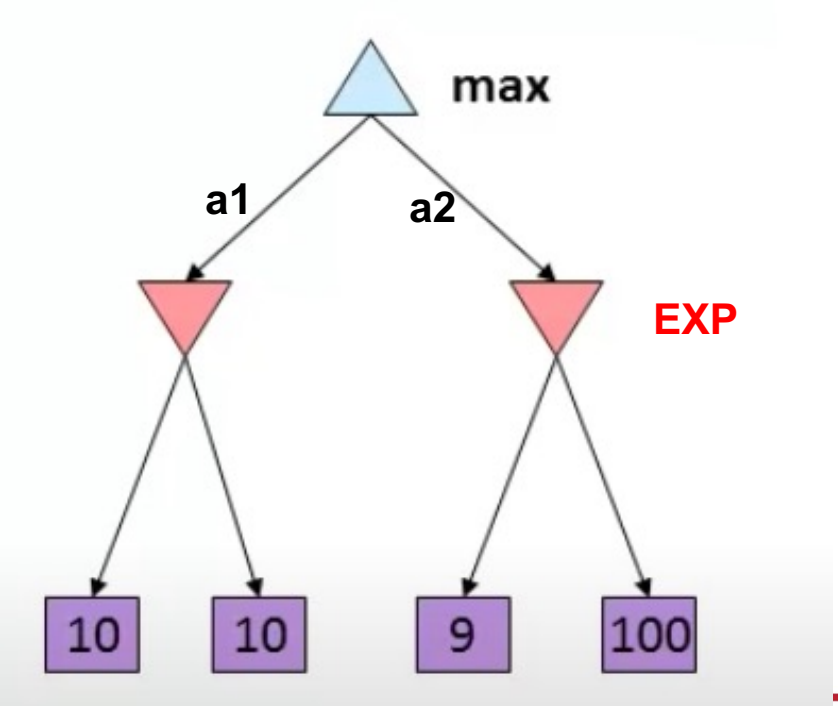
- **Note:**
  - Chance is not only due to flipping coins or rolling dice!
  - A similar algorithm (expectimax) could be applied if MIN is an **imperfect** player
    - In that case, depending on the “level” of MIN, player MAX might want to take its chance and choose a2
      - The “level” of MIN being translated into a probability, for MIN, to take the best action that it can



Important note: MINIMAX would always choose action a1, but, depending on the “level” of the opponent, expectimax algorithm might choose action a2…

What about imperfect opponents?

- **Note:**
  - Chance is not only due to flipping coins or rolling dice!
  - A similar algorithm (expectimax) could be applied if MIN is an **imperfect** player
    - In that case, depending on the “level” of MIN, player MAX might want to take its chance and choose a2
      - The “level” of MIN being translated into a probability, for MIN, to take the best action that it can
    - In that case, player MIN is renamed as EXP (imperfect player)



What about imperfect opponents?

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 = -∞
    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
```

Summary

Summary

- In the first part of this lecture, we focused on 2 player-games, deterministic, with perfect information (fully observable) and, in this context, we introduced:
  - Minimax algorithm
  - α-β pruning
    - computes the same optimal move as minimax, but achieves greater efficiency by eliminating subtrees that are provably irrelevant
  - To make imperfect, yet "good-enough" decisions, under time pressure
    - CUTOFF-TEST (*e.g.:* depth limit)
    - Heuristic evaluation function EVAL: estimated utility
- In the second part of this lecture, we gave some insights about:
  - Non-deterministic games and games with imperfect information (partially observable search problems)
    - Can be solved with expected minimax algorithm
  - Games with imperfect opponent
    - Can be implemented with expectimax algorithm

Chapter 3 – part 4

Questions

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!

soict.hust.edu.vn/ fb.com/groups/soict