CSCE A405/A605 (Adv) Artificial Intelligence

Adversarial Search and Games

Ref: Artificial Intelligence: A Modern Approach, 4th ed by Stuart Russell and Peter Norvig, chapter 5

Instructor: Masoumeh Heidari (mheidari2@Alaska.edu)

UAA College of Engineering
University of Alaska Anchorage

Tic-Tac-Toe

Max

Min

This Photo by Unknown Author is licensed under CC BY-NC

Tic-Tac-Toe

Max

Min

This Photo by Unknown Author is licensed unde CC BY-SA

Tic-Tac-Toe- Max's thoughts

This Photo by Unknown Author is licensed under CC BY-ND

This Photo by Unknown Author is licensed under CC BY-NC

This Photo by Unknown Author is licensed under CC BY-SA

This Photo by Unknown Author is licensed under CC BY-SA

Tic-Tac-Toe- Max's thoughts

This Photo by Unknown Author is licensed under CC BY-ND

Tic-Tac-Toe- Max's thoughts

This Photo by Unknown Author is licensed under CC BY-ND

Properties of environment

Properties	
Fully observable	Partially observable
Single-agent	Multiagent
Deterministic	Nondeterministic
Episodic	Sequential
Static	Dynamic
Discrete	Continues
Known	Unknown

The agents actively trying to defeat each other

Learning objective

- How to define a game
- Minimax value
- Minimax search algorithm
- Alpha-Beta pruning

Two-player zero-sum games

- The games most commonly studied within AI are what game theorists call deterministic, two-player, turn-taking, perfect information, zero-sum games.

- Zero-sum means that what is good for one player is just as bad for the other: there is no “win-win” outcome.

$$+1 + (-1) = 0$$

Problem formulation

- A game can be formally defined with the following elements:

$\boldsymbol{s_0}$: 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 states s.

Results(*s,a*): The transition model, which defines the state resulting from taking action a in state s.

IsTerminal(*s*): A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.

Utility(*s,p*): A utility function (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s.

Optimal Decisions in Games

- *MAX* wants to find a sequence of actions leading to a win, but *MIN* has something to say about it.
- This means that *MAX*'s strategy must be a conditional plan—a contingent strategy specifying a response to each of *MIN*'s possible moves.

Optimal Decisions in Games

Example- A two ply game tree

Example- A two ply game tree

Minimax value

- The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game. **What about Min?**
- The minimax value of a terminal state is just its utility.
- In a non-terminal state,
  - MAX prefers to move to a state of maximum value when it is MAX's turn to move.
  - MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN).

Minimax value

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

What if MIN does not play optimally?

What if MIN does not play optimally?

This Photo by Unknown Author is licensed under CC BY-NC

The minimax search algorithm

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

Max(v, value(successor))

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

Min(v, value(successor))

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

Minimax efficiency

- The minimax algorithm performs a complete depth-first exploration of the game tree.
- If the maximum depth of tree is *m* and there are *b* legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$ and the space complexity is $O(bm)$.

Chess:
b= 35 and the average game has depth of about 80 ply
It is not feasible to search $35^{80} \approx 10^{123}$ states

- MINIMAX does, however, serve as a basis for the mathematical analysis of games.
- By approximating the minimax analysis in various ways, we can derive more practical algorithms.

Alpha-Beta pruning

B is a min node which has a value of at most 3.

We have seen all B's successors states.

But we know B is worth ,
so *Max will never choose C*.

Alpha-Beta pruning

$$\text{MINIMAX}(root) = \max(\min(3,12,8), \min(2,x,y), \min(14,5,2))$$
$$= \max(3, \min(2,x,y),2)$$
$$= \max(3,z,2) \qquad \text{where } z = \min(2,x,y) \leq 2$$
$$= 3.$$

The value of the root and hence the minimax decision are independent of the values of the leaves and, and therefore they can be pruned.

Recap

- Problem formulation
- Minimax value (utility for Max)
- Minimax search algorithm (optimal decisions in games)
  - In two-player, discrete, deterministic, turn-taking zero-sum games with perfect information, the minimax algorithm can select optimal moves by a depth-first enumeration of the game tree.
- Alpha-Beta pruning (reducing computational complexity)
  - The alpha–beta search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.

Learning objective

- Heuristic Alpha-Beta search
- Evaluation function

Dealing with resource limit

- Even with alpha–beta pruning and clever move ordering, minimax won't work for games with too many states to be explored.

- **Type A strategy:** Considers all possible moves to a certain depth in the search tree, and then uses a heuristic evaluation function to estimate the utility of states at that depth. It explores a wide but shallow portion of the tree.

- **Type B strategy:** Ignores moves that look bad, and follows promising lines "as far as possible." It explores a deep but narrow portion of the tree.

Heuristic Alpha-Beta search

- To make use of our limited computation time, we <u>can cut off the search early</u> and apply a heuristic evaluation function to states, effectively <u>treating nonterminal nodes as if they were terminal</u>.

H- MINIMAX (s,d) =

$$\begin{cases} \text{Eval}(s, \text{MAX}) & \text{if IS-Cutoff}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

Heuristic Alpha-Beta search

- To make use of our limited computation time, we <u>can cut off the search early</u> and apply a heuristic evaluation function to states, effectively <u>treating nonterminal nodes as if they were terminal</u>.

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

Evaluation function

- A heuristic evaluation function *Eval(s,p)* returns an estimate of the expected utility of state *s* to player *p.*

- For terminal states, it must be *Eval(s,p)=Utility(s,p).*

- For nonterminal states, the evaluation must be somewhere between a loss and a win.

$$Utility(loss, p) \leq Eval(s, p) \leq Utility(win, p)$$

Evaluation function

- Beyond those requirements, what makes for a good evaluation function?

1. The computation must not take too long.
2. The evaluation function should be strongly correlated with the actual <u>chances of winning</u>.

If the search must be cut off at nonterminal states, then the algorithm will necessarily be uncertain about the final outcomes of those states.

Evaluation function

- For example, introductory chess books give an approximate material value for each piece:
  - Each pawn is worth 1.
  - A knight or bishop is worth 3.
  - A rook is worth 5.
  - The queen is worth 9.

- Other features such as “good pawn structure” and “king safety” might be worth half a pawn, say.
- These feature values are then simply added up to obtain the evaluation of the position.

Evaluation function

Weighted linear function

$$\text{Eval}(s) = \sum_{i=1}^{n} w_i f_i(s)$$

$$w_1 f_1$$

$$w_1 f_1 + w_2 f_2$$

$$w_1 f_1 + w_2 f_2 + w_3 f_3$$

$$\vdots$$

$$w_1 f_1 + w_2 f_2 + w_3 f_3 + \ldots + w_n f_n = \sum_{i=1}^{n} w_i f_i(s)$$

n =1

n =2

n =3

n =n

Evaluation function

$$w_1 f_1 + w_2 f_2 + w_3 f_3 + \ldots + w_n f_n = \sum_{i=1}^{n} w_i f_i(s)$$

A weight (saying how important that feature is).

Feature of the position (such as "number of white bishops")

The weights should be normalized so that the sum is always within the range of a loss (0) to a win (+1).

Adding up the values of features seems like a reasonable thing to do, but in fact it involves a strong assumption: that the contribution of each feature is independent of the values of the other features.

Evaluation function

- For this reason, current programs for chess and other games also use nonlinear combinations of features. For example, a pair of bishops might be worth more than twice the value of a single bishop, and a bishop is worth more in the endgame than earlier—when the *move number* feature is high or the *number of remaining pieces* feature is low.
- *Where do the features and weights come from?* They're not part of the rules of chess, but they are part of the culture of human chess-playing experience. In games where this kind of experience is not available, the weights of the evaluation function can be estimated by the machine learning techniques.

Recap

- Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we need to cut the search off at some point and apply a heuristic evaluation function that estimates the utility of a state.

- The evaluation function.

Learning objective

- Monte Carlo tree search algorithm
- Upper confidence bounds for selection policy
- Advantages and disadvantages of MCTS

3	7	8

3	7	8
		L

3	7	8
D		L

3	7	8
D		L
W		

3	7	8
D		L
W		
D		

3	7	8
D	W	L
W	W	L
D	W	D
D	W	D
W	W	D
W	W	L
D	W	D
		D

I Love
Me
W=0
V=1
W=1
V=1

I Love Me
W=1
V=1
W=0
V=1
W=1
V=1
W=0
V=1
W=1
V=1

I Love Me
W=1
V=2
W=0
V=1
W=0
V=1
W=0
V=1
W=1
V=1
W=1
V=1

I Love Me
W=1
V=3
W=0
V=1
W=0
V=2
W=0
V=1
W=1
V=1
W=0
V=2
W=1
V=1

I Love
Me

W=3
V=7
W=7
V=7
W=0
V=8
W=0
V=4
W=0
V=3
W=0
V=5
W=3
V=3
W=0
V=4
W=3
V=3
W=0
V=5

Monte Carlo Tree Search (MCTS)

- Monte Carlo tree search maintains a search tree and grows it on each iteration, based on the following four steps:

1. **Selection:** Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf.

What was the selection policy in our example?

Monte Carlo Tree Search (MCTS)

2. Expansion: We grow the search tree by generating a new child of the selected node;

3. Simulation: We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are not recorded in the search tree.

A simulation (also called a playout or rollout) chooses moves first for one player, than for the other, repeating until a terminal position is reached.

To get useful information from the playout we need a playout policy that biases the moves towards good ones. For Go and other games, playout policies have been successfully learned from self-play by using neural networks. Sometimes game-specific heuristics are used, such as “consider capture moves” in chess.

Monte Carlo Tree Search (MCTS)

4. Back-propagation: We use the result of the simulation to update all the search tree nodes going up to the root.

MCTS example

We repeat these four steps either for a set number of iterations, or until the allotted time has expired, and then return the move with the highest number of playouts.

Selection policy

- Upper confidence bounds applied to trees (UCT):

C is a constant that balances exploitation and exploration.

C = 1.4 C = 1.5

(a) Selection

$$UCB1(k,p) = E[win|k,p] + C\sqrt{\frac{2\ln(n_{parent(k)})}{n_k}}$$

$$\frac{w_{k,1}}{n_k}$$

Exploitation term: the average utility of k.

Exploration term: the average utility of k. This term will be high for nodes that have only been explored a few times. The exploration term goes to zero as the counts increase.

The Monte Carlo tree search 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)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

In any event, the UCB1 formula ensures that the node with the most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes up.

The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

Advantages

- Where the branching factor is very high (and thus alpha–beta can't search deep enough), or when it is difficult to define a good evaluation function.

- As long as we know the rules of the game, Monte Carlo search does not need any additional information.

- The selection and playout policies can make good use of hand-crafted expert knowledge when it is available, but good policies can be learned using neural networks trained by self-play alone.

Disadvantages:

- Monte Carlo search has a disadvantage when it is likely that a single move can change the course of the game, because the stochastic nature of Monte Carlo search means it might fail to consider that move. In other words, Type B pruning in Monte Carlo search means that a vital line of play might not be explored at all.

- Monte Carlo search also has a disadvantage when there are game states that are "obviously" a win for one side or the other (according to human knowledge and to an evaluation function), but where it will still take many moves in a playout to verify the winner.

Recap

- **Monte Carlo tree search (MCTS)** evaluates states not by applying a heuristic function, but by playing out the game all the way to the end and using the rules of the game to see who won. Since the moves chosen during the playout may not have been optimal moves, the process is repeated multiple times and the evaluation is an average of the results.

- **UCT** takes exploration and exploitation into account.

Optimal decisions in multiplayer games

Optimal decisions in multiplayer games

- In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.)

- In multiplayer games, we need to replace the single value for each node with a vector of values.

- For example, in a three-player game with players A, B , and C, a vector $<v_A, v_B, v_C>$ is associated with each node.

Optimal decisions in multiplayer games

Optimal decisions in multiplayer games

Optimal decisions in multiplayer games

The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Optimal decisions in multiplayer games

- Anyone who plays multiplayer games, becomes aware that much more is going on than in two-player games.
- Multiplayer games usually involve **alliances**, whether formal or informal, among the players.
- Alliances are made and broken as the game proceeds.