Adversarial Search

Håkon Måløy

September 23, 2021

Games

Games

We will be looking at searches in multiagent environments.

Games

We will be looking at searches in multiagent environments.

How should we represent the other agents?

Games

We will be looking at searches in multiagent environments.

How should we represent the other agents?

- We could consider them just a part of the environment that makes the environment nondeterministic (we miss out on them actually trying to defeat us).

Games

We will be looking at searches in multiagent environments.

How should we represent the other agents?

- We could consider them just a part of the environment that makes the environment nondeterministic (we miss out on them actually trying to defeat us).
- We can explicitly model adversarial agents using adversarial game-tree search.

Games

Games

Modelling using game-tree search.

Games

Modelling using game-tree search.

- Our agent needs to consider:

Games

Modelling using game-tree search.

- Our agent needs to consider:
  - The possible actions of the other agents.

Games

Modelling using game-tree search.

- Our agent needs to consider:
  - The possible actions of the other agents.
  - How these actions can affect the its own welfare.

Games

Games

Agents may interact in two ways:

Games

Agents may interact in two ways:

- Compete

Games

Agents may interact in two ways:

- Compete
- Collaborate

Games

Agents may interact in two ways:

- Compete
- Collaborate

AI game theory is often concerned with a specific subset of games:

Games

Agents may interact in two ways:

- Compete
- Collaborate

AI game theory is often concerned with a specific subset of games:

- Deterministic - Outcome of any action is certain and consistent

Games

Agents may interact in two ways:

- Compete
- Collaborate

AI game theory is often concerned with a specific subset of games:

- Deterministic - Outcome of any action is certain and consistent
- Turn-taking - Each player takes one turn each per round.

Games

Agents may interact in two ways:

- Compete
- Collaborate

AI game theory is often concerned with a specific subset of games:

- Deterministic - Outcome of any action is certain and consistent
- Turn-taking - Each player takes one turn each per round.
- Two-player - Only two players involved...

Games

Agents may interact in two ways:

- Compete
- Collaborate

AI game theory is often concerned with a specific subset of games:

- Deterministic - Outcome of any action is certain and consistent
- Turn-taking - Each player takes one turn each per round.
- Two-player - Only two players involved...
- Zero-sum - The total losses and gains of both agents sums to zero.

Games

Agents may interact in two ways:

- Compete
- Collaborate

AI game theory is often concerned with a specific subset of games:

- Deterministic - Outcome of any action is certain and consistent
- Turn-taking - Each player takes one turn each per round.
- Two-player - Only two players involved...
- Zero-sum - The total losses and gains of both agents sums to zero.
- Perfect information - All participants have full knowledge about their own cost and utility functions and game history.

Formally Defining a Game

- The initial state:

The initial state S_0 specifies how the game is set up at the start.

Formally Defining a Game

- The initial state:
- Which players turn it is to move:

The method TO-MOVE(s) returns the player whose turn it is to move in the state s.

Formally Defining a Game

- The initial state:
- Which players turn it is to move:
- The actions available:

The method ACTIONS(s) returns set of legal moves in state s.

Formally Defining a Game

- The initial state:
- Which players turn it is to move:
- The actions available:
- The transition model:

The transition model describes the outcome of each action: $RESULT(s, a)$ returns the state that results from doing action a in state s.

Formally Defining a Game

- The initial state:
- Which players turn it is to move:
- The actions available:
- The transition model:
- A terminal test:

The method IS-TERMINAL(s) returns true if the game is over and false otherwise. States where the game has ended are called *terminal states*.

Formally Defining a Game

- The initial state:
- Which players turn it is to move:
- The actions available:
- The transition model:
- A terminal test:
- A utility function:

The method UTILITY(s, p) returns the final numeric value to player p when the game ends in terminal state s.

Representing Games

How do we represent the game?

Representing Games

How do we represent the game?

- We call the two players: MAX and MIN.

Representing Games

How do we represent the game?

- We call the two players: MAX and MIN.
- We show the utility value for MAX at each leaf node.

Representing Games

How do we represent the game?

- We call the two players: MAX and MIN.
- We show the utility value for MAX at each leaf node.
- The player **we want to win** starts the game (MAX).

Representing Games

How do we represent the game?

- We call the two players: MAX and MIN.
- We show the utility value for MAX at each leaf node.
- The player **we want to win** starts the game (MAX).
- MAX wants to reach a state with maximum value.

Representing Games

How do we represent the game?

- We call the two players: MAX and MIN.
- We show the utility value for MAX at each leaf node.
- The player **we want to win** starts the game (MAX).
- MAX wants to reach a state with maximum value.
- MIN wants to reach a state with minimum value.

Representing Games

The Game Tree:

Representing Games

The Game Tree:

- In games the search tree is a tree where at each alternating level one of the players has the control/decisions.

Representing Games

The Game Tree:

- In games the search tree is a tree where at each alternating level one of the players has the control/decisions.
- One move involves a decision from each player. Each player decision is called a ply.

Representing Games

The Game Tree:

- In games the search tree is a tree where at each alternating level one of the players has the control/decisions.
- One move involves a decision from each player. Each player decision is called a ply.
- Each leaf in the search tree is assigned a utility value - usually:
  $+1 =$ win
  $-1 =$ lose
  $0 =$ draw

Example: NIM

Game rules:

Example: NIM

Game rules:

- There are x number of objects - e.g., fruits.

Example: NIM

Game rules:

- There are x number of objects - e.g., fruits.
- Each player picks up and eats either one or 2 fruits on their turn.

Example: NIM

Game rules:

- There are x number of objects - e.g., fruits.
- Each player picks up and eats either one or 2 fruits on their turn.
- The player that eats the last one wins.

Example: NIM

Game rules:

- There are x number of objects - e.g., fruits.
- Each player picks up and eats either one or 2 fruits on their turn.
- The player that eats the last one wins.

The game is:

Example: NIM

Game rules:

- There are x number of objects - e.g., fruits.
- Each player picks up and eats either one or 2 fruits on their turn.
- The player that eats the last one wins.

The game is:

- Deterministic.

Example: NIM

Game rules:

- There are x number of objects - e.g., fruits.
- Each player picks up and eats either one or 2 fruits on their turn.
- The player that eats the last one wins.

The game is:

- Deterministic.
- Two-Player.

Example: NIM

Game rules:

- There are x number of objects - e.g., fruits.
- Each player picks up and eats either one or 2 fruits on their turn.
- The player that eats the last one wins.

The game is:

- Deterministic.
- Two-Player.
- Turn-Taking.

Example: NIM

Game rules:

- There are x number of objects - e.g., fruits.
- Each player picks up and eats either one or 2 fruits on their turn.
- The player that eats the last one wins.

The game is:

- Deterministic.
- Two-Player.
- Turn-Taking.
- Zero-Sum.

Example: NIM

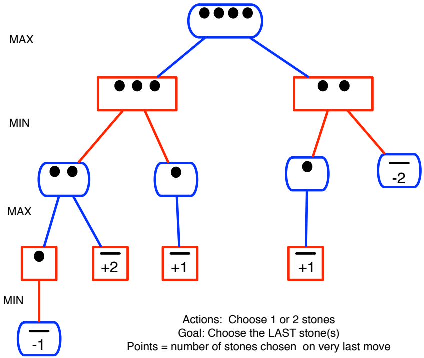
Game rules:

- There are x number of objects - e.g., fruits.
- Each player picks up and eats either one or 2 fruits on their turn.
- The player that eats the last one wins.

The game is:

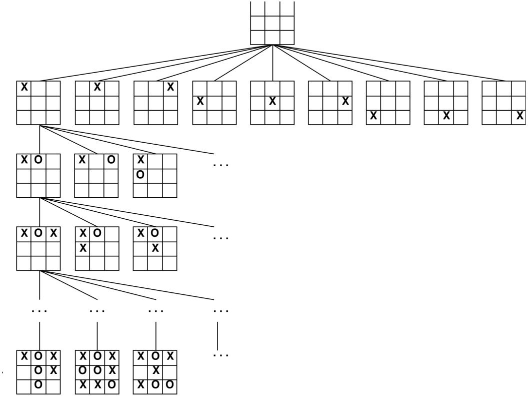
- Deterministic.
- Two-Player.
- Turn-Taking.
- Zero-Sum.
- Perfect Information.

Example: NIM

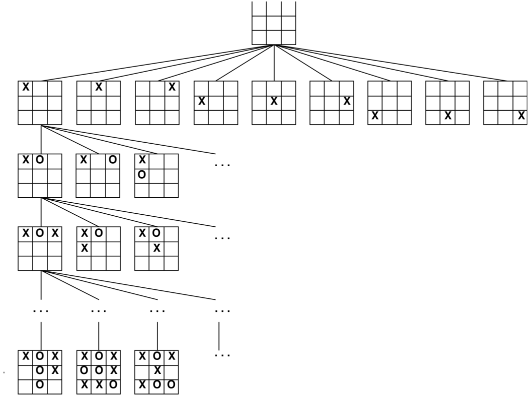


Actions: Choose 1 or 2 stones
Goal: Choose the LAST stone(s)
Points = number of stones chosen on very last move

Example: Tic-Tac-Toe

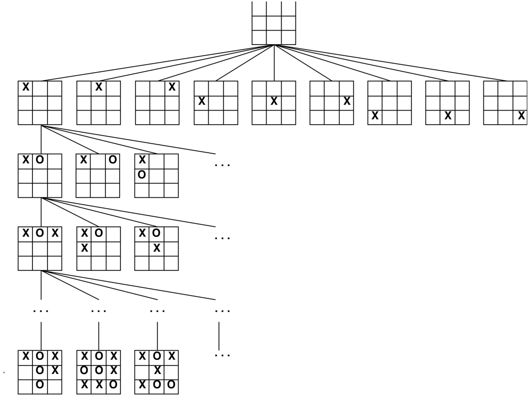


Example: Tic-Tac-Toe



What is the best strategy?

Example: Tic-Tac-Toe



What is the best strategy?

In tic-tac-toe, when both players play optimally, neither player will win.

The Optimal Game Strategy

The Optimal Game Strategy

- MAX wants to find a sequence of actions that leads to a win, but MIN has something to say about it.

The Optimal Game Strategy

- MAX wants to find a sequence of actions that leads to a win, but MIN has something to say about it.
- MAX must have a strategy with a response to each of MIN's possible moves.

The Optimal Game Strategy

- MAX wants to find a sequence of actions that leads to a win, but MIN has something to say about it.
- MAX must have a strategy with a response to each of MIN's possible moves.

The optimal strategy is one that leads to outcomes **at least as good** as any other strategy when playing an **infallible** opponent $\Rightarrow$ The MiniMax Value.

The MiniMax Value

The MiniMax Value

The MiniMax value of a node (MINIMAX(n)) is the **utility** for MAX of being in the corresponding state, assuming that both players play optimally from there on to the end of the game.

The MiniMax Value

The MiniMax value of a node (MINIMAX(n)) is the **utility** for MAX of being in the corresponding state, assuming that both players play optimally from there on to the end of the game.

- In **terminal states** the minimax value is just the utility of that state.

The MiniMax Value

The MiniMax value of a node (MINIMAX(n)) is the **utility** for MAX of being in the corresponding state, assuming that both players play optimally from there on to the end of the game.

- In **terminal states** the minimax value is just the utility of that state.
- If given a choice, MAX will always move to a state with maximum value.

The MiniMax Value

The MiniMax value of a node (MINIMAX(n)) is the **utility** for MAX of being in the corresponding state, assuming that both players play optimally from there on to the end of the game.

- In **terminal states** the minimax value is just the utility of that state.
- If given a choice, MAX will always move to a state with maximum value.
- MIN will always move to a state with minimum value.

The MiniMax Value

The MiniMax value of a node (MINIMAX(n)) is the **utility** for MAX of being in the corresponding state, assuming that both players play optimally from there on to the end of the game.

- In **terminal states** the minimax value is just the utility of that state.
- If given a choice, MAX will always move to a state with maximum value.
- MIN will always move to a state with minimum value.

$$MINIMAX(s) = \begin{cases} \text{UTILITY(s)} & \text{if TERMINAL-TEST(s)} \\ max_{a \in Actions(s)} \text{MINIMAX(RESULT(s, a))} & \text{if PLAYER(s) = MAX} \\ min_{a \in Actions(s)} \text{MINIMAX(RESULT(s, a))} & \text{if PLAYER(s) = MIN} \end{cases} \tag{1}$$

Adversarial Search - The Process

Adversarial Search - The Process

- From the start state, generate a search tree in a depth-first manner, alternating between MAX's and MIN's possible moves.

Adversarial Search - The Process

- From the start state, generate a search tree in a depth-first manner, alternating between MAX's and MIN's possible moves.
- Use UTILITY functions to evaluate the promise of the leaf nodes; then propagate values upwards.

Adversarial Search - The Process

- From the start state, generate a search tree in a depth-first manner, alternating between MAX's and MIN's possible moves.
- Use UTILITY functions to evaluate the promise of the leaf nodes; then propagate values upwards.
- Return to the root and choose the action, A_{best}, leading to the highest-rated child state, S_{best}.

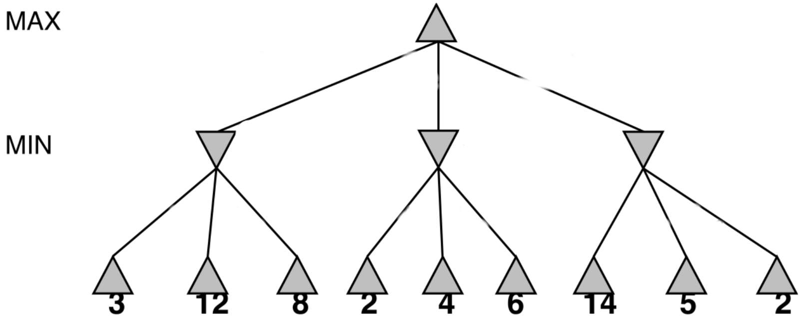
Adversarial Search - The Process

- From the start state, generate a search tree in a depth-first manner, alternating between MAX's and MIN's possible moves.
- Use UTILITY functions to evaluate the promise of the leaf nodes; then propagate values upwards.
- Return to the root and choose the action, A_{best}, leading to the highest-rated child state, S_{best}.
- Apply A_{best} to the current game state, producing S_{best}. Wait for the opponent to choose an action, which then produces the new game state, S_{new}.

Adversarial Search - The Process

- From the start state, generate a search tree in a depth-first manner, alternating between MAX's and MIN's possible moves.
- Use UTILITY functions to evaluate the promise of the leaf nodes; then propagate values upwards.
- Return to the root and choose the action, A_{best}, leading to the highest-rated child state, S_{best}.
- Apply A_{best} to the current game state, producing S_{best}. Wait for the opponent to choose an action, which then produces the new game state, S_{new}.
- Now, from S_{new} node choose and apply the action that leads to the highest-rated child state.

Adversarial Search - Intuitive Example



Adversarial Search - The Process

Let's formalize what we are doing in an algorithm $\Rightarrow$ The MiniMax Algorithm.

The MiniMax Algorithm

The MiniMax Algorithm

function MINIMAX-SEARCH(*game*, *state*)
    player ← *game*.TO-MOVE(*state*)
    value, *move* ← MAX-VALUE(*game*, *state*)
    return *move*

The MiniMax Algorithm

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

```
function MAX-VALUE(game, state)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← −∞
    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
```

The MiniMax Algorithm

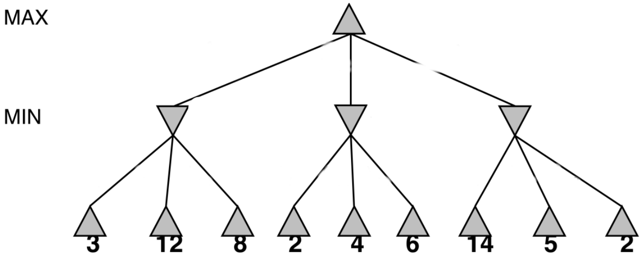
```
function MINIMAX-SEARCH(game, state)
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move
```

```
function MAX-VALUE(game, state)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← −∞
    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)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← +∞
    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
```

Adversarial Search - MiniMax Algorithm Example

```
function MINIMAX-SEARCH(game, state)
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move
function MAX-VALUE(game, state)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← −∞
    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)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← +∞
    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
```



Properties of the MiniMax Algorithm

- Complete: Only if tree is finite.

Properties of the MiniMax Algorithm

- Complete: Only if tree is finite.
- Optimal: Yes, against an optimal opponent.

Properties of the MiniMax Algorithm

- Complete: Only if tree is finite.
- Optimal: Yes, against an optimal opponent.
- Time complexity: $O(b^m)$.

Properties of the MiniMax Algorithm

- Complete: Only if tree is finite.
- Optimal: Yes, against an optimal opponent.
- Time complexity: $O(b^m)$.
- Space complexity: $O(bm)$ (depth-first exploration).

Properties of the MiniMax Algorithm

- Complete: Only if tree is finite.
- Optimal: Yes, against an optimal opponent.
- Time complexity: $O(b^m)$.
- Space complexity: $O(bm)$ (depth-first exploration).

- Chess: $b \approx 35$, $m \approx 100$ for "reasonable" games
  $\Rightarrow$ exact solution completely infeasible.

Properties of the MiniMax Algorithm

- Complete: Only if tree is finite.
- Optimal: Yes, against an optimal opponent.
- Time complexity: $O(b^m)$.
- Space complexity: $O(bm)$ (depth-first exploration).

- Chess: $b \approx 35$, $m \approx 100$ for "reasonable" games
  $\Rightarrow$ exact solution completely infeasible.
- But do we need to explore every path?

Properties of the MiniMax Algorithm

- Complete: Only if tree is finite.
- Optimal: Yes, against an optimal opponent.
- Time complexity: $O(b^m)$.
- Space complexity: $O(bm)$ (depth-first exploration).

- Chess: $b \approx 35$, $m \approx 100$ for "reasonable" games
  $\Rightarrow$ exact solution completely infeasible.
- But do we need to explore every path?
- NO

The MiniMax Algorithm

The MiniMax algorithm explores many paths that we know cannot improve the utility in the view of MAX, can we avoid this?

The MiniMax Algorithm

The MiniMax algorithm explores many paths that we know cannot improve the utility in the view of MAX, can we avoid this?

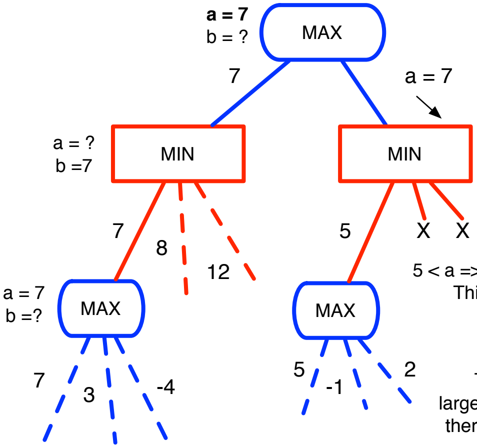
- Yes. If we keep track of the best values we have encountered, we can stop the search down a branch when we know a better value cannot be found.

The MiniMax Algorithm

The MiniMax algorithm explores many paths that we know cannot improve the utility in the view of MAX, can we avoid this?

- Yes. If we keep track of the best values we have encountered, we can stop the search down a branch when we know a better value cannot be found.
- This is called pruning.

Alpha-Beta Pruning



5 < a => Stop generating children.
This node's value cannot
win at the root.
Prune it!

There could be many
large subtrees down here, but
there is no need to generate
them, since they
cannot affect the choice made
at the root.

Alpha and Beta

Alpha

Beta

Alpha and Beta

Alpha

- A modifiable property of a MAX node.

Beta

Alpha and Beta

Alpha

- A modifiable property of a MAX node.
- Indicates whenever a **larger** evaluation is returned from a child MIN node.

Beta

Alpha and Beta

Alpha

- A modifiable property of a MAX node.
- Indicates whenever a **larger** evaluation is returned from a child MIN node.
- Used for pruning MIN nodes.

Beta

Alpha and Beta

Alpha

- A modifiable property of a MAX node.
- Indicates whenever a **larger** evaluation is returned from a child MIN node.
- Used for pruning MIN nodes.

Beta

- A modifiable property of a MIN node.

Alpha and Beta

Alpha

- A modifiable property of a MAX node.
- Indicates whenever a **larger** evaluation is returned from a child MIN node.
- Used for pruning MIN nodes.

Beta

- A modifiable property of a MIN node.
- Indicates whenever a **smaller** evaluation is returned from a child MAX node.

Alpha and Beta

Alpha

- A modifiable property of a MAX node.
- Indicates whenever a **larger** evaluation is returned from a child MIN node.
- Used for pruning MIN nodes.

Beta

- A modifiable property of a MIN node.
- Indicates whenever a **smaller** evaluation is returned from a child MAX node.
- Used for pruning MAX nodes.

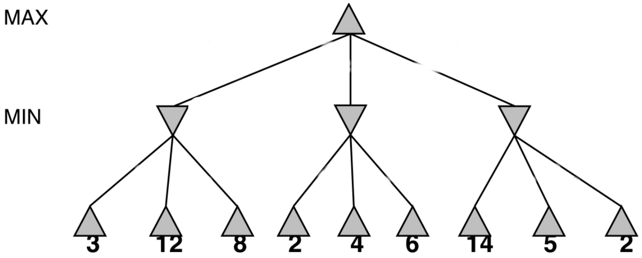
Both Alpha and Beta are passed between the MIN and MAX nodes.

The α-β algorithm

```
function ALPHA-BETA-SEARCH(game, state)
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, −∞, +∞)
    return move
function MAX-VALUE(game, state, α, β)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← −∞
    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)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← +∞
    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
```

α-β Pruning Example

```
function ALPHA-BETA-SEARCH(game, state)
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, −∞, +∞)
    return move
function MAX-VALUE(game, state, α, β)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← −∞
    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)
    if game.TERMINAL-STATE(state) then
        return game.UTILITY(state, player), null
    v, move ← +∞
    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
```



Properties of α and β

- Pruning does not affect final result.

Properties of α and β

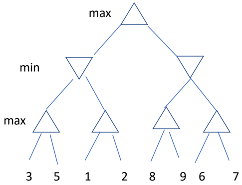
- Pruning does not affect final result.
- Good move ordering improves effectiveness of pruning.

Properties of α and β

- Pruning does not affect final result.
- Good move ordering improves effectiveness of pruning.
- Which nodes can be pruned in the following example:

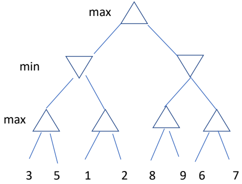
Properties of α and β

- Pruning does not affect final result.
- Good move ordering improves effectiveness of pruning.
- Which nodes can be pruned in the following example:



Properties of α and β

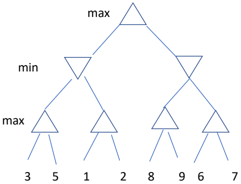
- Pruning does not affect final result.
- Good move ordering improves effectiveness of pruning.
- Which nodes can be pruned in the following example:



- With "perfect ordering," time complexity $=$ \$O(b$^{m/2}$)\$\ $\Rightarrow$ *doubles* solvable depth.

Properties of α and β

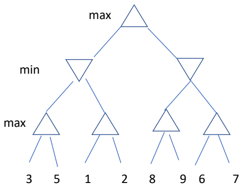
- Pruning does not affect final result.
- Good move ordering improves effectiveness of pruning.
- Which nodes can be pruned in the following example:



- With "perfect ordering," time complexity = \$O($b^{m/2}$)\$\ $\Rightarrow$ *doubles* solvable depth.
- A simple example of the value of reasoning about which computations are relevant (a form of).

Properties of α and β

- Pruning does not affect final result.
- Good move ordering improves effectiveness of pruning.
- Which nodes can be pruned in the following example:



- With "perfect ordering," time complexity = $O(b^{m/2})$\ $\Rightarrow$ *doubles* solvable depth.
- A simple example of the value of reasoning about which computations are relevant (a form of).
- Unfortunately, 35^{50} (e.g. chess) is still impossible!

Resource Limits

- Use CUTOFF-TEST instead of TERMINAL-TEST
  e.g., depth limit

Resource Limits

- Use CUTOFF-TEST instead of TERMINAL-TEST
  e.g., depth limit
- Use EVAL instead of UTILITY
  i.e., a heuristic *evaluation function* that estimates desirability of position

Evaluation Functions

Evaluation Functions

A evaluation function returns an estimate of the expected utility of the game in a given position. An inaccurate evaluation function can guide agent into horrible states.

Evaluation Functions

A evaluation function returns an estimate of the expected utility of the game in a given position. An inaccurate evaluation function can guide agent into horrible states.

- Properties of a good evaluation function:

Evaluation Functions

A evaluation function returns an estimate of the expected utility of the game in a given position. An inaccurate evaluation function can guide agent into horrible states.

- Properties of a good evaluation function:
  - It should order terminal states the same way as the true utility Function.

Evaluation Functions

A evaluation function returns an estimate of the expected utility of the game in a given position. An inaccurate evaluation function can guide agent into horrible states.

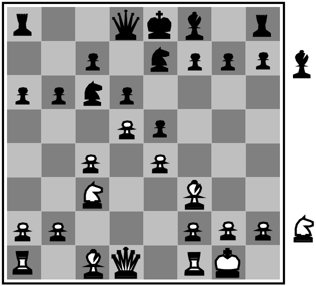
- Properties of a good evaluation function:
  - It should order terminal states the same way as the true utility Function.
  - Computation must not take too long (that's the whole point remember).

Evaluation Functions

A evaluation function returns an estimate of the expected utility of the game in a given position. An inaccurate evaluation function can guide agent into horrible states.

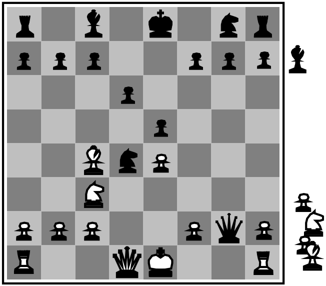
- Properties of a good evaluation function:
  - It should order terminal states the same way as the true utility Function.
  - Computation must not take too long (that's the whole point remember).
  - For non-terminal states, the evaluation function should be strongly correlated with the true chance of winning.

Evaluation Functions



Black to move

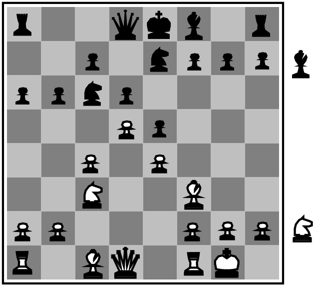
White slightly better



White to move

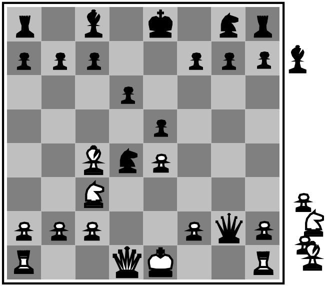
Black winning

Evaluation Functions



Black to move

White slightly better

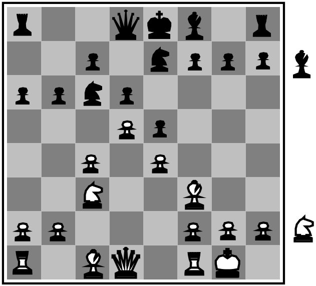


White to move

Black winning

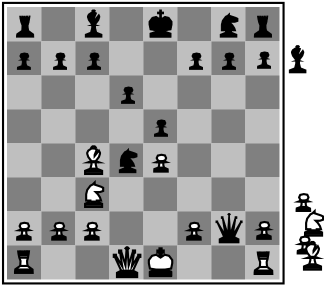
For chess, typically a linear weighted sum of features
$\mathrm{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) $-$ (number of black queens),
etc.

Evaluation Functions



Black to move

White slightly better



White to move

Black winning

For chess, typically a linear weighted sum of features

$\mathrm{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$

e.g., $w_1 = 9$ with

$f_1(s) =$ (number of white queens) $-$ (number of black queens),

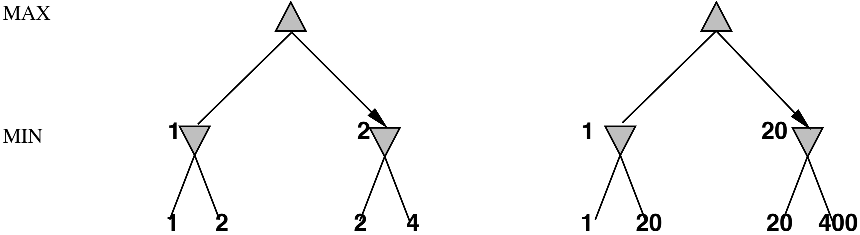
etc.

However, we can also learn the evaluation function (AlphaZero)

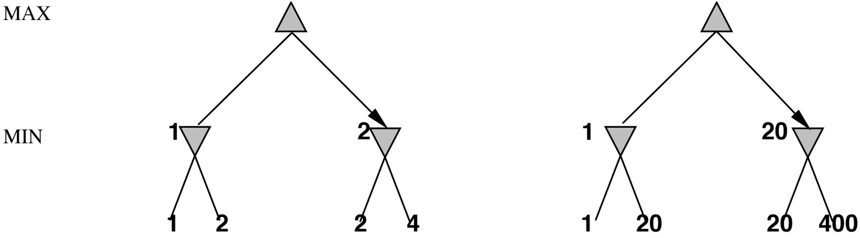
Cutting Off Search

```
function ALPHA-BETA-SEARCH-CUTOFF(game, state)
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, −∞, +∞)
    return move
function MAX-VALUE(game, state, α, β)
    if game.IS-CUTOFF(state, depth) then
        return game.EVAL(state, player), null
    v, move ← −∞
    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)
    if game.IS-CUTOFF(state, depth) then
        return game.EVAL(state, player), null
    v, move ← +∞
    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
```

Digression: Exact Values Don't Matter

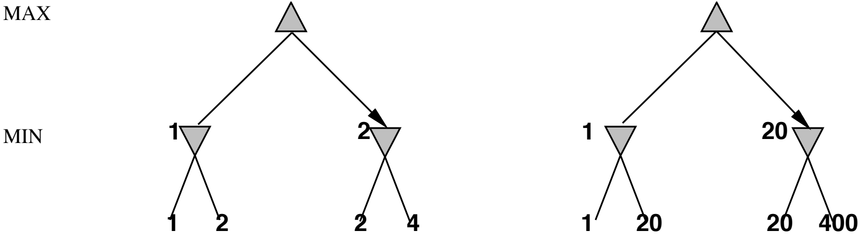


Digression: Exact Values Don't Matter



- Behaviour is preserved under any *monotonic* transformation of EVAL (the evaluation function doesn't have to be linearly correlated with the true chances of winning)

Digression: Exact Values Don't Matter



- Behaviour is preserved under any *monotonic* transformation of EVAL (the evaluation function doesn't have to be linearly correlated with the true chances of winning)
- Only the order matters:
  payoff in deterministic games acts as an ordinal utility function

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Othello: human champions refuse to compete against computers, who are too good.

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.
- Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.
- Othello: human champions refuse to compete against computers, who are too good.
- Go ($b > 300$): human champions refused to compete against computers, who were too bad - until 2015. In 2017 Future of Go Summit, AlphaGo beat Ke Jie, the world No.1 ranked player at the time. Some players have even decided to stop playing the game...

Stochastic (Nondeterministic) Games

In stochastic games, chance introduced by dice, card-shuffling etc.

Stochastic (Nondeterministic) Games

In stochastic games, chance introduced by dice, card-shuffling etc.

- We use chance nodes to represent chance in stochastic games:

Stochastic (Nondeterministic) Games

In stochastic games, chance introduced by dice, card-shuffling etc.

- We use chance nodes to represent chance in stochastic games:
  - Each branch represents a possible outcome (6 branches for a dice).

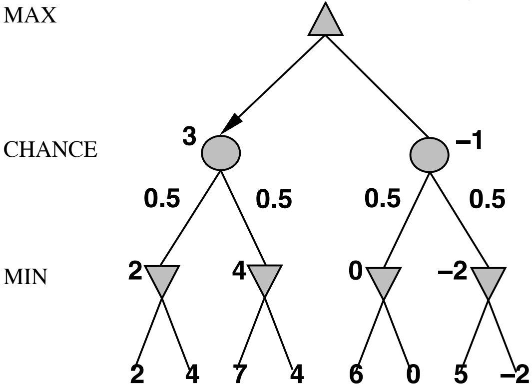
Stochastic (Nondeterministic) Games

In stochastic games, chance introduced by dice, card-shuffling etc.

- We use chance nodes to represent chance in stochastic games:
  - Each branch represents a possible outcome (6 branches for a dice).
  - Branches are labelled with the associated probability (1/6 for dice).

Stochastic Games in General

Simplified example with coin-flipping:



Algorithm for Nondeterministic Games

Expectiminimax gives perfect play

Algorithm for Nondeterministic Games

Expectiminimax gives perfect play

Just like Minimax, except we must also handle chance nodes:

Algorithm for Nondeterministic Games

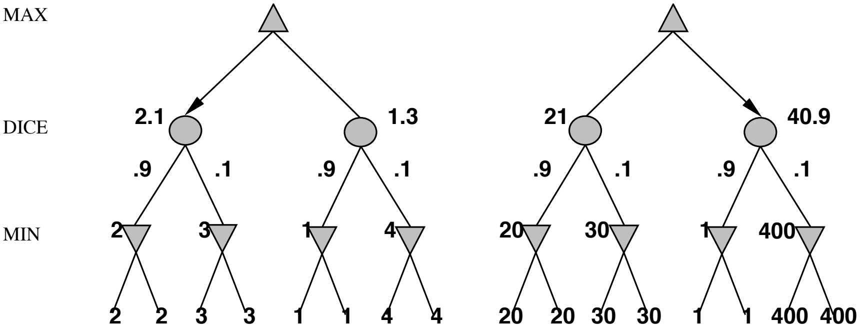
Expectiminimax gives perfect play

Just like Minimax, except we must also handle chance nodes:

. . .

1: **if** *state* is a MAX node **then return**
   the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
2: **if** *state*is a MIN node **then return**
   the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
3: **if** *state* is a chance node **then return**
   sum of ...... EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)

Digression: Exact values DO Matter



EVAL should be proportional to the expected payoff

Summary

Summary

- Games are fun to work on! (and dangerous)

Summary

- Games are fun to work on! (and dangerous)
- They illustrate several important points about AI

Summary

- Games are fun to work on! (and dangerous)
- They illustrate several important points about AI
  - perfection is unattainable $\Rightarrow$ must approximate

Summary

- Games are fun to work on! (and dangerous)
- They illustrate several important points about AI
  - perfection is unattainable $\Rightarrow$ must approximate
  - good idea to think about what to think about

Summary

- Games are fun to work on! (and dangerous)
- They illustrate several important points about AI
  - perfection is unattainable $\Rightarrow$ must approximate
  - good idea to think about what to think about
  - uncertainty constrains the assignment of values to states

Summary

- Games are fun to work on! (and dangerous)
- They illustrate several important points about AI
  - perfection is unattainable $\Rightarrow$ must approximate
  - good idea to think about what to think about
  - uncertainty constrains the assignment of values to states
- Games are to AI as grand prix racing is to automobile design