Intro. to Intelligent Systems

Sem. Ganjil 2023/2024

08–Adversarial Search

Lionov

INFORMATIKA
UNPAR

Overview

- Game Search
- Minimax & $\alpha - \beta$ Pruning
- Monte Carlo Tree Search
- Stochastic Game
- Partially Observable Games

Background

Adversarial

involving people opposing or disagreeing with each other

Background

Adversarial

involving people opposing or disagreeing with each other

≡ games

Background

Adversarial

involving people opposing or disagreeing with each other

$\equiv$ games

Games are a big deal in A.I. because they are hard to solve

Background

Adversarial

involving people opposing or disagreeing with each other

$\equiv$ games

Games are a big deal in A.I. because they are hard to solve

Hard? Chess: 35^{100} nodes, still needs to make decision even when the optimal decision is infeasible

Background

Adversarial

involving people opposing or disagreeing with each other

$\equiv$ games

Games are a big deal in A.I. because they are hard to solve

Hard? Chess: 35^{100} nodes, still needs to make decision even when the optimal decision is infeasible

Competitive environments: two or more agents have conflicting goals

Background

Adversarial

involving people opposing or disagreeing with each other

$\equiv$ games

Games are a big deal in A.I. because they are hard to solve

Hard? Chess: 35^{100} nodes, still needs to make decision even when the optimal decision is infeasible

Competitive environments: two or more agents have conflicting goals

An agent cannot control the plot of its opponent!

Background

Adversarial

involving people opposing or disagreeing with each other

$\equiv$ games

Games are a big deal in A.I. because they are hard to solve

Hard? Chess: 35^{100} nodes, still needs to make decision even when the optimal decision is infeasible

Competitive environments: two or more agents have conflicting goals

An agent cannot control the plot of its opponent!

Optimal solution is not a sequence of actions or a final state, but a **strategy** (policy). E.g., if opponent does x, the agent does y, else if opponent does z, the agent does something else.

Background

Adversarial

involving people opposing or disagreeing with each other

$\equiv$ games

Games are a big deal in A.I. because they are hard to solve

Hard? Chess: 35^{100} nodes, still needs to make decision even when the optimal decision is infeasible

Competitive environments: two or more agents have conflicting goals

An agent cannot control the plot of its opponent!

Optimal solution is not a sequence of actions or a final state, but a **strategy** (policy). E.g., if opponent does x, the agent does y, else if opponent does z, the agent does something else.

Games are modeled as search problems and use heuristic evaluation functions.

Background

Consider a **zero-sum game** with an environment that:

- Deterministic
- Perfect information (fully observable)
- Sequential (taking turn)
- Static
- Multi agents

We use term **move** for "action" and **position** for "state"

Background

Consider a **zero-sum game** with an environment that:

- Deterministic
- Perfect information (fully observable)
- Sequential (taking turn)
- Static
- Multi agents

We use term **move** for "action" and **position** for "state"

Examples:

- Tic-Tac-Toe
- Chess
- Go
- Checkers
- Hex

Background

Consider a **zero-sum game** with an environment that:

- Deterministic
- Perfect information (fully observable)
- Sequential (taking turn)
- Static
- Multi agents

We use term **move** for "action" and **position** for "state"

Examples:

- Tic-Tac-Toe
- Chess
- Go
- Checkers
- Hex

Background

Consider a **zero-sum game** with an environment that:

- Deterministic
- Perfect information (fully observable)
- Sequential (taking turn)
- Static
- Multi agents

Deep Blue vs Gary Kasparov, 1997
Deep Fritz vs Vladimir Kramnik, 2006

We use term **move** for “action” and **position** for “state”

Examples:

- Tic-Tac-Toe
- Chess
- Go
- Checkers
- Hex

Background

Consider a **zero-sum game** with an environment that:

- Deterministic
- Perfect information (fully observable)
- Sequential (taking turn)
- Static
- Multi agents

We use term **move** for "action" and **position** for "state"

Deep Blue vs Gary Kasparov, 1997
Deep Fritz vs Vladimir Kramnik, 2006

Examples:

- Tic-Tac-Toe
- Chess
- Go
- Checkers
- Hex

Google Deep Mind AlphaGo vs Lee Sedol

Background

Consider a **zero-sum game** with an environment that:

- Deterministic
- Perfect information (fully observable)
- Sequential (taking turn)
- Static
- Multi agents

Deep Blue vs Gary Kasparov, 1997
Deep Fritz vs Vladimir Kramnik, 2006

We use term **move** for "action" and **position** for "state"

Examples:

- Tic-Tac-Toe
- Chess
- Go
- Checkers
- Hex

Chinook vs Marion Tinsley. Chinook use database with 440 billion state, 1994

Google Deep Mind AlphaGo vs Lee Sedol

Background

Consider a **zero-sum game** with an environment that:

- Deterministic
- Perfect information (fully observable)
- Sequential (taking turn)
- Static
- Multi agents

Deep Blue vs Gary Kasparov, 1997
Deep Fritz vs Vladimir Kramnik, 2006

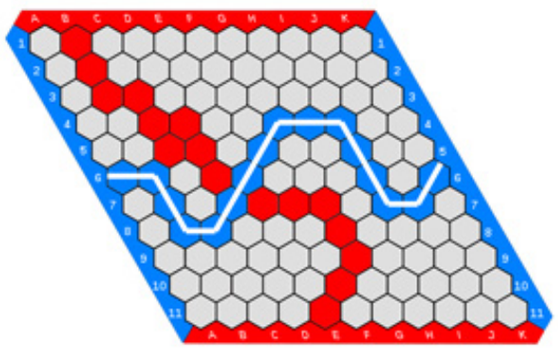
We use term **move** for “action” and **position** for “state”

Examples:

- Tic-Tac-Toe
- Chess
- Go
- Checkers
- Hex

Chinook vs Marion Tinsley. Chinook use database with 440 billion state, 1994

Google Deep Mind AlphaGo vs Lee Sedol



Background

Consider a **zero-sum game** with an environment that:

- Deterministic
- Perfect information (fully observable)
- Sequential (taking turn)
- Static
- Multi agents

Deep Blue vs Gary Kasparov, 1997
Deep Fritz vs Vladimir Kramnik, 2006

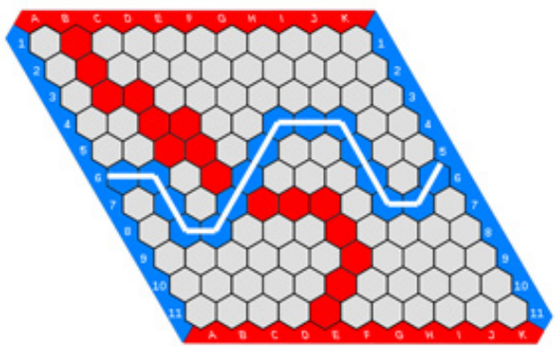
We use term **move** for “action” and **position** for “state”

Examples:

- Tic-Tac-Toe
- Chess
- Go
- Checkers
- Hex

Chinook vs Marion Tinsley. Chinook use database with 440 billion state, 1994

Google Deep Mind AlphaGo vs Lee Sedol



Othello: human champions *refuse* to compete :D

Game's Definition

A game with two players: ***MAX*** and ***MIN*** (***MAX*** always moves first)

Game's Definition

A game with two players: ***MAX*** and ***MIN*** (***MAX*** always moves first)

Embedded thinking (or backward reasoning)!!!

Game's Definition

A game with two players: ***MAX*** and ***MIN*** (***MAX*** always moves first)

Embedded thinking (or backward reasoning)!!!

A game is fomally define:

- S_0: The initial state, set-up at the start of the game

Game's Definition

A game with two players: ***MAX*** and ***MIN*** (***MAX*** always moves first)

Embedded thinking (or backward reasoning)!!!

A game is fomally define:

- S_0: The initial state, set-up at the start of the game
- **To-Move(s)**: The player whose turn it is to move in state s.

Game's Definition

A game with two players: ***MAX*** and ***MIN*** (***MAX*** always moves first)

Embedded thinking (or backward reasoning)!!!

A game is fomally define:

- S_0: The initial state, set-up at the start of the game
- **To-Move(s)**: The player whose turn it is to move in state s.
- **Actions(s)**: The set of legal moves in state s

Game's Definition

A game with two players: ***MAX*** and ***MIN*** (***MAX*** always moves first)

Embedded thinking (or backward reasoning)!!!

A game is fomally define:

- S_0: The initial state, set-up at the start of the game
- **To-Move(s)**: The player whose turn it is to move in state s.
- **Actions(s)**: The set of legal moves in state s
- **Result (s, a)**: The transition model defines the state resulting from taking action a in state s

Game's Definition

A game with two players: ***MAX*** and ***MIN*** (***MAX*** always moves first)

Embedded thinking (or backward reasoning)!!!

A game is fomally define:

- S_0: The initial state, set-up at the start of the game
- **To-Move(s)**: The player whose turn it is to move in state s.
- **Actions(s)**: The set of legal moves in state s
- **Result (s, a)**: The transition model defines the state resulting from taking action a in state s
- **Is-Terminal(s)** : true when the game is over (at terminal states)

Game's Definition

A game with two players: ***MAX*** and ***MIN*** (***MAX*** always moves first)

Embedded thinking (or backward reasoning)!!!

A game is fomally define:

- S_0: The initial state, set-up at the start of the game
- **To-Move(s)**: The player whose turn it is to move in state s.
- **Actions(s)**: The set of legal moves in state s
- **Result (s, a)**: The transition model defines the state resulting from taking action a in state s
- **Is-Terminal(s)** : true when the game is over (at terminal states)
- **Utility(s, p)**: A utility function (objective/ payoff function) that defines the final numeric value to players p when the game ends in terminal state s

State-space Graph in a Game

The intial state $\mathcal{S}_0$, ACTIONS, and RESULT define the **State-space Graph**

State-space Graph in a Game

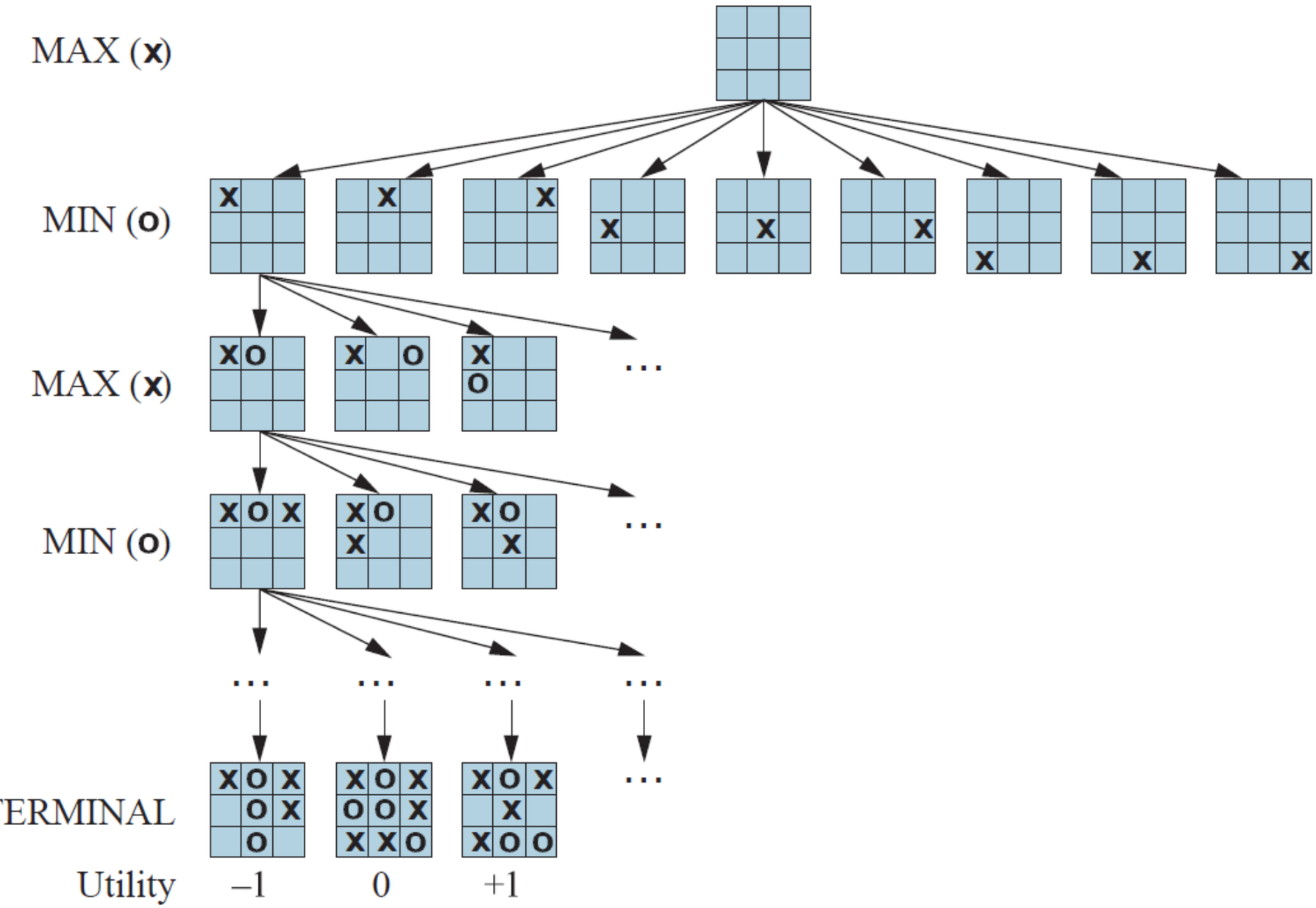
The intial state $\mathcal{S}_0$, ACTIONS, and RESULT define the **State-space Graph**

Search tree over part of that graph to determine what move (***ply***) to make

State-space Graph in a Game

The intial state $\mathcal{S}_0$, ACTIONS, and RESULT define the **State-space Graph**

Search tree over part of that graph to determine what move (***ply***) to make



Minimax Search

Both players search for a strategy that can win the game

Minimax Search

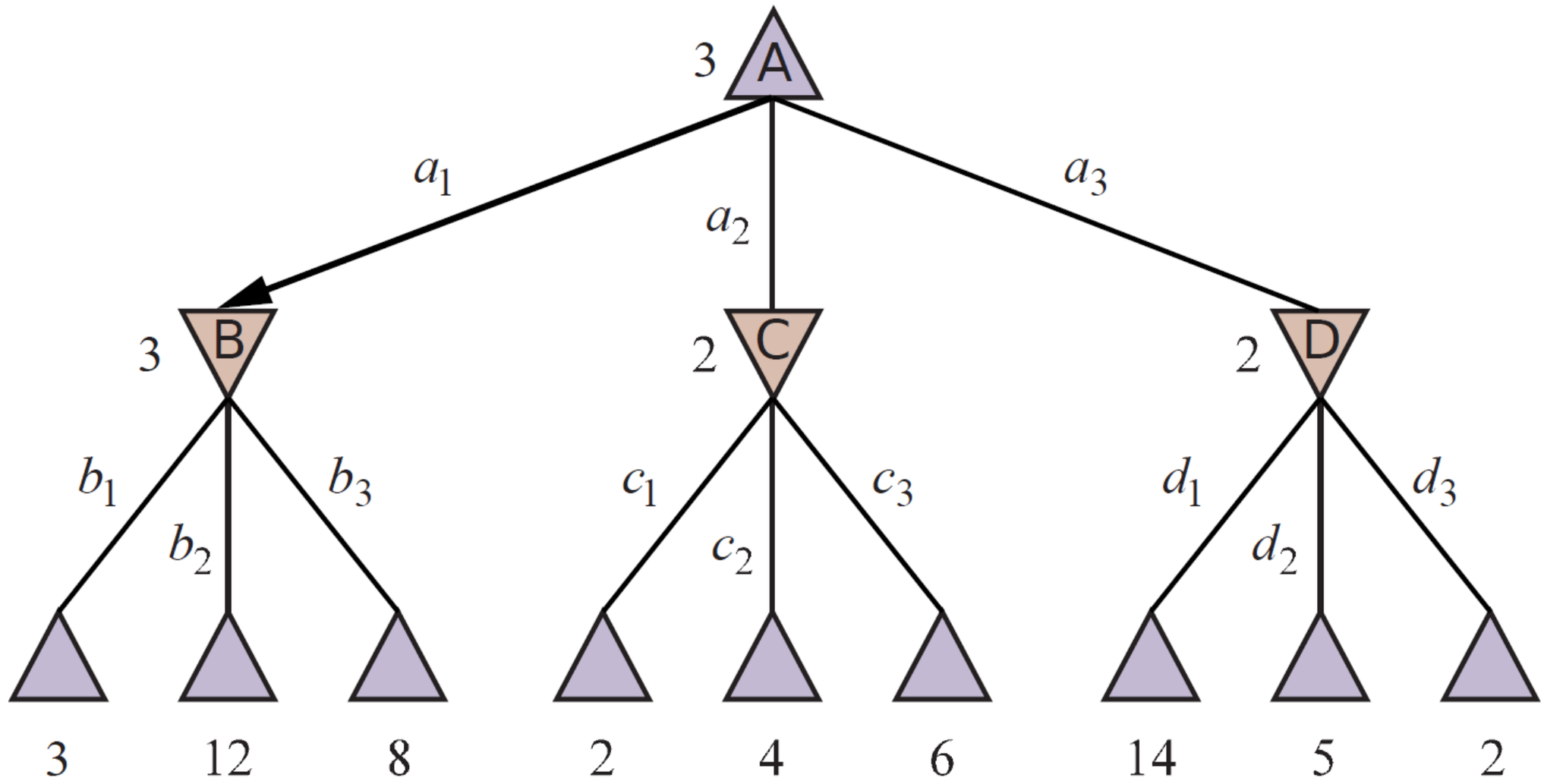
Both players search for a strategy that can win the game

- ***MAX*** player refers moves that maximise the value of utility function
- ***MIN*** player refers moves that minimise the value of utility function

Minimax Search

Both players search for a strategy that can win the game

- ***MAX*** player refers moves that maximise the value of utility function
- ***MIN*** player refers moves that minimise the value of utility function



Minimax Search

Both players search for a strategy that can win the game

- ***MAX*** player refers moves that maximise the value of utility function
- ***MIN*** player refers moves that minimise the value of utility function

Optimal strategy can be determined by working out the minimax value of each state in the tree (MINIMAX(s)).

- assume both players play optimally from there to the terminal state

Minimax Search

Both players search for a strategy that can win the game

- ***MAX*** player refers moves that maximise the value of utility function
- ***MIN*** player refers moves that minimise the value of utility function

Optimal strategy can be determined by working out the minimax value of each state in the tree (MINIMAX(s)).

- assume both players play optimally from there to the terminal state

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

Minimax Search

Both players search for a strategy that can win the game

- ***MAX*** player refers moves that maximise the value of utility function
- ***MIN*** player refers moves that minimise the value of utility function

Optimal strategy can be determined by working out the minimax value of each state in the tree (MINIMAX(s)).

- assume both players play optimally from there to the terminal state

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

We can identify the **minimax decision** at the root

Minimax Algorithm

```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move

function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move ← v2, a
    return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

Time complexity is $O(b^m)$ and the space complexity is $O(bm)$

Where m is the maximum depth of the tree and there are b legal moves at each point

Minimax in Multiplayer Games

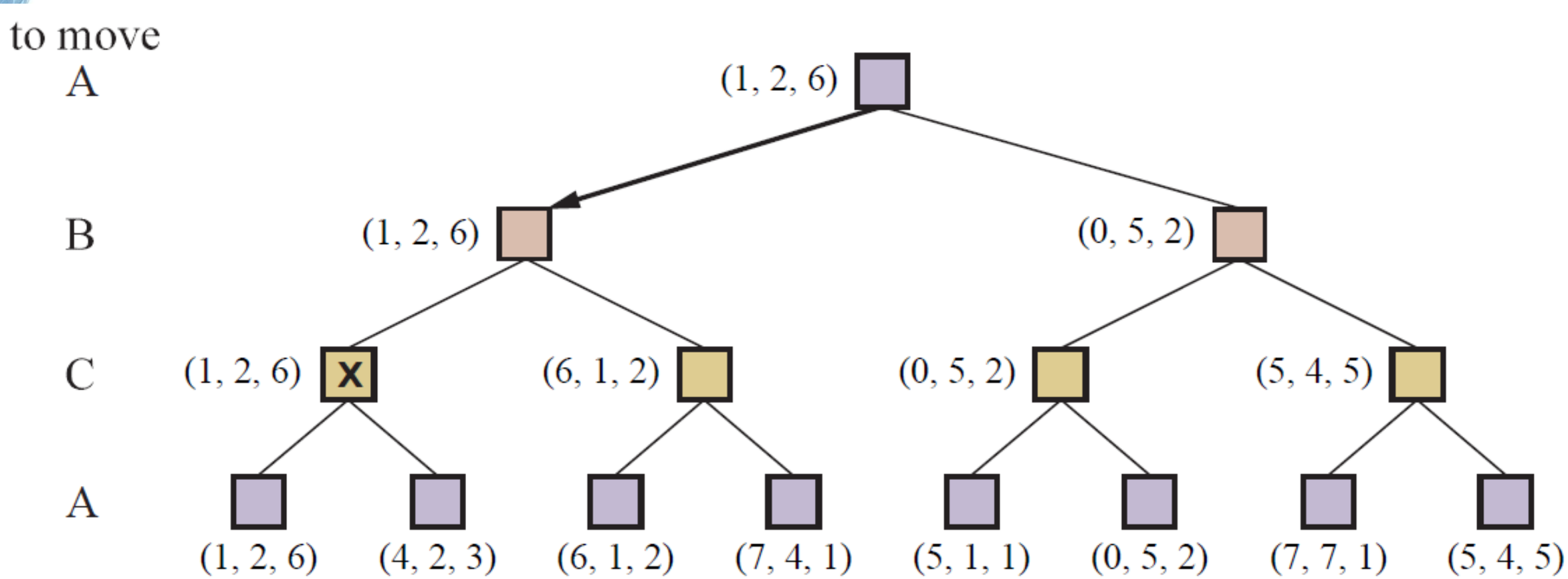
Returns a vector of n values (n = the number of players)

Natural alliance may emerge

Minimax in Multiplayer Games

Returns a vector of n values (n = the number of players)

Natural alliance may emerge



Alpha-Beta Pruning

Pruning: cut off parts of the tree that make no difference to the outcome

Alpha-Beta Pruning

Pruning: cut off parts of the tree that make no difference to the outcome

α: the value of the best (highest-value) choice found so far for ***MAX***("at least")

β: the value of the best (lowest-value) choice found so far for ***MIN*** ("at most")

Alpha-Beta Pruning

Pruning: cut off parts of the tree that make no difference to the outcome

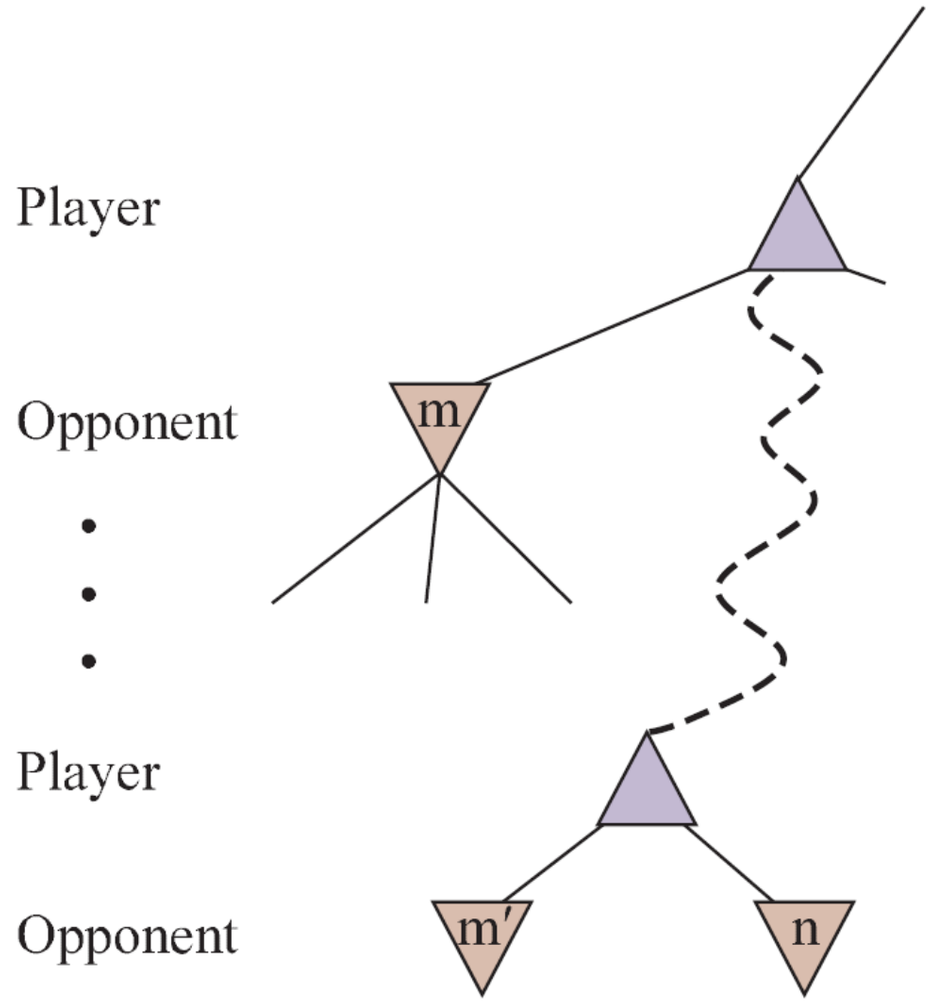
α: the value of the best (highest-value) choice found so far for ***MAX***("at least")

β: the value of the best (lowest-value) choice found so far for ***MIN*** ("at most")

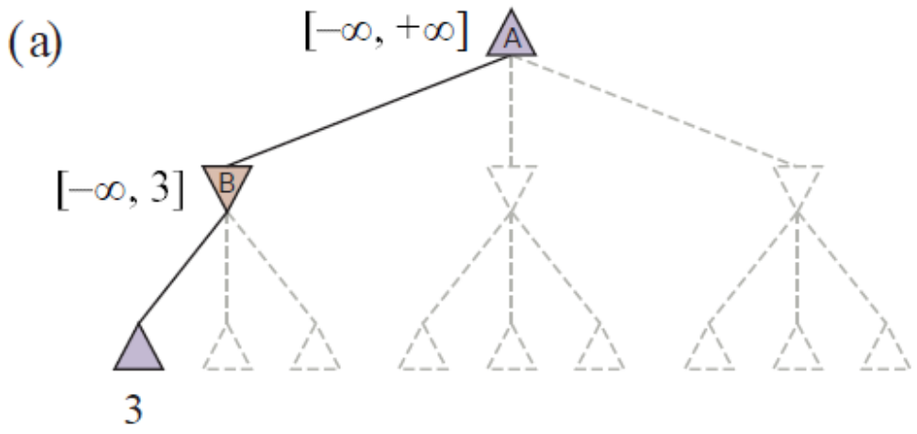
The general principle:
Consider a node n such that a player has a choice of moving to n.

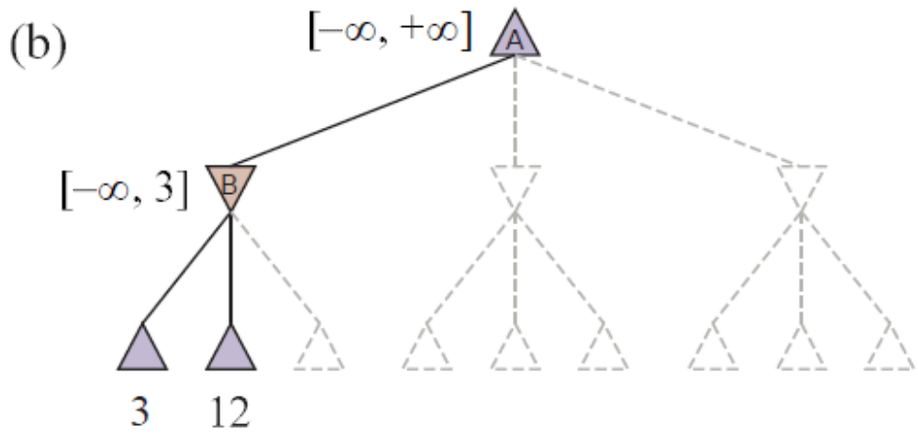
If the player has **a better choice** either at the same level (m') or at any point higher up in the tree (m), then the player will never move to n.

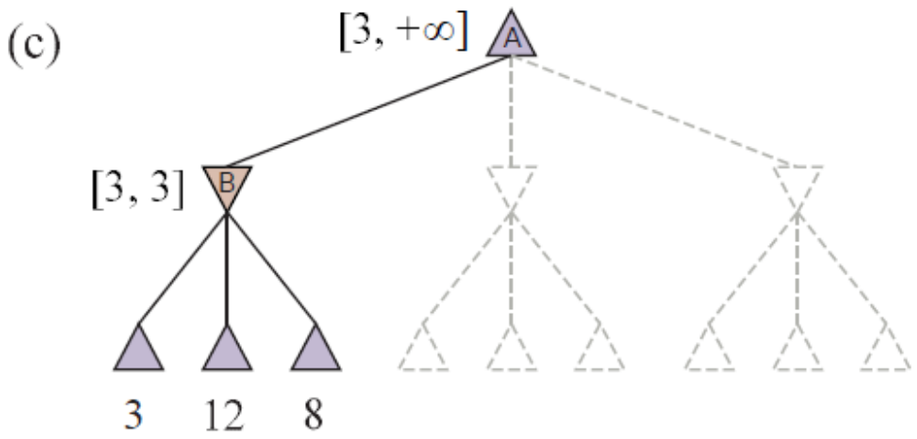
Once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

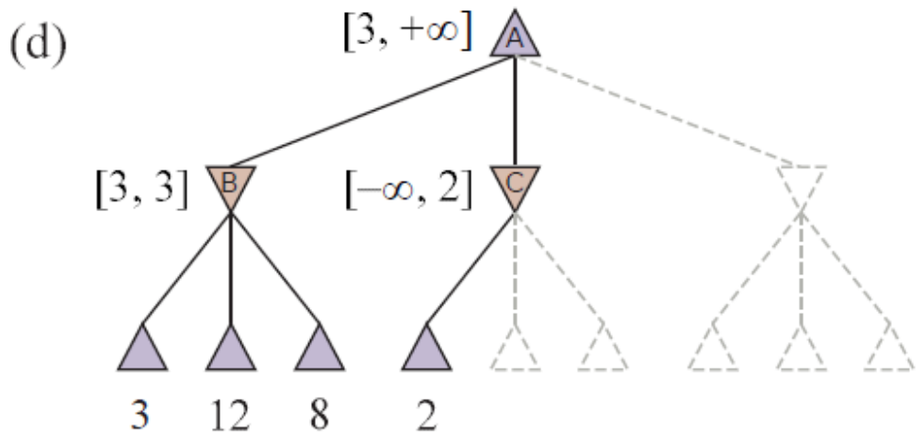


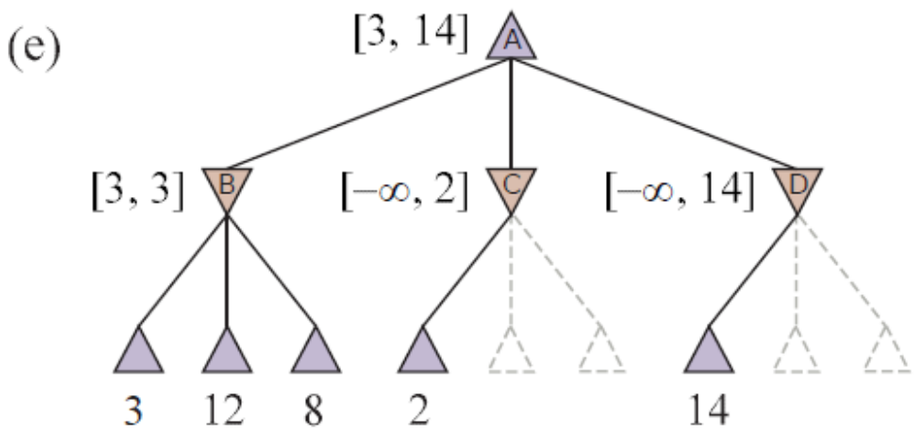
Alpha-Beta Pruning

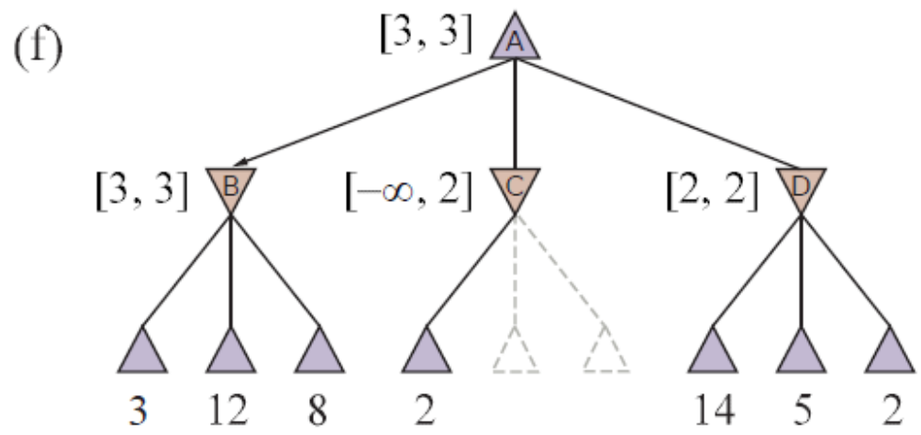












INFORMATIKA UNPAR

$\alpha - \beta$ Pruning Algorithm

```
function ALPHA-BETA-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, −∞, +∞)
    return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
        if v2 > v then
            v, move ← v2, a
            α ← MAX(α, v)
        if v ≥ β then return v, move
    return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
        if v2 < v then
            v, move ← v2, a
            β ← MIN(β, v)
        if v ≤ α then return v, move
    return v, move
```

Move Ordering

Performance of the Alpha-Beta Pruning depends on the order of the moves

Move Ordering

Performance of the Alpha-Beta Pruning depends on the order of the moves

- Killer moves heuristic: try the “killer moves” first

Move Ordering

Performance of the Alpha-Beta Pruning depends on the order of the moves

- Killer moves heuristic: try the "killer moves" first
- Use *Iterative Deepening* to find the killer moves

Move Ordering

Performance of the Alpha-Beta Pruning depends on the order of the moves

- Killer moves heuristic: try the "killer moves" first
- Use *Iterative Deepening* to find the killer moves

Transpositions: different permutations of the move sequence end up in the same position (and becomes repeated states)

Move Ordering

Performance of the Alpha-Beta Pruning depends on the order of the moves

- Killer moves heuristic: try the "killer moves" first
- Use *Iterative Deepening* to find the killer moves

Transpositions: different permutations of the move sequence end up in the same position (and becomes repeated states)

Transposition Table: caches the heuristic value of states

Move Ordering

Performance of the Alpha-Beta Pruning depends on the order of the moves

- Killer moves heuristic: try the "killer moves" first
- Use *Iterative Deepening* to find the killer moves

Transpositions: different permutations of the move sequence end up in the same position (and becomes repeated states)

Transposition Table: caches the heuristic value of states

Even with above strategies, minimax is not suitable for "large" games

Move Ordering

Performance of the Alpha-Beta Pruning depends on the order of the moves

- Killer moves heuristic: try the "killer moves" first
- Use *Iterative Deepening* to find the killer moves

Transpositions: different permutations of the move sequence end up in the same position (and becomes repeated states)

Transposition Table: caches the heuristic value of states

Even with above strategies, minimax is not suitable for "large" games

- **Type A strategy:** considers all possible moves to a certain depth in the search tree, and then uses a heuristic evaluation function to estimate the utility of states at that depth.

Move Ordering

Performance of the Alpha-Beta Pruning depends on the order of the moves

- Killer moves heuristic: try the "killer moves" first
- Use *Iterative Deepening* to find the killer moves

Transpositions: different permutations of the move sequence end up in the same position (and becomes repeated states)

Transposition Table: caches the heuristic value of states

Even with above strategies, minimax is not suitable for "large" games

- **Type A strategy**: considers all possible moves to a certain depth in the search tree, and then uses a heuristic evaluation function to estimate the utility of states at that depth.
- **Type B strategy**: ignores moves that look bad, and follows promising lines "as far as possible".

Heuristic for $\alpha - \beta$ Search

In real game playing, the game tree may be very large and we only have limited computation time

Heuristic for $\alpha - \beta$ Search

In real game playing, the game tree may be very large and we only have limited computation time

Cut off the search early:

Heuristic for $\alpha - \beta$ Search

In real game playing, the game tree may be very large and we only have limited computation time

Cut off the search early:

- Replace the terminal test by a **cutoff-test**: return TRUE for terminal states, otherwise its free to decide when to cut off the search, based on the search depth and any other properties of the state.

Heuristic for $\alpha - \beta$ Search

In real game playing, the game tree may be very large and we only have limited computation time

Cut off the search early:

- Replace the terminal test by a **cutoff-test**: return TRUE for terminal states, otherwise its free to decide when to cut off the search, based on the search depth and any other properties of the state.
- Apply a heuristic **evaluation function** to states: treating nonterminal nodes as if they were terminal by replacing UITLITY function with EVAL, which estimates a stateâs utility.

Heuristic for $\alpha - \beta$ Search

In real game playing, the game tree may be very large and we only have limited computation time

Cut off the search early:

- Replace the terminal test by a **cutoff-test**: return TRUE for terminal states, otherwise its free to decide when to cut off the search, based on the search depth and any other properties of the state.
- Apply a heuristic **evaluation function** to states: treating nonterminal nodes as if they were terminal by replacing UITLITY function with EVAL, which estimates a stateâs utility.

$$\text{H-MINIMAX}(s) = \begin{cases} \text{EVAL}(s, \text{MAX}) & \text{if IS-CUTOFF}(s, d) \\ \max_a \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_a \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: EVAL(s, p)=UTILITY(s, p).

Evaluation Function

The heuristic Eval(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: Eval(s, p)=Utility(s, p).
- For nonterminal states: Utility$(loss, p) \leqslant$ Eval$(s, p) \leqslant$ Utility(win, p)

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: EVAL(s, p)=UTILITY(s, p).
- For nonterminal states: UTILITY$(loss, p) \leqslant$ EVAL$(s, p) \leqslant$ UTILITY(win, p)

Properties of a good evaluation function:

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: EVAL(s, p)=UTILITY(s, p).
- For nonterminal states: UTILITY($loss, p$) $\leqslant$ EVAL(s, p) $\leqslant$ UTILITY(win, p)

Properties of a good evaluation function:

- the computation must be fast

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: EVAL(s, p)=UTILITY(s, p).
- For nonterminal states: UTILITY$(loss, p) \leqslant$ EVAL$(s, p) \leqslant$ UTILITY(win, p)

Properties of a good evaluation function:

- the computation must be fast
- should be strongly correlated with the actual chances of winning

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: EVAL(s, p)=UTILITY(s, p).
- For nonterminal states: UTILITY($loss, p$) $\leqslant$ EVAL(s, p) $\leqslant$ UTILITY(win, p)

Properties of a good evaluation function:

- the computation must be fast
- should be strongly correlated with the actual chances of winning

Most evaluation functions work by calculating various features of the state

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: EVAL(s, p)=UTILITY(s, p).
- For nonterminal states: UTILITY$(loss, p) \leqslant$ EVAL$(s, p) \leqslant$ UTILITY(win, p)

Properties of a good evaluation function:

- the computation must be fast
- should be strongly correlated with the actual chances of winning

Most evaluation functions work by calculating various features of the state

Weighted combination of features of a game state:

$$\text{EVAL}(s) = \omega f_1(s) + \omega f_2(s) + \ldots + \omega f_n(s) = \Sigma_i^n \omega f_i(s)$$

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: EVAL(s, p)=UTILITY(s, p).
- For nonterminal states: UTILITY$(loss, p) \leqslant$ EVAL$(s, p) \leqslant$ UTILITY(win, p)

Properties of a good evaluation function:

- the computation must be fast
- should be strongly correlated with the actual chances of winning

Most evaluation functions work by calculating various features of the state

Weighted combination of features of a game state:

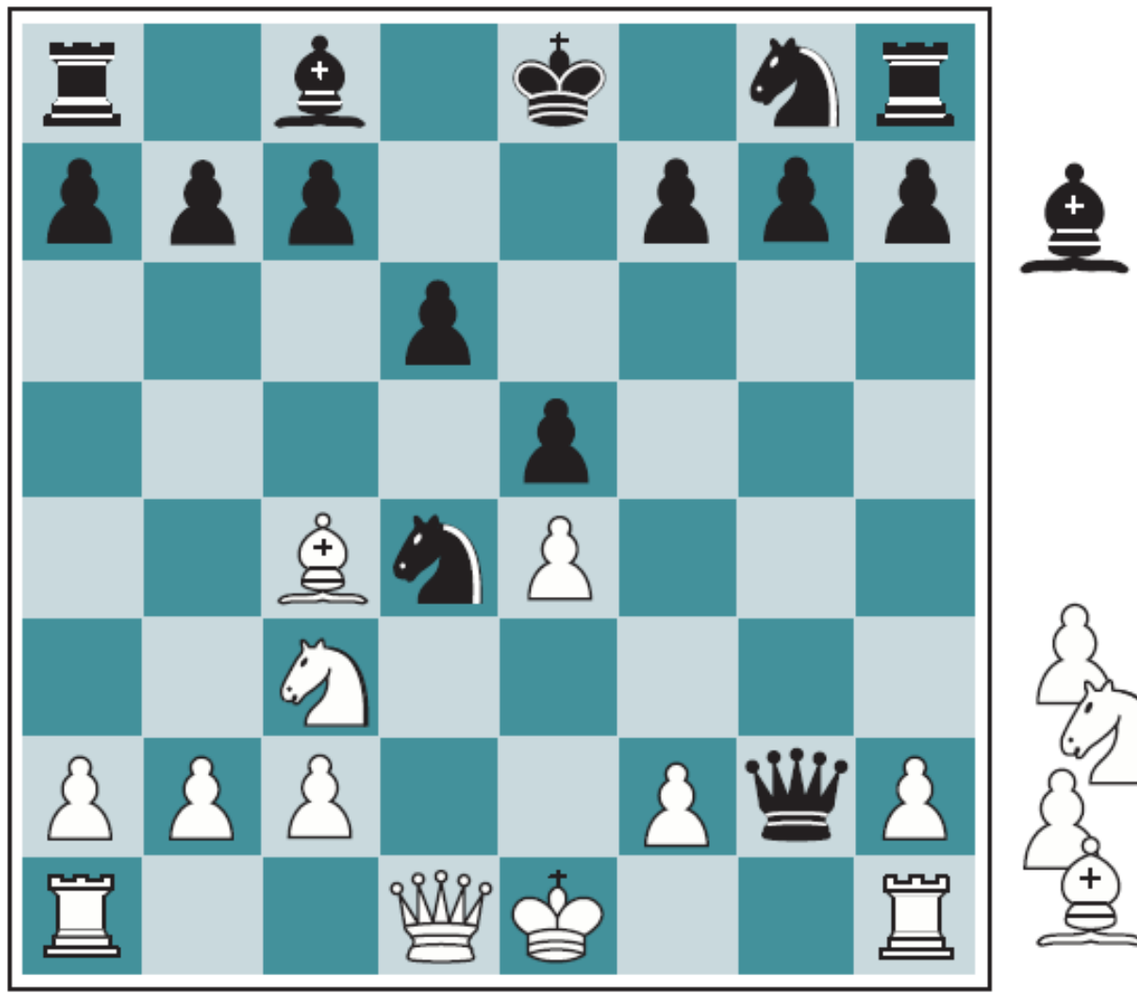
$$\text{EVAL}(s) = \omega f_1(s) + \omega f_2(s) + \ldots + \omega f_n(s) = \Sigma_i^n \omega f_i(s)$$

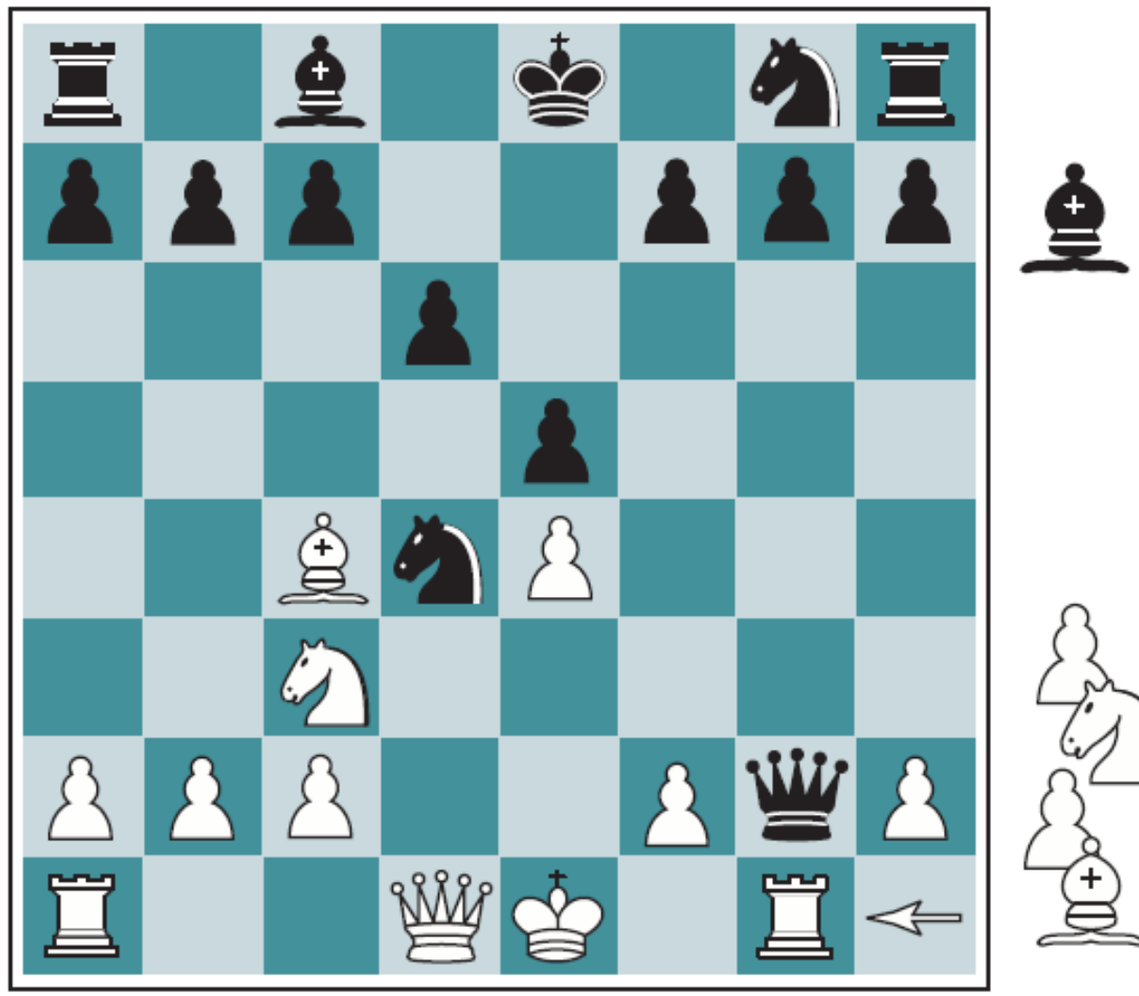
The features and their weights are results of years of experience or estimated by machine learning techniques

Evaluation Function

The heuristic EVAL(s, p) returns an *estimate* of the expected utility of state s

- For terminal states: EVAL(s, p)=UTILITY(s, p).
- For nonterminal states: UTILITY($loss, p$) $\leqslant$ EVAL(s, p) $\leqslant$ UTILITY(win, p)





Cutting-off Search

Cutting-off Search

Modify the ALPHA-BETA-SEARCH: it calls the EVAL function to cut off the search

if game. IS-CUTOFF(STATE, DEPTH) **then return** GAME.EVAL(STATE, PLAYER), NULL

Cutting-off Search

Modify the ALPHA-BETA-SEARCH: it calls the EVAL function to cut off the search

if game. IS-CUTOFF(STATE, DEPTH) **then return** GAME.EVAL(STATE, PLAYER), NULL

The simplest approach: set a fixed depth limit for IS-CUTOFF(STATE, DEPTH)

Cutting-off Search

Modify the ALPHA-BETA-SEARCH: it calls the EVAL function to cut off the search

if game. IS-CUTOFF(STATE, DEPTH) **then return** GAME.EVAL(STATE, PLAYER), NULL

The simplest approach: set a fixed depth limit for IS-CUTOFF(STATE, DEPTH)

Iterative Deepening can be used to choose the depth

Cutting-off Search

Modify the ALPHA-BETA-SEARCH: it calls the EVAL function to cut off the search

if game. IS-CUTOFF(STATE, DEPTH) **then return** GAME.EVAL(STATE, PLAYER), NULL

The simplest approach: set a fixed depth limit for IS-CUTOFF(STATE, DEPTH)

Iterative Deepening can be used to choose the depth

Transposition table to keep states visited and use it to improve move ordering

Cutting-off Search

Modify the ALPHA-BETA-SEARCH: it calls the EVAL function to cut off the search

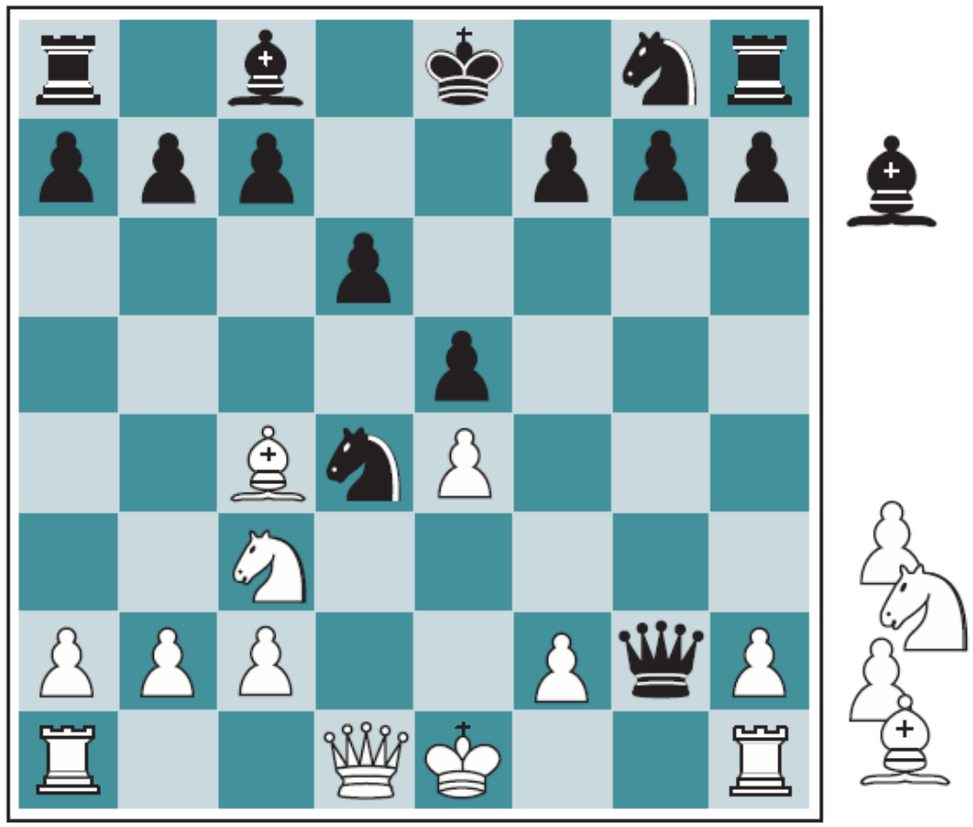
if game. IS-CUTOFF(STATE, DEPTH) **then return** GAME.EVAL(STATE, PLAYER), NULL

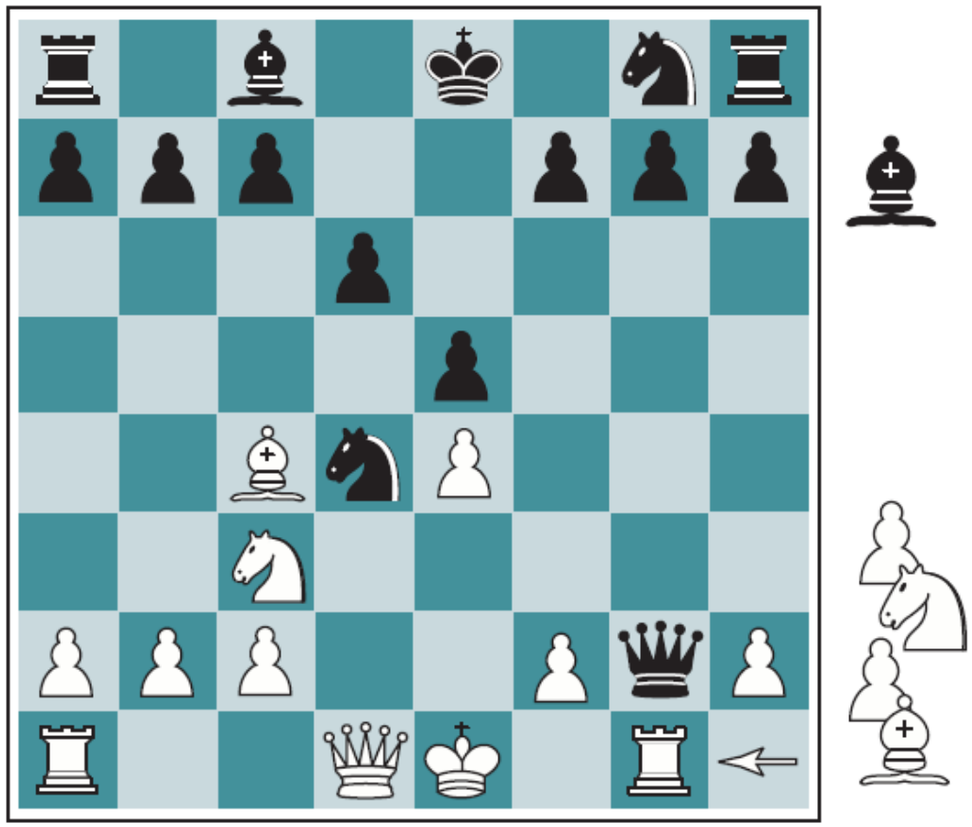
The simplest approach: set a fixed depth limit for IS-CUTOFF(STATE, DEPTH)

Iterative Deepening can be used to choose the depth

Transposition table to keep states visited and use it to improve move ordering

This strategies may lead to error:





Search vs Lookup

In chess, start a game by considering a tree of a billion states is not efficient

Search vs Lookup

In chess, start a game by considering a tree of a billion states is not efficient

- For the openings: the computer is mostly relying on the expertise of humans and it can be stored in the table for the computer's use

Search vs Lookup

In chess, start a game by considering a tree of a billion states is not efficient

- For the openings: the computer is mostly relying on the expertise of humans and it can be stored in the table for the computer's use
- Near the end of the game: fewer possible positions, and thus it is easier to do lookup. Computer is much better than human in this aspect.

Search vs Lookup

In chess, start a game by considering a tree of a billion states is not efficient

- For the openings: the computer is mostly relying on the expertise of humans and it can be stored in the table for the computer's use
- Near the end of the game: fewer possible positions, and thus it is easier to do lookup. Computer is much better than human in this aspect.

  Map every possible state to the best move in that state. The computer can play perfectly by looking up the right move in this table.

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big
- Difficult to define a “perfect” evaluation functions

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big
- Difficult to define a "perfect" evaluation functions

Monte Carlo Tree Search

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big
- Difficult to define a “perfect” evaluation functions

Monte Carlo Tree Search

MCTS does not use a heuristic evaluation function

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big
- Difficult to define a "perfect" evaluation functions

Monte Carlo Tree Search

MCTS does not use a heuristic evaluation function

The value of a state: the average utility over a number of **simulations** (**playout** or **rollout**) of complete games starting from the state.

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big
- Difficult to define a "perfect" evaluation functions

Monte Carlo Tree Search

MCTS does not use a heuristic evaluation function

The value of a state: the average utility over a number of **simulations** (**playout** or **rollout**) of complete games starting from the state.

How do we choose what moves to make during the playout?

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big
- Difficult to define a "perfect" evaluation functions

Monte Carlo Tree Search

MCTS does not use a heuristic evaluation function

The value of a state: the average utility over a number of **simulations** (**playout** or **rollout**) of complete games starting from the state.

How do we choose what moves to make during the playout?

Needs a playout policy that biases the moves towards good ones: learned from self-plays b using neural networks or use game-specific heuristics

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big
- Difficult to define a "perfect" evaluation functions

Monte Carlo Tree Search

MCTS does not use a heuristic evaluation function

The value of a state: the average utility over a number of **simulations** (**playout** or **rollout**) of complete games starting from the state.

How do we choose what moves to make during the playout?

Needs a playout policy that biases the moves towards good ones: learned from self-plays b using neural networks or use game-specific heuristics

From what positions do we start & how many playouts do we allocate to each position?

Monte Carlo Tree Search

Heuristic of Alpha-Beta Tree Search have two major weaknesses:

- Branching factor is way too big
- Difficult to define a "perfect" evaluation functions

Monte Carlo Tree Search

MCTS does not use a heuristic evaluation function

The value of a state: the average utility over a number of **simulations** (**playout** or **rollout**) of complete games starting from the state.

How do we choose what moves to make during the playout?

Needs a playout policy that biases the moves towards good ones: learned from self-plays b using neural networks or use game-specific heuristics

From what positions do we start & how many playouts do we allocate to each position?

Pure Monte Carlo search: do N simulations starting from the current state and track which of the possible moves has the highest win percentage

Monte Carlo Tree Search

Pure MCTS is not enough, we need a selection policy that focuses the computational resources on the important parts of the game tree and balace:

- **exploration** of states that have had few playouts

Monte Carlo Tree Search

Pure MCTS is not enough, we need a selection policy that focuses the computational resources on the important parts of the game tree and balace:

- **exploration** of states that have had few playouts
- **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value

Monte Carlo Tree Search

Pure MCTS is not enough, we need a selection policy that focuses the computational resources on the important parts of the game tree and balace:

- **exploration** of states that have had few playouts
- **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value

MCTS maintains a search tree and grows it on each iteration with this four steps:

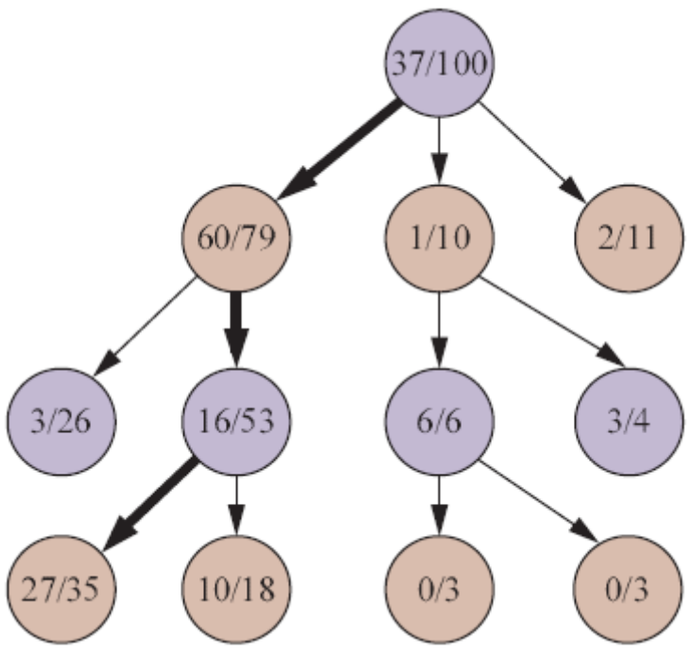
1. Selection
2. Expansion
3. Simulation
4. Back-Propagation

Monte Carlo Tree Search

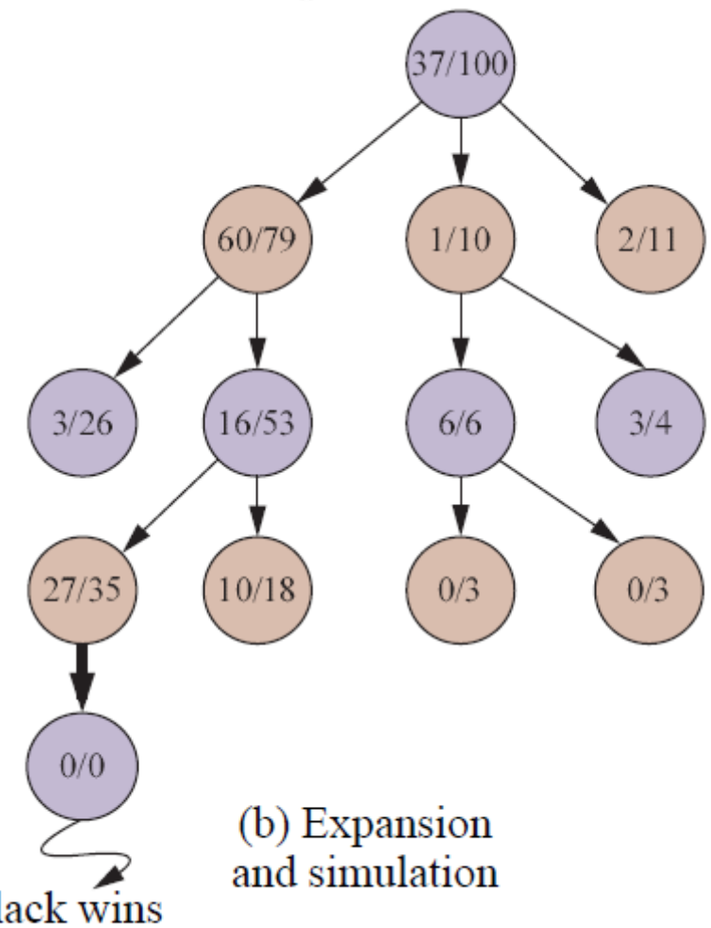
Pure MCTS is not enough, we need a selection policy that focuses the computational resources on the important parts of the game tree and balace:

- **exploration** of states that have had few playouts
- **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value

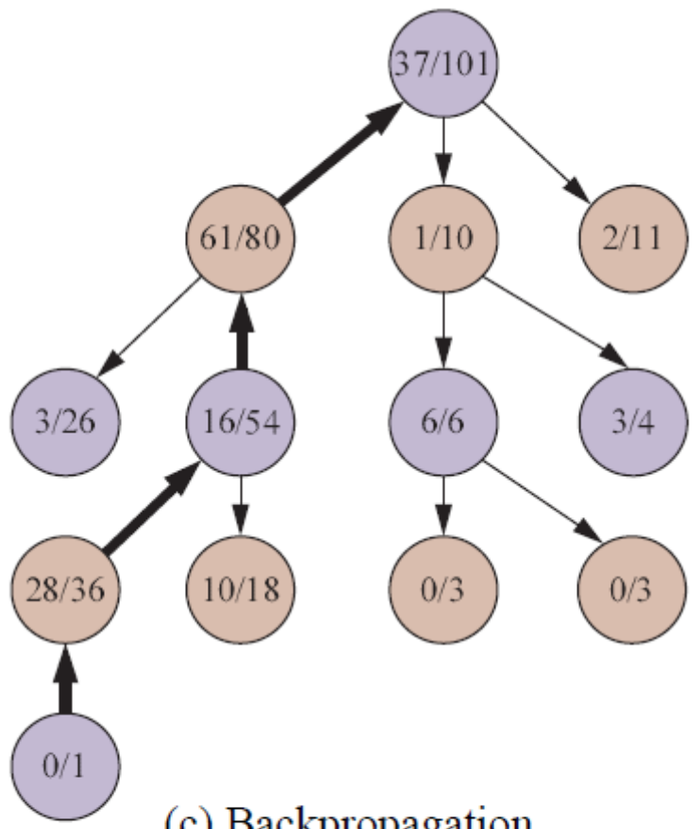
MCTS maintains a search tree and grows it on each iteration with this four steps:



(a) Selection



(b) Expansion and simulation



(c) Backpropagation

Monte Carlo Tree Search

Pure MCTS is not enough, we need a selection policy that focuses the computational resources on the important parts of the game tree and balace:

- **exploration** of states that have had few playouts
- **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value

MCTS maintains a search tree and grows it on each iteration with this four steps:

1. Selection
2. Expansion
3. Simulation
4. Back-Propagation

Exp. of a selection policy: "upper confidence bounds applied to trees" or **UCT**.
UCT ranks possible move based on an upper confidence bound formula **UCB1**

Monte Carlo Tree Search

Pure MCTS is not enough, we need a selection policy that focuses the computational resources on the important parts of the game tree and balace:

- **exploration** of states that have had few playouts
- **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value

MCTS maintains a search tree and grows it on each iteration with this four steps:

1. Selection
2. Expansion
3. Simulation
4. Back-Propagation

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

Stochastic Games

Games have unpredictability, like in a real life by including a random element, such as the throwing of dice

Stochastic Games

Games have unpredictability, like in a real life by including a random element, such as the throwing of dice

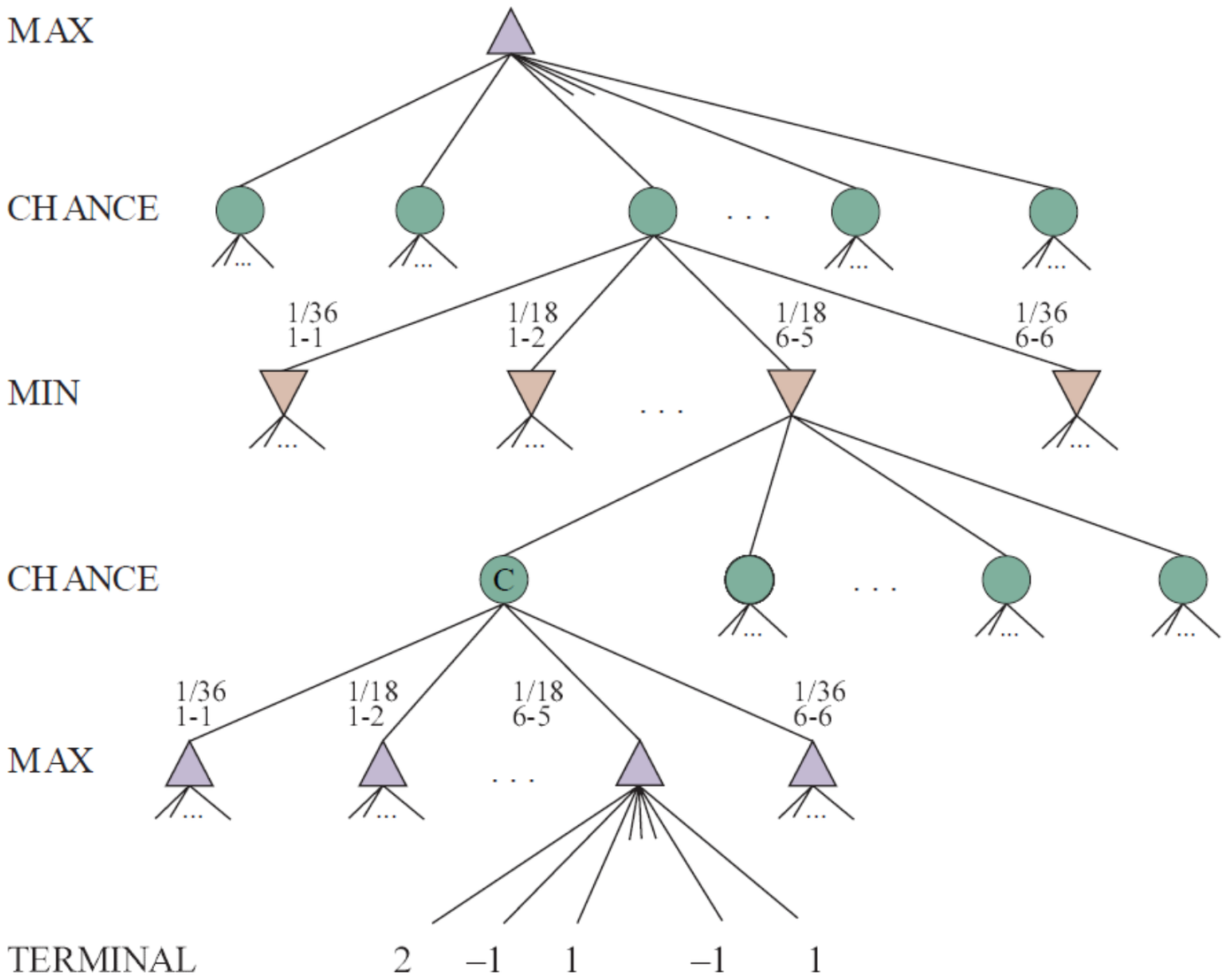
The optimal moves of the playerâs opponent (and also the playerâs next moves) depend on external events

Stochastic Games

Games have unpredictability, like in a real life by including a random element, such as the throwing of dice

The optimal moves of the playerâs opponent (and also the playerâs next moves) depend on external events

The game tree must include chance nodes in addition to MAX and MIN nodes



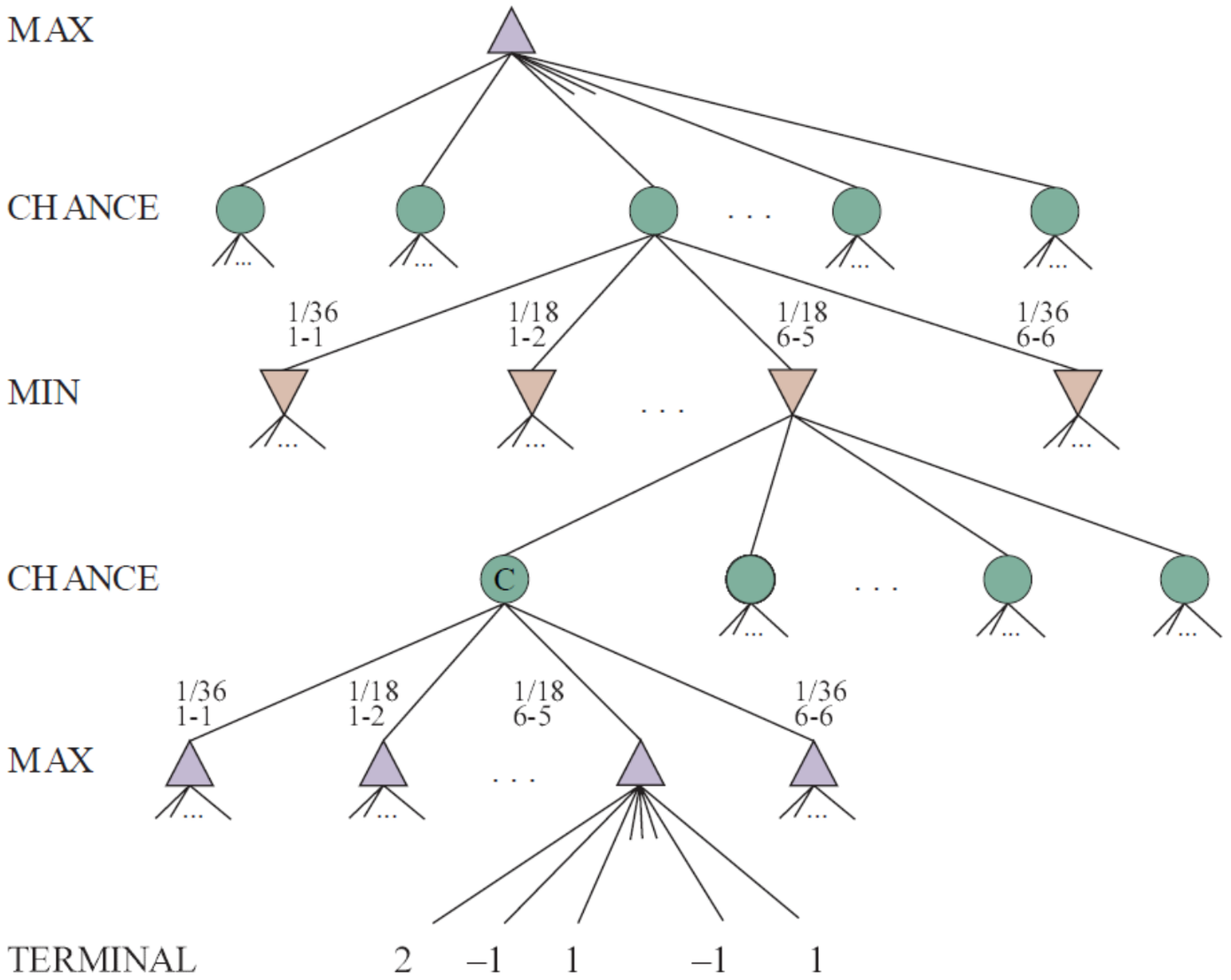
Stochastic Games

Games have unpredictability, like in a real life by including a random element, such as the throwing of dice

The optimal moves of the playerâs opponent (and also the playerâs next moves) depend on external events

The game tree must include chance nodes in addition to MAX and MIN nodes

For chance nodes, we compute the expected value of a position: the average over all possible outcomes of the chance nodes.



Stochastic Games

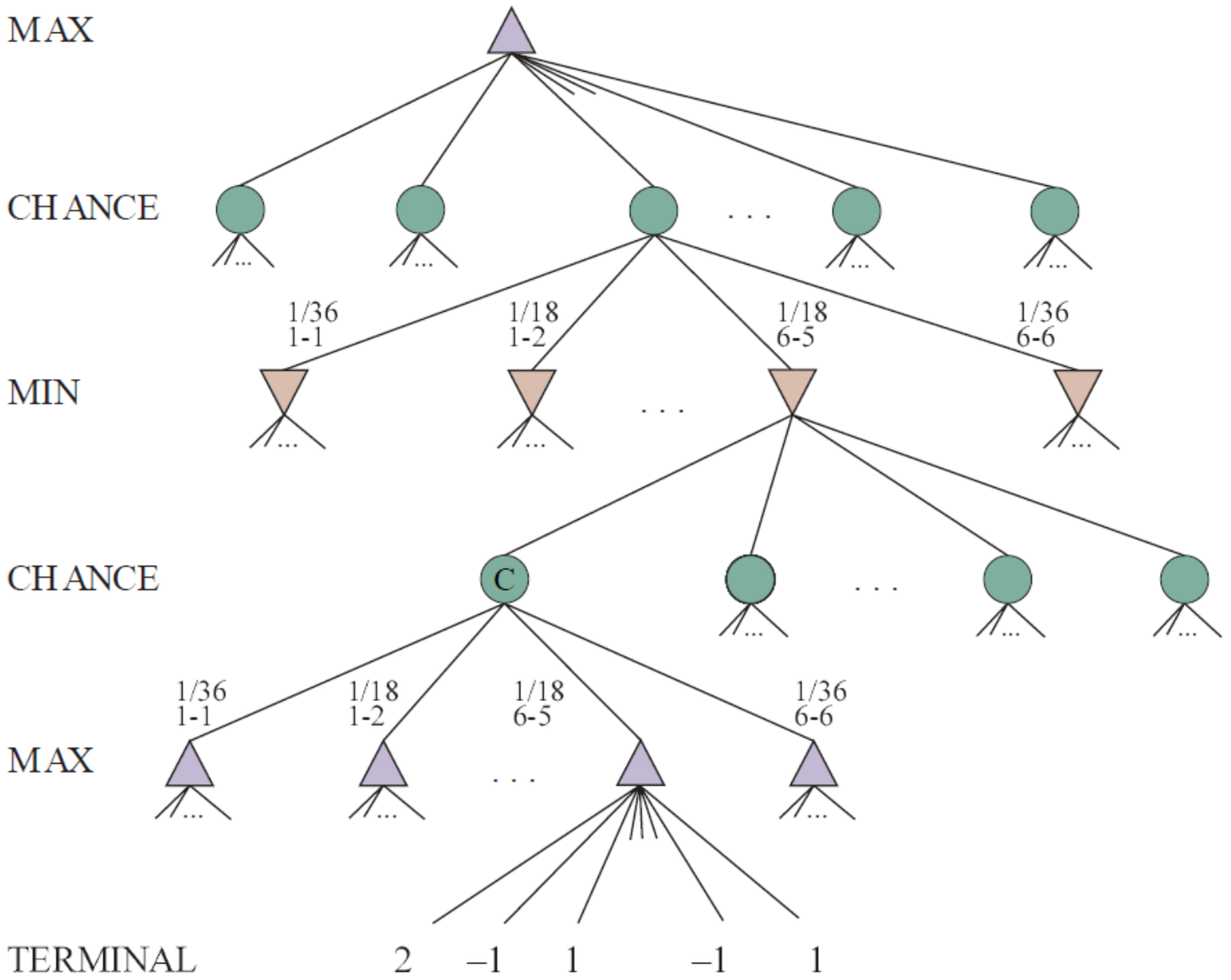
Games have unpredictability, like in a real life by including a random element, such as the throwing of dice

The optimal moves of the playerâs opponent (and also the playerâs next moves) depend on external events

The game tree must include chance nodes in addition to MAX and MIN nodes

For chance nodes, we compute the expected value of a position: the average over all possible outcomes of the chance nodes.

Expectiminimax value for games with chance nodes, a generalization of the minimax value for deterministic games



Partially Observable Games

In games of imperfect information, such as Kriegspiel and poker, optimal play requires reasoning about the current and future belief states of each player. A simple approximation can be obtained by averaging the value of an action over each possible configuration of missing information.

Limitations of Game Search

Calculating optimal decisions in complex games is intractable

Limitations of Game Search

Calculating optimal decisions in complex games is intractable

Both algoritms suffer from fundamental limitation:

Limitations of Game Search

Calculating optimal decisions in complex games is intractable

Both algoritms suffer from fundamental limitation:

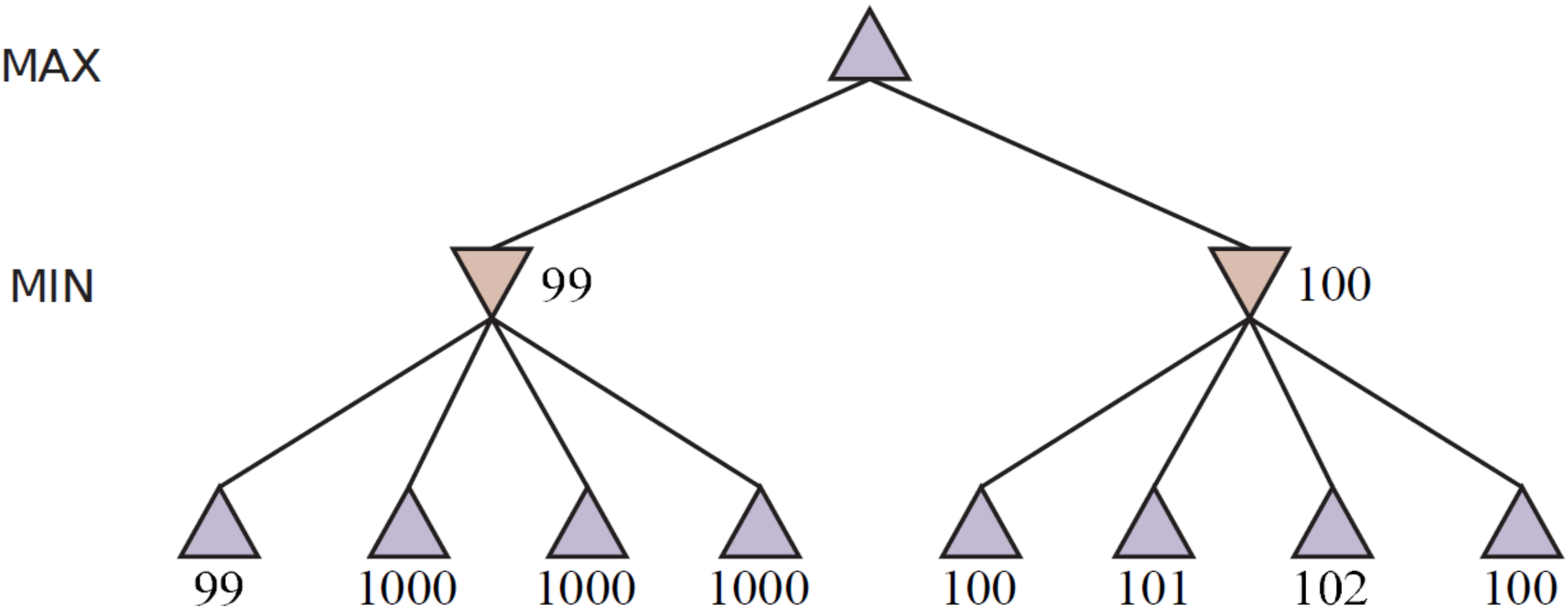
- Vulnerability to errors in the heuristic function

Limitations of Game Search

Calculating optimal decisions in complex games is intractable

Both algoritms suffer from fundamental limitation:

- Vulnerability to errors in the heuristic function



Limitations of Game Search

Calculating optimal decisions in complex games is intractable

Both algoritms suffer from fundamental limitation:

- Vulnerability to errors in the heuristic function
- The algorithms are designed to calculate bounds. A better algorithm would use the idea of the utility of a node expansion to select node that are likely to lead to a better move.

Limitations of Game Search

Calculating optimal decisions in complex games is intractable

Both algoritms suffer from fundamental limitation:

- Vulnerability to errors in the heuristic function
- The algorithms are designed to calculate bounds. A better algorithm would use the idea of the utility of a node expansion to select node that are likely to lead to a better move.
- Reasoning are done at the level of individual moves. Humans play games differently: reasoning at the abstract level

Limitations of Game Search

Calculating optimal decisions in complex games is intractable

Both algoritms suffer from fundamental limitation:

- Vulnerability to errors in the heuristic function
- The algorithms are designed to calculate bounds. A better algorithm would use the idea of the utility of a node expansion to select node that are likely to lead to a better move.
- Reasoning are done at the level of individual moves. Humans play games differently: reasoning at the abstract level
- No ability to incorporate machine learning into the game search process

INFORMATIKA
UNPAR

informatika.unpar.ac.id
informatika@unpar.ac.id
if.unpar
if.unpar