Adversarial Games

Presented by Yasin Ceran

October 29, 2024

1. Games
2. Optimal Decisions in Games
3. Heuristic ABTS
4. MCTS
5. Stochastic Games

Game theory

There are three stances we can take towards multi-agent environbments:

- When there is a very large number of agents, consider them in the aggregate as an economy, e.g., predicting that increasing demand will cause prices to rise
- We would consider adversarial agents as just a part of the environment, like rain sometimes falls and sometimes does not. In this case, we miss the idea that adversaries are actively trying to defeat us.
- Explicitly model the adversarial agents with the techniques of adversarial game-tree search

Game Theory

Definition: Game theory is the study of mathematical models of strategic interaction between rational decision-makers.

- Focuses on decision-making in competitive environments.
- Players aim to maximize their payoff by choosing optimal strategies.
- Types of games:
  - Zero-sum games: One player's gain is another player's loss.
  - Non-zero-sum games: Both players can win or lose together.
- Common solution concepts:
  - Nash equilibrium
  - Dominant strategies
  - Minimax strategy

Nash Equilibrium

Definition: A Nash equilibrium is a situation in a game where no player can benefit by changing their strategy while the other players keep their strategies unchanged.

- At Nash equilibrium, all players are playing optimally given the strategies of others.
- No player has an incentive to deviate from their strategy.
- A game can have one, multiple, or no Nash equilibria.

Example: Prisoner's Dilemma

- The dominant strategy for both prisoners is to "confess," resulting in a Nash equilibrium where both confess.
- Even though mutual silence would have been better for both, the rational choice leads them to confess.

Dominant Strategies

Definition: A strategy is dominant if, regardless of what the other players do, the strategy earns a player a higher payoff than any other.

- A dominant strategy is the best strategy for a player, no matter what the opponents decide.
- If all players in a game have a dominant strategy, the outcome of the game is determined.
- Not all games have dominant strategies for all players.

Key Points:

- A player with a dominant strategy will always choose it.
- The existence of a dominant strategy simplifies the decision-making process for that player.

Example:

- In the Prisoner's Dilemma, the dominant strategy for each player is to "confess," as it leads to a better outcome for them regardless of the opponent's choice.

Optimal Decisions in Games

Goal: Find the best possible move in a game given the current state.

- **Minimax Algorithm:** A decision rule for minimizing the possible loss in a worst-case scenario.
  - Each player selects the move that maximizes their payoff, assuming the opponent will minimize it.
  - Typically applied in two-player, zero-sum games.
- **Minimax Principle:** Choose the action that maximizes the minimum gain (for the max player) or minimizes the maximum loss (for the min player).
- **Optimality:** The minimax algorithm leads to optimal decisions in deterministic games where all information is available.

Games in General

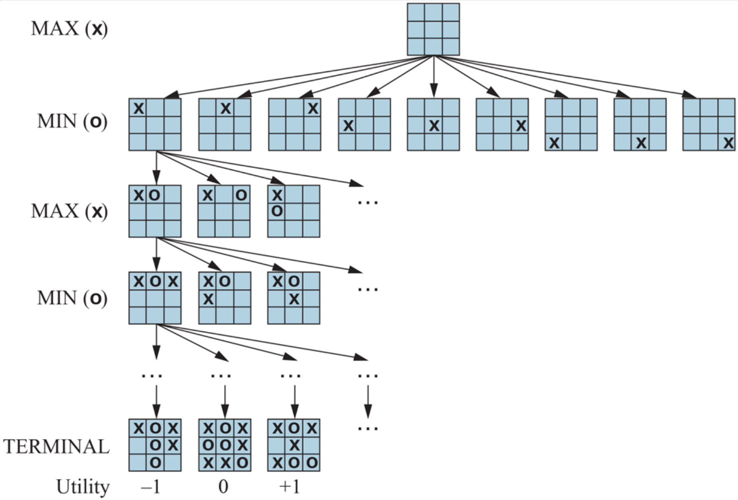
- In multiagent environments, each agent considers the actions of the other agents
- In competitive environments, agents' goals are in conflict. This competition results in adversarial search
- A game of perfect information happens in a deterministic and fully observable environments
- In a game of imperfect information, agents don't see the actions of the other agents but they may know the possibilities
- Pruning allows us to ignore the portions of the search tree that make no difference to the final choice.
- The heuristic evaluation functions allow us to approximate the true utility of a state without doing a complete search.

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Two-Player Zero-Sum Games

- zero-sum means what is good for one player is just as bad for the other
- move is a synonym for "action", and position is a synonym for "state"
- We call our two players 'MAX' and 'MIN'
- S_0: The initial state, which specifies how the game is set up at the start
- $TO - MOVE(s)$: 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
- $IS - TERMINAL(s)$: A terminal test, which is true when the game is over and false otherwise
- $UTILITY(s, p)$: A utility function (objective function or payoff function) defines the final numeric value to player p when the game ends in terminal state s
- game tree is a search tree that follows every sequence of moves all the way to a terminal state

Example: Game tree (2-player, deterministic, turns)

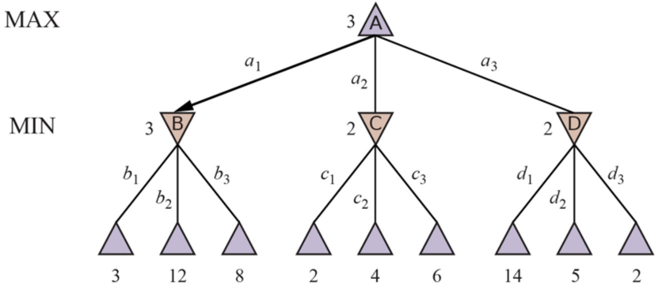


Minimax

- Perfect play for deterministic, perfect-information games
- MAX wants to find a sequence of actions to a win, but MIN has something to say about it; MAX's strategy must be a conditional plan
- Idea: choose move to position with highest

  minimax value= best achievable payoff against best play

E.g., 2-ply game:



Minimax Algorithm

function MINIMAX_SEARCH(*game, state*) **returns** *an action*
  inputs: *state*, current state in game

  player $\leftarrow$ *game.TO_MOVE(state)*
  value, *move* $\leftarrow$ *MAX_VALUE(game, state)*
  return *move*

---

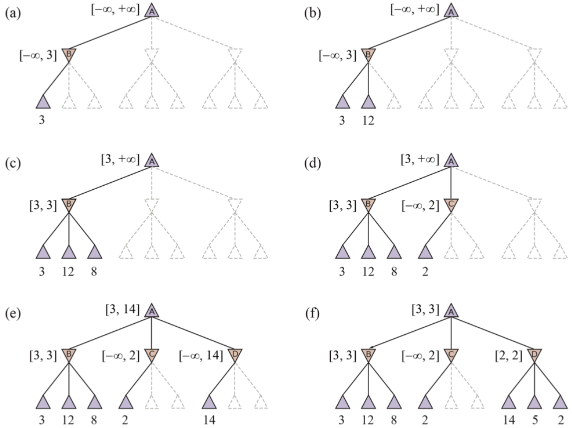
function MAX_VALUE(*game, state*) **returns** a *(utility, move)* pair
  if game.IS_TERMINAL(*state*) **then return** game.UTILITY(*(state, player), null*)
  $v \leftarrow -\infty$
  for each *a* **in** game.ACTIONS(*state*) **do**
    v2, a2 $\leftarrow$ MIN_VALUE(*game, game.*RESULT(*state,a*))
    if $v2 > v$ **then**
      v,move $\leftarrow$ *v2,a*
  return *v, move*

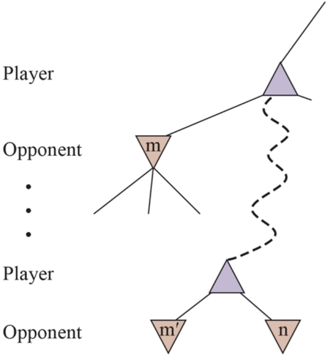
---

function MIN_VALUE(*game, state*) **returns** a *(utility, move)* pair
  if game.IS_TERMINAL(*state*) **then return** game.UTILITY(*(state, player), null*)
  $v \leftarrow +\infty$
  for each *a* **in** game.ACTIONS(*state*) **do**
    v2, a2 $\leftarrow$ MAX_VALUE(*game, game.*RESULT(*state,a*))
    if $v2 < v$ **then**
      v,move $\leftarrow$ *v2,a*
  return *v, move*

α–β Pruning

- The number of game states is exponential in the depth of the tree.
- Pruning large parts of the tree that make no difference to the outcome will reduce the number of states





- If m or m' is better than n for Player, we will never get to n in play.
- α: The value of the best (highest value) choice we have found so far at any choice along the path for *MAX*. Think: $\alpha =$ '*at_least*'
- β: The value of the best (lowest value) choice we have found so far at any choice along the path for *MIN*. Think: $\beta =$ '*at_most*'

Alpha-Beta Algorithm

```
function ALPHA_BETA_SEARCH(game, state) returns an action
   inputs: state, current state in game

    player ← game.TO_MOVE(state)
    value, move ← MAX_VALUE(game, state, −∞, +∞ )
   return move
```

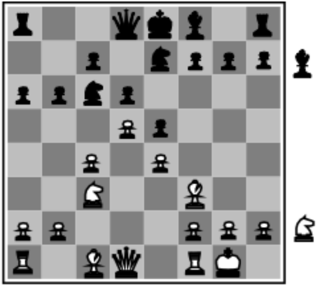
```
function MAX_VALUE(game, state, α, β) returns a (utility, move)
pair
    if game.IS_TERMINAL(state) then return game.UTILITY((state,
player), null)
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN_VALUE(game, game.RESULT(state,a), α, β)
      if v2>v then
         v,move ← v2,a
         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
```

Evaluation Functions

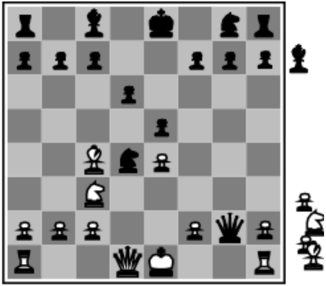
- A heuristic evaluation function $EVAL(s, p)$ returns an estimate of the expected utility of state s to player p
- For terminal states, $EVAL(s, p) = UTILITY(s, p)$
- For non-terminal states, $UTILITY(loss, p) \leq EVAL(s, p) \leq UTILITY(win, p)$
- The computation must not take too long
- The evaluation function should be strongly correlated with the actual chances of winning
- Most evaluation functions work by calculating various features of the state
- Most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value

Evaluation Functions-Chess



Black to move

White slightly better



White to move

Black winning

For chess, typically linear weighted sum of features

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

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens), etc.

Monte Carlo Tree Search (MCTS)

- The basic MCTS strategy does not use a heuristic evaluation function
- The value of a state is estimated as the average utility over a number of simulations of complete games starting from the state
- A simulation chooses moves first for one player, than for the other, repeating until a terminal position is reached
- The get useful information from the simulation, we need playout policy that biases the moves towards the good ones
- Selection is starting at the root of the search tree and choosing a move guided by the selection policy
- Expansion is growing the search tree by a new child of the selected node
- Simulation is performing a playout from the newly generated child node, choosing move for both players according to playout policy
- Back-propagation is using the result of the simulation to update all the search tree nodes going up to the root

MCTS Algorithm

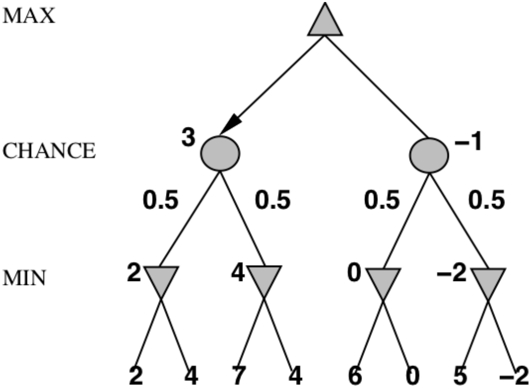
function MONTE_CARLO_TREE_SEARCH(*state*) **returns** *an action*

tree ← NODE(*state*)
while IS_TIME_REMAINING() **do**
  leaf ← SELECT(*tree*)
  child ← EXPAND(*leaf*)
  result ← SIMULATE(*child*)
  function BACK_PROPOGATE(*result, child*) **returns**
return the move in ACTION(*state*) whose node has the highest number of playouts

Stochastic Games in General

In nondeterministic games, chance introduced by dice, card-shuffling

Simplified example with coin-flipping:



Algorithm for Stochastic Games

We can only calculate expected value of a position: the avreage over all possible outcomes of the chance nodes

Expectiminimax gives perfect play

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

EXPECTIMINIMAX(s)=
if IS_TERMINAL(s)
then UTILITY(s, MAX)
if TO_MOVE(s)=MAX
then $\max_a$EXPECTIMINIMAX($RESULT(s, a)$))
if TO_MOVE(s)=MIN
then $\min_a$EXPECTIMINIMAX($RESULT(s, a)$))
if TO_MOVE(s)=CHANCE
then $\sum_r P(r)$EXPECTIMINIMAX($RESULT(s, r)$))

Summary

Games are fun to work on! (and dangerous)

They illustrate several important points about AI

- Game theory and types of games (zero-sum, non-zero-sum)
- Optimal decisions in games using the Minimax algorithm
- Heuristic improvements with Alpha-Beta pruning
- Monte Carlo Tree Search for large and complex search spaces
- Stochastic games and decision-making under uncertainty