COMP250: Artificial Intelligence

5: Game Tree Search

Game trees

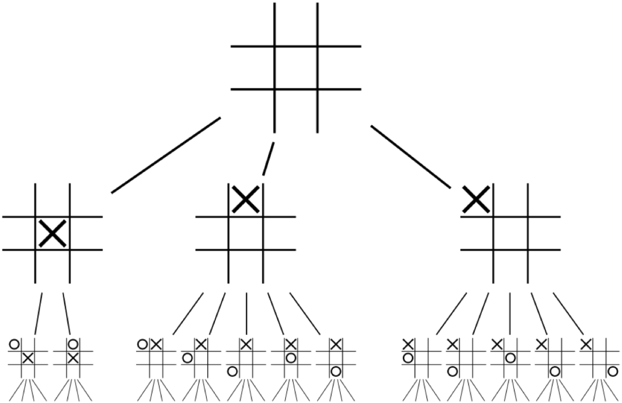
State-action graphs

- We have already discussed **state-action graphs**
  - Nodes: environment states
  - Edges: actions, which cause transitions from one state to another
- Such graphs could contain **cycles** or **transpositions**
  - Cycle: a sequence of actions which takes us from a state s_0 back to the same state s_0
  - Transpositions: two different sequences of actions which both take us from a state s_1 to a state s_2

State-action trees

- Searching graphs with cycles and transpositions is tricky
- Compare with pathfinding — needing to store a closed set of already searched nodes
- So we often use a **tree** representation instead
- Same state may appear multiple times in the tree
- However, there are guaranteed no cycles or transpositions

Example



Minimax search

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

- 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.pptx`

Minimax search pseudocode

```
procedure MINIMAX(state, currentPlayer)
    if state is terminal then
        return value of state
    else if currentPlayer = 1 then
        bestValue = −∞
        for each possible nextState do
            v = MINIMAX(nextState, 3− currentPlayer)
            bestValue = MAX(bestValue, v)
        return bestValue
    else if currentPlayer = 2 then
        bestValue = +∞
        for each possible nextState do
            v = MINIMAX(nextState, 3− currentPlayer)
            bestValue = MIN(bestValue, v)
        return bestValue
```

Stopping early

for each possible nextState **do**
  v = MINIMAX(nextState, 3$-$ currentPlayer)
  bestValue = MAX(bestValue, v)

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

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

- 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

- 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

- 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

- 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
- This is essentially a **utility-based AI**

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

- 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
- What if we don't possess such knowledge?

Monte Carlo evaluation

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, ...

Aside: “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 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

- **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

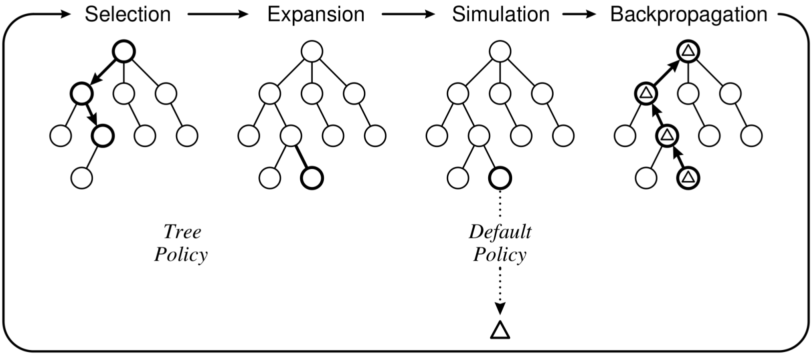
Monte Carlo Tree Search (MCTS)

- 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

- MCTS builds a **subtree** of the game 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 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

- 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)

- 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)

- 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

```
InteractiveDemo.exe
```

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