Adversarial Search

Rob Platt
Northeastern University

Some images and slides are used from:
AIMA
CS188 UC Berkeley
What is adversarial search?

Adversarial search: planning used to play a game such as chess or checkers – algorithms are similar to graph search except that we plan under the assumption that our opponent will maximize his own advantage...
Some types of games

- Chess: Solved/unsolved?
- Checkers: Solved/unsolved?
- Tic-tac-toe: Solved/unsolved?
- Go: Solved/unsolved?

Outcome of game can be predicted from any initial state assuming both players play perfectly
Examples of adversarial search

<table>
<thead>
<tr>
<th>Game</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>Unsolved</td>
</tr>
<tr>
<td>Checkers</td>
<td>Solved</td>
</tr>
<tr>
<td>Tic-tac-toe</td>
<td>Solved</td>
</tr>
<tr>
<td>Go</td>
<td>Unsolved</td>
</tr>
</tbody>
</table>

Outcome of game can be predicted from any initial state assuming both players play perfectly
### Examples of adversarial search

<table>
<thead>
<tr>
<th>Game</th>
<th>Status</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>Unsolved</td>
<td>~$10^{40}$</td>
</tr>
<tr>
<td>Checkers</td>
<td>Solved</td>
<td>~$10^{20}$</td>
</tr>
<tr>
<td>Tic-tac-toe</td>
<td>Solved</td>
<td>Less than $9! = 362k$</td>
</tr>
<tr>
<td>Go</td>
<td>Unsolved</td>
<td>?</td>
</tr>
</tbody>
</table>

Outcome of game can be predicted from any initial state assuming both players play perfectly.
Different types of games

- Deterministic / stochastic
- Two player / multi player?
- Zero-sum / non zero-sum
- Perfect information / imperfect information
Different types of games

Deterministic / stochastic

Two player / multi player?

Zero-sum / non zero-sum

Perfect information / imperfect information

**Zero Sum:**
- utilities of all players sum to zero
- pure competition

**Non-Zero Sum:**
- utility function of each play could be arbitrary
- optimal strategies could involve cooperation
Formalizing a Game

Given:

- $S_0$: The **initial state**, which specifies how the game is set up at the start.
- $\text{PLAYER}(s)$: Defines which player has the move in a state.
- $\text{ACTIONS}(s)$: Returns the set of legal moves in a state.
- $\text{RESULT}(s, a)$: The **transition model**, which defines the result of a move.
- $\text{TERMINAL-TEST}(s)$: A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- $\text{UTILITY}(s, p)$: A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state $s$ for a player $p$. In chess, the outcome is a win, loss, or draw, with values $+1$, $0$, or $\frac{1}{2}$. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from $0$ to $+192$. A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either $0 + 1$, $1 + 0$ or $\frac{1}{2} + \frac{1}{2}$. “Constant-sum” would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of $\frac{1}{2}$.

**Calculate a policy**: $\pi(s, p)$ **Action that player p should take from state s**
Formalizing a Game

Given:

- \( S_0 \): The initial state, which specifies how the game is set up at the start.
- \( \text{PLAYER}(s) \): Defines which player has the move in a state.
- \( \text{ACTIONS}(s) \): Returns the set of legal moves in a state.
- \( \text{RESULT}(s, a) \): The transition model, which defines the result of a move.
- \( \text{TERMINAL-TEST}(s) \): A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
- \( \text{UTILITY}(s, p) \): A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state \( s \) for a player \( p \). In chess, the outcome is a win, loss, or draw, with values +1, 0, or \( \frac{1}{2} \). Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192. A zero-sum game is (confusingly) defined as one where the total payoff to all players is the same for every game; this term would have been a better term, but zero-sum is traditional; imagine each player is charged an entry fee of \( \frac{1}{2} \).

How?

Calculate a policy: \( \pi(s, p) \)  Action that player \( p \) should take from state \( s \)
How solve for a policy?

Use adversarial search!
– build a game tree
This is a game tree for tic-tac-toe
This is a game tree for tic-tac-toe
This is a game tree for tic-tac-toe

MAX (x)

MIN (o)

MAX (x)

MIN (o)

TERMINAL

Utility

-1 0 +1

You

Them
This is a game tree for tic-tac-toe

MAX (x)

MIN (o)

MAX (x)

MIN (o)

TERMINAL

Utility

-1 0 +1

You

Them

You
This is a game tree for tic-tac-toe
This is a game tree for tic-tac-toe

MAX (x)

MIN (o)

MAX (x)

MIN (o)

TERMINAL

Utility

You

Them

Utility
What is Minimax?

Consider a simple game:
  1. you make a move
  2. your opponent makes a move
  3. game ends
What is Minimax?

Consider a simple game:
1. you make a move
2. your opponent makes a move
3. game ends

What does the minimax tree look like in this case?
What is Minimax?

Consider a simple game:
1. you make a move
2. your opponent makes a move
3. game ends

What does the minimax tree look like in this case?
What is Minimax?

Max (you)

Min (them)

Max (you)

These are terminal utilities – assume we know what these values are
What is Minimax?

Max (you)

Min (them)

Max (you)

\[ V(s) = \min_{s' \in \text{successors}(s)} V(s') \]
What is Minimax?

\[ V(s) = \max_{s' \in \text{successors}(s)} V(s') \]
What is Minimax?

Max (you)

Min (them)

Max (you)

This is called “backing up” the values

\[ V(s) = \max_{s' \in \text{successors}(s)} V(s') \]
Okay – so we know how to back up values ...

... but, how do we construct the tree?

This tree is already built...
Minimax

Notice that we only get utilities at the *bottom* of the tree … – therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the bottom of the tree ... – therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the *bottom* of the tree ... – therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the *bottom* of the tree ... – therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the \textit{bottom} of the tree … – therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the bottom of the tree ... therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the *bottom* of the tree ... – therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the bottom of the tree ... – therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the bottom of the tree ... – therefore, DFS makes sense.
Minimax

Notice that we only get utilities at the *bottom* of the tree …
– therefore, DFS makes sense.
– since most games have forward progress, the distinction between tree search and graph search is less important.
function MINIMAX-DECISION(state) returns an action
    return arg max, a ∈ ACTIONS(s) MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, a)))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, a)))
    return v

Figure 5.3  An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation arg max, a ∈ S f(a) computes the element a of set S that has the maximum value of f(a).
Is it always correct to assume your opponent plays optimally?

Minimax properties

Max (you)

Min (them)

Max (you)

10

10

9

100
Minimax properties

Is minimax optimal? Is it complete?
Minimax properties

Is minimax optimal? Is it complete?

Time complexity = ?

Space complexity = ?
Minimax properties

Is minimax optimal? Is it complete?

Time complexity = $O(b^d)$

Space complexity = $O(bd)$
Is minimax optimal? Is it complete?

Time complexity = $O(b^d)$

Space complexity = $O(bd)$

Is it practical? In chess, $b=35$, $d=100$
Minimax properties

Is minimax optimal? Is it complete?

Time complexity = $O(b^d)$

Space complexity = $O(bd)$

Is it practical? In chess, $b=35, d=100$

$O(35^{100})$ is a big number...
Minimax properties

Is minimax optimal? Is it complete?

Time complexity = $O(b^d)$

Space complexity = $O(bd)$

Is it practical? In chess, $b=35, d=100$

$O(35^{100})$ is a big number...

So what can we do?
Key idea: cut off search at a certain depth and give the corresponding nodes an estimated value.

Evaluation functions

Cut off recursion here
Key idea: cut off search at a certain depth and give the corresponding nodes an estimated value.

Evaluation functions

the evaluation function makes this estimate.

Cut off recursion here
Evaluation functions

How does the evaluation function make the estimate?
– depends upon domain

For example, in chess, the value of a state might equal the sum of piece values.
– a pawn counts for 1
– a rook counts for 5
– a knight counts for 3
...

![Chess board diagram](image1)
Black to move
White slightly better

![Chess board diagram](image2)
White to move
Black winning
A weighted linear evaluation function

$$eval(s) = w_1 f_1(s) + \cdots + w_n f_n(s)$$

$$f_1(s) \equiv \text{number of pawns on the board}$$

$$f_2(s) \equiv \text{number of knights on the board}$$

$$w_1 = 1 \quad \text{A pawn counts for 1}$$

$$w_2 = 3 \quad \text{A knight counts for 3}$$

Eval = 3 - 2.5 = 0.5

Eval = 3 + 2.5 + 1 + 1 - 2.5 = 5
A weighted linear evaluation function

\[ eval(s) = w_1 f_1(s) + \cdots + w_n f_n(s) \]

\[ f_1(s) \equiv \text{number of pawns on the board} \]
\[ f_2(s) \equiv \text{number of knights on the board} \]

\[ w_1 = 1 \quad \text{A pawn counts for 1} \]
\[ w_2 = 3 \quad \text{A knight counts for 3} \]

Maybe consider other factors as well?

Eval = 3 - 2.5 = 0.5  Eval = 3 + 2.5 + 1 + 1 - 2.5 = 5
Evaluation functions

Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

Instead, search only to a limited depth in the tree
Replace terminal utilities with an evaluation function for non-terminal positions

Example:

Suppose we have 100 seconds
Can explore 10K nodes / sec
So can check 1M nodes per move

Guarantee of optimal play is gone

More plies makes a BIG difference

Use iterative deepening for an anytime algorithm
At what depth do you run the evaluation function?

Option 1: cut off search at a fixed depth

Option 2: cut off search at particular states deeper than a certain threshold

The deeper your threshold, the less the quality of the evaluation function matters...
Alpha/Beta pruning
Alpha/Beta pruning

3

12

8
Alpha/Beta pruning
Alpha/Beta pruning
Alpha/Beta pruning
Alpha/Beta pruning

We don't need to expand this node!
We don't need to expand this node! Why?
We don't need to expand this node!

Why?

Max

Min

We don't need to expand this node!

Why?
Alpha/Beta pruning

Max

Min

3

12

8

2

14

5

2
So, we don't need to expand these nodes in order to back up correct values!
Alpha/Beta pruning

So, we don't need to expand these nodes in order to back up correct values! That's alpha-beta pruning.
Alpha/Beta pruning: algorithm

α: MAX’s best option on path to root
β: MIN’s best option on path to root

**def max-value(state, α, β):**
initialize v = -∞
for each successor of state:
  v = max(v, value(successor, α, β))
  if v ≥ β return v
  α = max(α, v)
return v

**def min-value(state, α, β):**
initialize v = +∞
for each successor of state:
  v = min(v, value(successor, α, β))
  if v ≤ α return v
  β = min(β, v)
return v
Alpha/Beta pruning

(-inf, +inf)
Alpha/Beta pruning

(-inf, +inf)

(-inf, +inf)
Best value for far for MIN along path to root

(-inf,3) --> 3

(-inf, +inf)
Best value for far for MIN along path to root

(-inf, 3) → 3 → 3, 12 → (-inf, +inf)
Alpha/Beta pruning

Best value for far for MIN along path to root

(-inf,3) → 3 → 3, 12, 8 → (-inf, +inf)
Alpha/Beta pruning

Best value for far for MAX along path to root

(3, +\infty)

(-\infty, 3)

3

3, 12, 8
Alpha/Beta pruning

\[(3, +\infty)\]

\[(-\infty, 3)\]

\[
\begin{array}{c}
3 \\
3 \quad 12 \quad 8 \\
\end{array}
\]

\[
(3, +\infty)
\]
Alpha/Beta pruning

(-\infty, 3) → 3 → 3, 12, 8 → (3, +\infty)

(3, +\infty) → (3, +\infty) → 2 → 2
Alpha/Beta pruning

Prune because value (2) is out of alpha-beta range
Alpha/Beta pruning

(-\infty, 3)

3

3

12

8

(3, +\infty)

(3, +\infty)

(3, +\infty)
Alpha/Beta pruning

(-\infty, 3) → (3, +\infty) → (3, +\infty) → (3, 14) → 14
Alpha/Beta pruning

(-∞, 3)

(3, +∞)

(-∞, 3)

(3, +∞)

(3, 5)

(3, +∞)
Alpha/Beta pruning

```
(3, +inf)
```

```
(3, -inf)
```

```
(3, +inf)
```

```
(3, 5)
```

```
3 - inf, 3
```

```
3 12 8
```

```
2
```

```
14 5 2
```

```
2
```

```
3 12 8
```

```
2
```

```
(3, 5)
```

Alpha/Beta algorithm

function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, −∞, +∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(Result(s, a), α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v

function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(Result(s, a), α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v
Alpha/Beta properties

Is it complete?
Alpha/Beta properties

Is it complete?

How much does alpha/beta help relative to minimax?

Minimax time complexity = $O(b^m)$

Alpha/beta time complexity $\geq O(b^{\frac{m}{2}})$

– the improvement w/ alpha/beta depends upon move ordering...
Alpha/Beta properties

Is it complete?

How much does alpha/beta help relative to minimax?

Minimax time complexity = $O(b^m)$

Alpha/beta time complexity $\geq O(b^{\frac{m}{2}})$

– the improvement w/ alpha/beta depends upon move ordering...

The order in which we expand a node.
Alpha/Beta properties

Is it complete?

How much does alpha/beta help relative to minimax?

Minimax time complexity = $O(b^m)$

Alpha/beta time complexity $\geq O(b^{\frac{m}{2}})$

– the improvement with alpha/beta depends upon move ordering...

The order in which we expand a node.

How to choose move ordering? Use IDS.
– on each iteration of IDS, use prior run to inform ordering of next node expansions.
What if your opponent does not maximize his/her utility? – e.g. suppose he/she picks moves uniformly at random?
Expectimax

Minimax backup for a rational agent:

Max (you) -> Min (them) -> Max (you)
Expectimax

Minimax backup for agent who selects actions uniformly at random:

Max (you)

Min (them)

Max (you)
Expectimax

Minimax backup for agent who selects actions uniformly at random:

Max (you)

Min (them)

Max (you)

Instead of backing up min values for min-plys, back up the *average* – could also account for agents who are somewhere in between rational and uniformly random. How? – later, this idea will be generalized using Markov Decision Processes
Mixing these ideas: Nondeterministic games

Backgammon
Nondeterministic games in general

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

Simplified example with coin-flipping: