

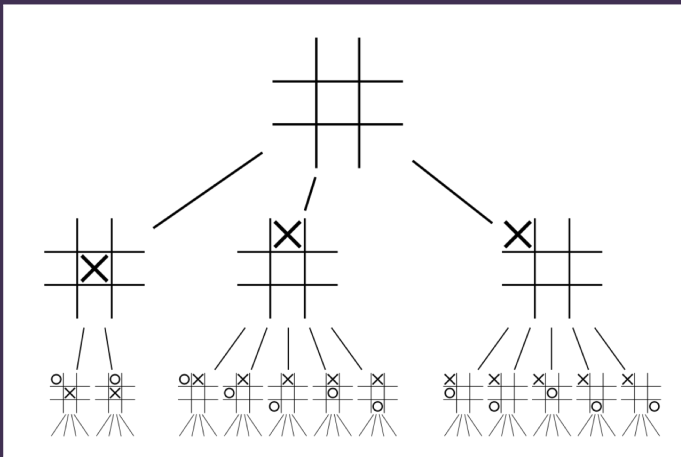
COMP250: Artificial Intelligence

5: Game Tree Search

Minimax search



Game trees



Minimax

Minimax

- ▶ Terminal game states have a **value**

Minimax

- ▶ Terminal game states have a **value**
 - ▶ E.g. +1 for a win, -1 for a loss, 0 for a draw

Minimax

- ▶ Terminal game states have a **value**
 - ▶ E.g. +1 for a win, -1 for a loss, 0 for a draw
- ▶ I want to **maximise** the value

Minimax

- ▶ Terminal game states have a **value**
 - ▶ E.g. +1 for a win, -1 for a loss, 0 for a draw
- ▶ I want to **maximise** the value
- ▶ My opponent wants to **minimise** the value

Minimax

- ▶ Terminal game states have a **value**
 - ▶ E.g. +1 for a win, -1 for a loss, 0 for a draw
- ▶ I want to **maximise** the value
- ▶ My opponent wants to **minimise** the value
- ▶ Therefore I want to **maximise** the **minimum** value my opponent can achieve

Minimax

- ▶ Terminal game states have a **value**
 - ▶ E.g. +1 for a win, -1 for a loss, 0 for a draw
- ▶ I want to **maximise** the value
- ▶ My opponent wants to **minimise** the value
- ▶ Therefore I want to **maximise** the **minimum** value my opponent can achieve
- ▶ This is generally only true for **two-player zero-sum** games

Minimax search

Minimax search

- ▶ Recursively defines a **value** for non-terminal game states

Minimax search

- ▶ Recursively defines a **value** for non-terminal game states
- ▶ Consider each possible “next state”, i.e. each possible move

Minimax search

- ▶ Recursively defines a **value** for non-terminal game states
- ▶ Consider each possible “next state”, i.e. each possible move
- ▶ If it's my turn, the value is the **maximum** value over next states

Minimax search

- ▶ Recursively defines a **value** for non-terminal game states
- ▶ Consider each possible “next state”, i.e. each possible move
- ▶ If it's my turn, the value is the **maximum** value over next states
- ▶ If it's my opponent's turn, the value is the **minimum** value over next states

Minimax search – example

Minimax search pseudocode

procedure MINIMAX(state, currentPlayer)

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
    bestValue =  $-\infty$ 
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
    bestValue =  $-\infty$ 
    for each possible nextState do
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
    bestValue =  $-\infty$ 
    for each possible nextState do
      v = MINIMAX(nextState, 3 - currentPlayer)
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
    bestValue =  $-\infty$ 
    for each possible nextState do
      v = MINIMAX(nextState, 3 - currentPlayer)
      bestValue = MAX(bestValue, v)
```


Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
    bestValue =  $-\infty$ 
    for each possible nextState do
      v = MINIMAX(nextState, 3 - currentPlayer)
      bestValue = MAX(bestValue, v)
  return bestValue
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
    bestValue =  $-\infty$ 
    for each possible nextState do
      v = MINIMAX(nextState, 3 - currentPlayer)
      bestValue = MAX(bestValue, v)
    return bestValue
  else if currentPlayer = 2 then
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
    bestValue =  $-\infty$ 
    for each possible nextState do
       $v = \text{MINIMAX}(\text{nextState}, 3 - \text{currentPlayer})$ 
      bestValue = MAX(bestValue,  $v$ )
    return bestValue
  else if currentPlayer = 2 then
    bestValue =  $+\infty$ 
    for each possible nextState do
       $v = \text{MINIMAX}(\text{nextState}, 3 - \text{currentPlayer})$ 
      bestValue = MIN(bestValue,  $v$ )
    return bestValue
```

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
  if state is terminal then
    return value of state
  else if currentPlayer = 1 then
    bestValue =  $-\infty$ 
    for each possible nextState do
       $v = \text{MINIMAX}(\text{nextState}, 3 - \text{currentPlayer})$ 
      bestValue = MAX(bestValue,  $v$ )
    return bestValue
  else if currentPlayer = 2 then
    bestValue =  $+\infty$ 
    for each possible nextState do
       $v = \text{MINIMAX}(\text{nextState}, 3 - \text{currentPlayer})$ 
      bestValue = MIN(bestValue,  $v$ )
    return bestValue
```

Stopping early

```
for each possible nextState do  
     $v = \text{MINIMAX}(\text{nextState}, 3 - \text{currentPlayer})$   
     $\text{bestValue} = \text{MAX}(\text{bestValue}, v)$ 
```

Stopping early

for each possible nextState **do**

$v = \text{MINIMAX}(\text{nextState}, 3 - \text{currentPlayer})$

$\text{bestValue} = \text{MAX}(\text{bestValue}, v)$

- State values are always between -1 and $+1$

Stopping early

for each possible nextState **do**

$v = \text{MINIMAX}(\text{nextState}, 3 - \text{currentPlayer})$

$\text{bestValue} = \text{MAX}(\text{bestValue}, v)$

- ▶ State values are always between -1 and $+1$
- ▶ So if we ever have $\text{bestValue} = 1$, we can stop early

Stopping early

for each possible nextState **do**

$v = \text{MINIMAX}(\text{nextState}, 3 - \text{currentPlayer})$

$\text{bestValue} = \text{MAX}(\text{bestValue}, v)$

- ▶ State values are always between -1 and $+1$
- ▶ So if we ever have $\text{bestValue} = 1$, we can stop early
- ▶ Similarly when minimising if $\text{bestValue} = -1$

Using minimax search

Using minimax search

- ▶ To decide what move to play next...

Using minimax search

- ▶ To decide what move to play next...
- ▶ Calculate the minimax value for each move

Using minimax search

- ▶ To decide what move to play next...
- ▶ Calculate the minimax value for each move
- ▶ Choose the move with the maximum score

Using minimax search

- ▶ To decide what move to play next...
- ▶ Calculate the minimax value for each move
- ▶ Choose the move with the maximum score
- ▶ If there are several with the same score, choose one at random

Minimax and game theory

Minimax and game theory

- For a **two-player zero-sum** game with **perfect information** and **sequential moves**

Minimax and game theory

- ▶ For a **two-player zero-sum** game with **perfect information** and **sequential moves**
- ▶ Minimax search will always find a **Nash equilibrium**

Minimax and game theory

- ▶ For a **two-player zero-sum** game with **perfect information** and **sequential moves**
- ▶ Minimax search will always find a **Nash equilibrium**
- ▶ I.e. a minimax player plays **perfectly**

Minimax and game theory

- ▶ For a **two-player zero-sum** game with **perfect information** and **sequential moves**
- ▶ Minimax search will always find a **Nash equilibrium**
- ▶ I.e. a minimax player plays **perfectly**
- ▶ **But...**

Minimax for larger games

Minimax for larger games

- ▶ The game tree for noughts and crosses has only a few thousand states

Minimax for larger games

- ▶ The game tree for noughts and crosses has only a few thousand states
- ▶ Most games are too large to search fully

Minimax for larger games

- ▶ The game tree for noughts and crosses has only a few thousand states
- ▶ Most games are too large to search fully
 - ▶ Connect 4 has $\approx 10^{13}$ states

Minimax for larger games

- ▶ The game tree for noughts and crosses has only a few thousand states
- ▶ Most games are too large to search fully
 - ▶ Connect 4 has $\approx 10^{13}$ states
 - ▶ Chess has $\approx 10^{47}$ states

Heuristics for search



Depth limiting

Depth limiting

- ▶ Standard minimax needs to search all the way to **terminal** (game over) states

Depth limiting

- ▶ Standard minimax needs to search all the way to **terminal** (game over) states
- ▶ **Depth limiting** is a common technique to apply minimax to larger games

Depth limiting

- ▶ Standard minimax needs to search all the way to **terminal** (game over) states
- ▶ **Depth limiting** is a common technique to apply minimax to larger games
- ▶ Still evaluate terminal states as $+1 / 0 / -1$

Depth limiting

- ▶ Standard minimax needs to search all the way to **terminal** (game over) states
- ▶ **Depth limiting** is a common technique to apply minimax to larger games
- ▶ Still evaluate terminal states as $+1 / 0 / -1$
- ▶ For nonterminal states at depth d , apply a heuristic evaluation instead of searching deeper

Depth limiting

- ▶ Standard minimax needs to search all the way to **terminal** (game over) states
- ▶ **Depth limiting** is a common technique to apply minimax to larger games
- ▶ Still evaluate terminal states as $+1 / 0 / -1$
- ▶ For nonterminal states at depth d , apply a heuristic evaluation instead of searching deeper
- ▶ Evaluation is a number between -1 and $+1$, estimating the probable outcome of the game

1-ply search

1-ply search

- ▶ Case $d = 1$

1-ply search

- ▶ Case $d = 1$
- ▶ For each move, evaluate the state resulting from playing that move

1-ply search

- ▶ Case $d = 1$
- ▶ For each move, evaluate the state resulting from playing that move
- ▶ This is computationally fast

1-ply search

- ▶ Case $d = 1$
- ▶ For each move, evaluate the state resulting from playing that move
- ▶ This is computationally fast
- ▶ Often easier to design a “which state is better” heuristic than to directly design a “which move to play” heuristic

Move ordering

Move ordering

- ▶ Minimax can **stop early** if it sees a value of $+1$ for maximising player or -1 for minimising player

Move ordering

- ▶ Minimax can **stop early** if it sees a value of $+1$ for maximising player or -1 for minimising player
- ▶ Modifications to minimax algorithm (e.g. **alpha-beta pruning**) lead to more of this

Move ordering

- ▶ Minimax can **stop early** if it sees a value of $+1$ for maximising player or -1 for minimising player
- ▶ Modifications to minimax algorithm (e.g. **alpha-beta pruning**) lead to more of this
- ▶ Thus ordering moves from **best to worst** means faster search

Move ordering

- ▶ Minimax can **stop early** if it sees a value of $+1$ for maximising player or -1 for minimising player
- ▶ Modifications to minimax algorithm (e.g. **alpha-beta pruning**) lead to more of this
- ▶ Thus ordering moves from **best to worst** means faster search
- ▶ How do we know which moves are “best” and “worst”? Use a heuristic!

Designing heuristics

Designing heuristics

- ▶ The **playing strength** of depth limited minimax depends heavily on the design of the **heuristic**

Designing heuristics

- ▶ The **playing strength** of depth limited minimax depends heavily on the design of the **heuristic**
- ▶ Good heuristic design requires **in-depth knowledge** of the tactics and strategy of the game

Designing heuristics

- ▶ The **playing strength** of depth limited minimax depends heavily on the design of the **heuristic**
- ▶ Good heuristic design requires **in-depth knowledge** of the tactics and strategy of the game
- ▶ Next time we will look at what we can do if we don't possess such knowledge

Monte Carlo evaluation



From last time

From last time

- ▶ It is useful to have a **heuristic evaluation function** for nonterminal states

From last time

- ▶ It is useful to have a **heuristic evaluation function** for nonterminal states
- ▶ Allows 1-ply search, depth-limited minimax, ...

From last time

- ▶ It is useful to have a **heuristic evaluation function** for nonterminal states
- ▶ Allows 1-ply search, depth-limited minimax, ...
- ▶ Designing a good heuristic requires in-depth knowledge of the game

From last time

- ▶ It is useful to have a **heuristic evaluation function** for nonterminal states
- ▶ Allows 1-ply search, depth-limited minimax, ...
- ▶ Designing a good heuristic requires in-depth knowledge of the game
- ▶ What if you don't have such knowledge?

Expected value

Expected value

- ▶ Let X be a **random variable**

Expected value

- ▶ Let X be a **random variable**
- ▶ Let $p(x)$ be the probability that X has value x

Expected value

- ▶ Let X be a **random variable**
- ▶ Let $p(x)$ be the probability that X has value x
- ▶ Then the **expected value** of X is

$$\sum_x x \cdot p(x)$$

Expected value — example

Expected value — example

- ▶ A slot machine pays out:

Expected value — example

- ▶ A slot machine pays out:
 - ▶ £1 with probability 0.05

Expected value — example

- ▶ A slot machine pays out:
 - ▶ £1 with probability 0.05
 - ▶ £5 with probability 0.03

Expected value — example

- ▶ A slot machine pays out:
 - ▶ £1 with probability 0.05
 - ▶ £5 with probability 0.03
 - ▶ £10 with probability 0.02

Expected value — example

- ▶ A slot machine pays out:
 - ▶ £1 with probability 0.05
 - ▶ £5 with probability 0.03
 - ▶ £10 with probability 0.02
 - ▶ Nothing with probability 0.9

Expected value — example

- ▶ A slot machine pays out:
 - ▶ £1 with probability 0.05
 - ▶ £5 with probability 0.03
 - ▶ £10 with probability 0.02
 - ▶ Nothing with probability 0.9
- ▶ The expected payout is

$$1 \times 0.05 + 5 \times 0.03 + 10 \times 0.02 + 0 \times 0.9 = 0.4$$

i.e. £0.40

Expected value — example

- ▶ A slot machine pays out:
 - ▶ £1 with probability 0.05
 - ▶ £5 with probability 0.03
 - ▶ £10 with probability 0.02
 - ▶ Nothing with probability 0.9
- ▶ The expected payout is

$$1 \times 0.05 + 5 \times 0.03 + 10 \times 0.02 + 0 \times 0.9 = 0.4$$

i.e. £0.40

- ▶ What this means: if you play the slot machine N times, on average you will win $N \times £0.40$

“Randomness” in computing

“Randomness” in computing

- ▶ Digital computers are **deterministic**, so there's no such thing as true randomness

“Randomness” in computing

- ▶ Digital computers are **deterministic**, so there’s no such thing as true randomness
 - ▶ Cryptographically secure systems use an external source of randomness e.g. atmospheric noise, radioactive decay

“Randomness” in computing

- ▶ Digital computers are **deterministic**, so there's no such thing as true randomness
 - ▶ Cryptographically secure systems use an external source of randomness e.g. atmospheric noise, radioactive decay
- ▶ What we actually have are **pseudo-random number generators (PRNGs)**

“Randomness” in computing

- ▶ Digital computers are **deterministic**, so there's no such thing as true randomness
 - ▶ Cryptographically secure systems use an external source of randomness e.g. atmospheric noise, radioactive decay
- ▶ What we actually have are **pseudo-random number generators (PRNGs)**
- ▶ A PRNG is an algorithm which gives an **unpredictable** sequence of numbers based on a **seed**

“Randomness” in computing

- ▶ Digital computers are **deterministic**, so there's no such thing as true randomness
 - ▶ Cryptographically secure systems use an external source of randomness e.g. atmospheric noise, radioactive decay
- ▶ What we actually have are **pseudo-random number generators (PRNGs)**
- ▶ A PRNG is an algorithm which gives an **unpredictable** sequence of numbers based on a **seed**
- ▶ Sequence is **uniformly distributed**, i.e. all numbers have equal probability

“Randomness” in computing

- ▶ Digital computers are **deterministic**, so there's no such thing as true randomness
 - ▶ Cryptographically secure systems use an external source of randomness e.g. atmospheric noise, radioactive decay
- ▶ What we actually have are **pseudo-random number generators (PRNGs)**
- ▶ A PRNG is an algorithm which gives an **unpredictable** sequence of numbers based on a **seed**
- ▶ Sequence is **uniformly distributed**, i.e. all numbers have equal probability
- ▶ Seed is generally based on some source of **entropy**, e.g. system clock, mouse input, electronic noise

Monte Carlo methods

Monte Carlo methods

- In computing, a **Monte Carlo method** is an algorithm based on **averaging over random samples**

Monte Carlo methods

- ▶ In computing, a **Monte Carlo method** is an algorithm based on **averaging over random samples**
- ▶ The **average** over a large number of samples is a good approximation of the **expected value**

Monte Carlo methods

- ▶ In computing, a **Monte Carlo method** is an algorithm based on **averaging over random samples**
- ▶ The **average** over a large number of samples is a good approximation of the **expected value**
- ▶ Used for **quickly approximating** quantities over **large domains**

Monte Carlo methods

- ▶ In computing, a **Monte Carlo method** is an algorithm based on **averaging over random samples**
- ▶ The **average** over a large number of samples is a good approximation of the **expected value**
- ▶ Used for **quickly approximating** quantities over **large domains**
- ▶ Generally designed to **converge in the limit**

Monte Carlo methods

- ▶ In computing, a **Monte Carlo method** is an algorithm based on **averaging over random samples**
- ▶ The **average** over a large number of samples is a good approximation of the **expected value**
- ▶ Used for **quickly approximating** quantities over **large domains**
- ▶ Generally designed to **converge in the limit**
 - ▶ An **infinite** number of samples would give an **exact** answer

Monte Carlo methods

- ▶ In computing, a **Monte Carlo method** is an algorithm based on **averaging over random samples**
- ▶ The **average** over a large number of samples is a good approximation of the **expected value**
- ▶ Used for **quickly approximating** quantities over **large domains**
- ▶ Generally designed to **converge in the limit**
 - ▶ An **infinite** number of samples would give an **exact** answer
 - ▶ As the **number of samples** increases, the **accuracy** of the answer improves

Monte Carlo methods

- ▶ In computing, a **Monte Carlo method** is an algorithm based on **averaging over random samples**
- ▶ The **average** over a large number of samples is a good approximation of the **expected value**
- ▶ Used for **quickly approximating** quantities over **large domains**
- ▶ Generally designed to **converge in the limit**
 - ▶ An **infinite** number of samples would give an **exact** answer
 - ▶ As the **number of samples** increases, the **accuracy** of the answer improves
- ▶ Applications in physics, engineering, finance, weather forecasting, graphics, ...

Monte Carlo evaluation in games

Monte Carlo evaluation in games

- ▶ Based on **random rollouts**

Monte Carlo evaluation in games

- ▶ Based on **random rollouts**

while s is not terminal **do**

 let m be a random legal move from s

 update s by playing m

Monte Carlo evaluation in games

- ▶ Based on **random rollouts**

while s is not terminal **do**

 let m be a random legal move from s
 update s by playing m

- ▶ The **value** of a rollout is the **value** of the terminal state it reaches (i.e. 1 for a win, -1 for a loss, 0 for a draw)

Monte Carlo evaluation in games

- ▶ Based on **random rollouts**

while s is not terminal **do**

 let m be a random legal move from s

 update s by playing m

- ▶ The **value** of a rollout is the **value** of the terminal state it reaches (i.e. 1 for a win, -1 for a loss, 0 for a draw)
- ▶ Averaging gives the **expected value** of the initial state

Monte Carlo evaluation in games

- ▶ Based on **random rollouts**

while s is not terminal **do**

 let m be a random legal move from s

 update s by playing m

- ▶ The **value** of a rollout is the **value** of the terminal state it reaches (i.e. 1 for a win, -1 for a loss, 0 for a draw)
- ▶ Averaging gives the **expected value** of the initial state
- ▶ Higher expected value = more chance of winning

Monte Carlo search

Monte Carlo search

- ▶ **Flat Monte Carlo search:** 1-ply search with Monte Carlo evaluation

Monte Carlo search

- ▶ **Flat Monte Carlo search:** 1-ply search with Monte Carlo evaluation
- ▶ How about minimax with $d > 1$ and Monte Carlo evaluation?

Monte Carlo search

- ▶ **Flat Monte Carlo search:** 1-ply search with Monte Carlo evaluation
- ▶ How about minimax with $d > 1$ and Monte Carlo evaluation?
 - ▶ Minimax assumes the evaluation is **deterministic**, but Monte Carlo is not

Monte Carlo search

- ▶ **Flat Monte Carlo search:** 1-ply search with Monte Carlo evaluation
- ▶ How about minimax with $d > 1$ and Monte Carlo evaluation?
 - ▶ Minimax assumes the evaluation is **deterministic**, but Monte Carlo is not
 - ▶ Not commonly used, mainly because there's something better...

Monte Carlo Tree Search



Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS)

- ▶ Like Monte Carlo evaluation, based on **rollouts**

Monte Carlo Tree Search (MCTS)

- ▶ Like Monte Carlo evaluation, based on **rollouts**
- ▶ First few rollouts are **random**

Monte Carlo Tree Search (MCTS)

- ▶ Like Monte Carlo evaluation, based on **rollouts**
- ▶ First few rollouts are **random**
- ▶ However, statistics from these rollouts are used to **bias** future rollouts

Monte Carlo Tree Search (MCTS)

- ▶ Like Monte Carlo evaluation, based on **rollouts**
- ▶ First few rollouts are **random**
- ▶ However, statistics from these rollouts are used to **bias** future rollouts
- ▶ Bias rollouts towards **plausible** lines of play, i.e. where each player is trying to play the best move

The MCTS algorithm

The MCTS algorithm

- ▶ MCTS builds a **tree**

The MCTS algorithm

- ▶ MCTS builds a **tree**
- ▶ Initially, the tree consists of a single **root node**

The MCTS algorithm

- ▶ MCTS builds a **tree**
- ▶ Initially, the tree consists of a single **root node**
- ▶ Each rollout has four stages:

The MCTS algorithm

- ▶ MCTS builds a **tree**
- ▶ Initially, the tree consists of a single **root node**
- ▶ Each rollout has four stages:
 - ▶ **Selection**: Starting from the root, descend the tree by choosing moves. Continue until we reach a node which does not yet have children for all legal moves.

The MCTS algorithm

- ▶ MCTS builds a **tree**
- ▶ Initially, the tree consists of a single **root node**
- ▶ Each rollout has four stages:
 - ▶ **Selection:** Starting from the root, descend the tree by choosing moves. Continue until we reach a node which does not yet have children for all legal moves.
 - ▶ **Expansion:** Choose a random legal move for which the current node does not have a child node. Add this new node to the tree.

The MCTS algorithm

- ▶ MCTS builds a **tree**
- ▶ Initially, the tree consists of a single **root node**
- ▶ Each rollout has four stages:
 - ▶ **Selection**: Starting from the root, descend the tree by choosing moves. Continue until we reach a node which does not yet have children for all legal moves.
 - ▶ **Expansion**: Choose a random legal move for which the current node does not have a child node. Add this new node to the tree.
 - ▶ **Simulation**: Perform a Monte Carlo rollout, playing random moves until a terminal state is reached.

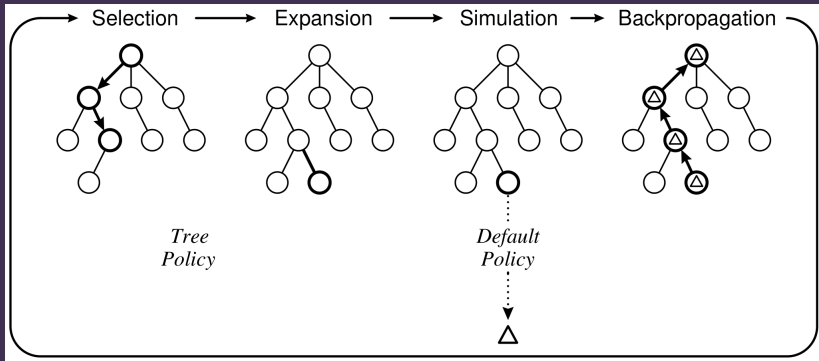
The MCTS algorithm

- ▶ MCTS builds a **tree**
- ▶ Initially, the tree consists of a single **root node**
- ▶ Each rollout has four stages:
 - ▶ **Selection**: Starting from the root, descend the tree by choosing moves. Continue until we reach a node which does not yet have children for all legal moves.
 - ▶ **Expansion**: Choose a random legal move for which the current node does not have a child node. Add this new node to the tree.
 - ▶ **Simulation**: Perform a Monte Carlo rollout, playing random moves until a terminal state is reached.
 - ▶ **Backpropagation**: For each node visited during **selection** and **expansion**, update the node's statistics based on the result of the simulation.

The MCTS algorithm

- ▶ MCTS builds a **tree**
- ▶ Initially, the tree consists of a single **root node**
- ▶ Each rollout has four stages:
 - ▶ **Selection**: Starting from the root, descend the tree by choosing moves. Continue until we reach a node which does not yet have children for all legal moves.
 - ▶ **Expansion**: Choose a random legal move for which the current node does not have a child node. Add this new node to the tree.
 - ▶ **Simulation**: Perform a Monte Carlo rollout, playing random moves until a terminal state is reached.
 - ▶ **Backpropagation**: For each node visited during **selection** and **expansion**, update the node's statistics based on the result of the simulation.
- ▶ Perform many rollouts, then use the statistics at the top level of the tree to choose the best move

The MCTS algorithm



Selection policy

Selection policy

- ▶ Selection must balance:

Selection policy

- ▶ Selection must balance:
 - ▶ **Exploitation** of moves that are known to be good

Selection policy

- ▶ Selection must balance:
 - ▶ **Exploitation** of moves that are known to be good
 - ▶ **Exploration** of moves that have not often been tried

Selection policy

- ▶ Selection must balance:
 - ▶ **Exploitation** of moves that are known to be good
 - ▶ **Exploration** of moves that have not often been tried
- ▶ This can be modelled as a **multi-armed bandit problem**

Multi-armed bandits

Multi-armed bandits

- ▶ We have a row of one-armed bandits (slot machines)

Multi-armed bandits

- ▶ We have a row of one-armed bandits (slot machines)
- ▶ We **do not know** the payout probabilities of any of them, and they're all different

Multi-armed bandits

- ▶ We have a row of one-armed bandits (slot machines)
- ▶ We **do not know** the payout probabilities of any of them, and they're all different
- ▶ How to maximise our winnings?

Multi-armed bandits

- ▶ We have a row of one-armed bandits (slot machines)
- ▶ We **do not know** the payout probabilities of any of them, and they're all different
- ▶ How to maximise our winnings?
- ▶ Again must balance

Multi-armed bandits

- ▶ We have a row of one-armed bandits (slot machines)
- ▶ We **do not know** the payout probabilities of any of them, and they're all different
- ▶ How to maximise our winnings?
- ▶ Again must balance
 - ▶ **Exploitation** of machines that are known to have a high expected payout

Multi-armed bandits

- ▶ We have a row of one-armed bandits (slot machines)
- ▶ We **do not know** the payout probabilities of any of them, and they're all different
- ▶ How to maximise our winnings?
- ▶ Again must balance
 - ▶ **Exploitation** of machines that are known to have a high expected payout
 - ▶ **Exploration** of machines that have not been tried often, to get a better estimate of their expected payout

Upper Confidence Bound (UCB)

Upper Confidence Bound (UCB)

- ▶ For each machine m , record:

Upper Confidence Bound (UCB)

- ▶ For each machine m , record:
 - ▶ n_m : the number of plays of this machine

Upper Confidence Bound (UCB)

- ▶ For each machine m , record:
 - ▶ n_m : the number of plays of this machine
 - ▶ V_m : the total winnings from playing this machine

Upper Confidence Bound (UCB)

- ▶ For each machine m , record:
 - ▶ n_m : the number of plays of this machine
 - ▶ V_m : the total winnings from playing this machine
 - ▶ $n = \sum_m n_m$, total number of plays across all machines

Upper Confidence Bound (UCB)

- ▶ For each machine m , record:
 - ▶ n_m : the number of plays of this machine
 - ▶ V_m : the total winnings from playing this machine
 - ▶ $n = \sum_m n_m$, total number of plays across all machines
- ▶ At each stage, play the machine for which

$$\frac{V_m}{n_m} + c \sqrt{\frac{\log n}{n_m}}$$

is largest

Upper Confidence Bound (UCB)

- ▶ For each machine m , record:
 - ▶ n_m : the number of plays of this machine
 - ▶ V_m : the total winnings from playing this machine
 - ▶ $n = \sum_m n_m$, total number of plays across all machines
- ▶ At each stage, play the machine for which

$$\frac{V_m}{n_m} + c \sqrt{\frac{\log n}{n_m}}$$

is largest

- ▶ $\frac{V_m}{n_m}$ is the **exploitation** part: average payout from this machine so far

Upper Confidence Bound (UCB)

- ▶ For each machine m , record:
 - ▶ n_m : the number of plays of this machine
 - ▶ V_m : the total winnings from playing this machine
 - ▶ $n = \sum_m n_m$, total number of plays across all machines
- ▶ At each stage, play the machine for which

$$\frac{V_m}{n_m} + c\sqrt{\frac{\log n}{n_m}}$$

is largest

- ▶ $\frac{V_m}{n_m}$ is the **exploitation** part: average payout from this machine so far
- ▶ $\sqrt{\frac{\log n}{n_m}}$ is the **exploration** part: large if n_m is small

Upper Confidence Bound (UCB)

- ▶ For each machine m , record:
 - ▶ n_m : the number of plays of this machine
 - ▶ V_m : the total winnings from playing this machine
 - ▶ $n = \sum_m n_m$, total number of plays across all machines
- ▶ At each stage, play the machine for which

$$\frac{V_m}{n_m} + c \sqrt{\frac{\log n}{n_m}}$$

is largest

- ▶ $\frac{V_m}{n_m}$ is the **exploitation** part: average payout from this machine so far
- ▶ $\sqrt{\frac{\log n}{n_m}}$ is the **exploration** part: large if n_m is small
- ▶ c is a parameter for adjusting the balance between exploitation and exploration

UCB demo

`http://orangehelicopter.com/academic/bandits.
html?ucb`

Upper Confidence Bound for Trees (UCT)

Upper Confidence Bound for Trees (UCT)

- ▶ Use UCB as the selection policy

Upper Confidence Bound for Trees (UCT)

- ▶ Use UCB as the selection policy
- ▶ In each node x , record:

Upper Confidence Bound for Trees (UCT)

- ▶ Use UCB as the selection policy
- ▶ In each node x , record:
 - ▶ n_x : the number of visits to this node

Upper Confidence Bound for Trees (UCT)

- ▶ Use UCB as the selection policy
- ▶ In each node x , record:
 - ▶ n_x : the number of visits to this node
 - ▶ V_x : the total value of rollouts through this node

Upper Confidence Bound for Trees (UCT)

- ▶ Use UCB as the selection policy
- ▶ In each node x , record:
 - ▶ n_x : the number of visits to this node
 - ▶ V_x : the total value of rollouts through this node
- ▶ From node p , choose the child q such that

$$\frac{V_q}{n_q} + c \sqrt{\frac{\log n_p}{n_q}}$$

is largest

UCT demo

Benefits of MCTS

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**
- ▶ But if game-specific heuristics are available, they can be used to **enhance** MCTS

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**
- ▶ But if game-specific heuristics are available, they can be used to **enhance** MCTS
- ▶ MCTS is **anytime**

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**
- ▶ But if game-specific heuristics are available, they can be used to **enhance** MCTS
- ▶ MCTS is **anytime**
 - ▶ Can stop it after **any** amount of computation (within reason) and get a reasonably good answer

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**
- ▶ But if game-specific heuristics are available, they can be used to **enhance** MCTS
- ▶ MCTS is **anytime**
 - ▶ Can stop it after **any** amount of computation (within reason) and get a reasonably good answer
 - ▶ Compare with minimax: $O(e^d)$ for depth d

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**
- ▶ But if game-specific heuristics are available, they can be used to **enhance** MCTS
- ▶ MCTS is **anytime**
 - ▶ Can stop it after **any** amount of computation (within reason) and get a reasonably good answer
 - ▶ Compare with minimax: $O(e^d)$ for depth d
- ▶ Does not suffer from **horizon effect**

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**
- ▶ But if game-specific heuristics are available, they can be used to **enhance** MCTS
- ▶ MCTS is **anytime**
 - ▶ Can stop it after **any** amount of computation (within reason) and get a reasonably good answer
 - ▶ Compare with minimax: $O(e^d)$ for depth d
- ▶ Does not suffer from **horizon effect**
 - ▶ Minimax at depth d cannot “see” what happens $d + 1$ moves in the future

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**
- ▶ But if game-specific heuristics are available, they can be used to **enhance** MCTS
- ▶ MCTS is **anytime**
 - ▶ Can stop it after **any** amount of computation (within reason) and get a reasonably good answer
 - ▶ Compare with minimax: $O(e^d)$ for depth d
- ▶ Does not suffer from **horizon effect**
 - ▶ Minimax at depth d cannot “see” what happens $d + 1$ moves in the future
 - ▶ MCTS can build the tree as deep as it likes

Benefits of MCTS

- ▶ “Vanilla” MCTS is **game independent**
- ▶ But if game-specific heuristics are available, they can be used to **enhance** MCTS
- ▶ MCTS is **anytime**
 - ▶ Can stop it after **any** amount of computation (within reason) and get a reasonably good answer
 - ▶ Compare with minimax: $O(e^d)$ for depth d
- ▶ Does not suffer from **horizon effect**
 - ▶ Minimax at depth d cannot “see” what happens $d + 1$ moves in the future
 - ▶ MCTS can build the tree as deep as it likes
 - ▶ Selects which parts of the tree to expand more deeply

Workshop



Workshop

- ▶ Clone `https://github.com/Falmouth-Games-Academy/comp250-workshop-5`
- ▶ This is an implementation of Connect-4 in Python 3 with PyGame
- ▶ Edit `ai_player.py` to implement MCTS
- ▶ Use `http://mcts.ai/code/python.html` as a guide