Introduction to Game Theory

4. Game Tree Search

Dana Nau
University of Maryland
Finite perfect-information zero-sum games

*Finite*: finitely many agents, actions, states

*Perfect information*: every agent knows the current state, all of the actions, and what they do
No simultaneous actions – agents move one-at-a-time

*Constant-sum*: regardless of how the game ends, \( \sum \{ \text{agents' utilities} \} = k \).
For every such game, there’s an equivalent game in which \( k = 0 \).
Thus constant-sum games usually are called *zero-sum* games

Examples:

- **Deterministic**: chess, checkers, go, othello (reversi), connect-four, qubic, mancala (awari, kalah), 9 men’s morris (merelles, morels, mill)
- **Stochastic**: backgammon, monopoly, yahtzee, parcheesi, roulette, craps

For now, we’ll consider just the deterministic games
Outline

♦ A brief history of work on this topic
♦ The minimax theorem
♦ Game trees
♦ The minimax algorithm
♦ α-β pruning
♦ Resource limits and approximate evaluation
♦ Games of chance (briefly)
A brief history

1846 (Babbage): machine to play tic-tac-toe

1928 (von Neumann): minimax theorem

1944 (von Neumann & Morgenstern): backward-induction algorithm (produces perfect play)

1950 (Shannon): minimax algorithm (finite horizon, approximate evaluation)

1951 (Turing): program (on paper) for playing chess

1952–7 (Samuel): checkers program, capable of beating its creator

1956 (McCarthy): pruning to allow deeper search

1957 (Bernstein): first complete chess program, on an IBM 704 vacuum-tube computer, could examine about 350 positions/minute
A brief history, continued

1967 (Greenblatt): first program to compete in human chess tournaments: 3 wins, 3 draws, 12 losses

1992 (Schaeffer): Chinook won the 1992 US Open checkers tournament

1994 (Schaeffer): Chinook became world checkers champion; Tinsley (human champion) withdrew for health reasons

1997 (Hsu et al): Deep Blue won 6-game chess match against world chess champion Gary Kasparov

2007 (Schaeffer et al, 2007): Checkers solved: with perfect play, it’s a draw. This took $10^{14}$ calculations over 18 years
Quick review

Recall that
◇ A strategy tells what an agent will do in every possible situation
◇ Strategies may be pure (deterministic) or mixed (probabilistic)

Suppose agents 1 and 2 use strategies $s$ and $t$ to play a two-person zero-sum game $G$. Then

- Agent 1’s expected utility is $u_1(s, t)$
  From now on, we’ll just call this $u(s, t)$

- Since $G$ is zero-sum, $u_2(s, t) = -u(s, t)$

We’ll call agent 1 Max, and agent 2 Min

Max wants to maximize $u$ and Min wants to minimize it
The Minimax Theorem (von Neumann, 1928)

◊ A restatement of the Minimax Theorem that refers directly to the agents’ minimax strategies:

**Theorem.** Let $G$ be a two-person finite zero-sum game. Then there are strategies $s^*$ and $t^*$, and a number $u^*$, called $G$’s minimax value, such that

- If Min uses $t^*$, Max’s expected utility is $\leq u^*$, i.e., $\max_s u(s, t^*) = u^*$
- If Max uses $s^*$, Max’s expected utility is $\geq u^*$, i.e., $\min_t u(s^*, t) = u^*$

**Corollary 1:** $u(s^*, t^*) = u^*$.

**Corollary 2:** If $G$ is a perfect-information game, then there are pure strategies $s^*$ and $t^*$ that satisfy the theorem.
Game trees

**Root node** = the initial state

**Children of a node** = the states an agent can move to

**Terminal node** = a state where the game ends

### Graphical Representation

- **MAX (X)**: Node with moves for the maximizing player.
- **MIN (O)**: Node with moves for the minimizing player.
- **Terminals** represent the end state of the game.
- **Utility** values indicate the outcome of the game.

-1, 0, +1 represent the utility values.
Strategies on game trees

To construct a pure strategy for Max:

- At each node where it’s Max’s move, choose one branch
- At each node where it’s Min’s move, include all branches

Let $b =$ the branching factor (max. number of children of any node)
$h =$ the tree’s height (max. depth of any node)

The number of pure strategies for Max $\leq b^{\lceil h/2 \rceil}$,
with equality if every node of height $< h$ node has $b$ children
Strategies on game trees

To construct a pure strategy for Min:
- At each node where it’s Min’s move, choose one branch
- At each node where it’s Max’s move, include all branches

The number of pure strategies for Min $\leq b^{\lceil h/2 \rceil}$
with equality if every node of height $< h$ node has $b$ children
Finding the best strategy

Brute-force way to find Max’s and Min’s best strategies:

Construct the sets $S$ and $T$ of all of Max’s and Min’s pure strategies, then choose

$$s^* = \arg \max_{s \in S} \min_{t \in T} u(s, t)$$

$$t^* = \arg \min_{t \in T} \max_{s \in S} u(s, t)$$

Complexity analysis:

- Need to construct and store $O(b^{h/2} + b^{h/2}) = O(b^{h/2})$ strategies
- Each strategy is a tree that has $O(b^{h/2})$ nodes
- Thus space complexity is $O(b^{h/2}b^{h/2}) = O(b^h)$
- Time complexity is slightly worse

But there’s an easier way to find the strategies
Backward induction

Depth-first implementation of the backward induction (from Session 3):

```
function BACKWARD-INDUCTION(x) returns a utility value
    if x is a terminal state then return Max’s payoff at x
    else if it is Max’s move at x then
        return \max\{BACKWARD-INDUCTION(y) : y is a child of x\}
    else return \min\{BACKWARD-INDUCTION(y) : y is a child of x\}
```

Returns $x$’s minimax utility $u^*(x)$
To get the action to perform at $x$, return \arg\max or \arg\min
Properties

Space complexity: $O(bh)$, where $b$ and $h$ are as defined earlier

Time complexity: $O(b^h)$

For chess:

$b \approx 35, h \approx 100$ for "reasonable" games

$35^{100} \approx 10^{135}$ nodes

Number of particles in the universe $\approx 10^{87}$

Number of nodes is $\approx 10^{55}$ times the number of particles in the universe

$\Rightarrow$ no way to examine every node!
Minimax algorithm (Shannon, 1950)

Modified version of the backward-induction algorithm:

- \( d \) (an integer) is an upper bound on the search depth
- \( e(x) \), the *static evaluation function*, returns an estimate of \( u^*(x) \)
- Whenever we reach a nonterminal node of depth \( d \), return \( e(x) \)

If \( d = \infty \), then \( e \) will never be called, and \( \text{MINIMAX} \) will return \( u^*(x) \)

```plaintext
function \text{MINIMAX}(x, d) returns an estimate of \( x \)'s utility value
    inputs: \( x \), current state in game
             \( d \), an upper bound on the search depth
    if \( x \) is a terminal state then return Max’s payoff at \( x \)
    else if \( d = 0 \) then return \( e(s) \)
    else if it is Max’s move at \( x \) then
        return \( \max \{ \text{MINIMAX}(y, d-1) : y \text{ is a child of } x \} \)
    else return \( \min \{ \text{MINIMAX}(y, d-1) : y \text{ is a child of } x \} \)
```

Nau: Game Theory 14
Evaluation functions

\( e(x) \) is often a weighted sum of features

\[ e(x) = w_1 f_1(x) + w_2 f_2(x) + \ldots + w_n f_n(x) \]

E.g., in chess,

1 (white pawns – black pawns) + 3 (white knights – black knights) + \ldots
Exact values for $E_{VAL}$ don’t matter

Behavior is preserved under any monotonic transformation of $E_{VAL}$

Only the order matters:

Payoff acts as an ordinal utility function
Pruning example 1

**Backward-Induction** and **Minimax** both look at nodes that don’t need to be examined.
**Pruning example 1**

**Backward-Induction** and **Minimax** both look at nodes that don’t need to be examined.

Max will never go to $f$, because Max gets a higher utility by going to $b$.

Since $f$ is worse than $b$, it can’t affect $a$’s minimax value.
Pruning example 1

**BACKWARD-INDUCTION** and **MINIMAX** both look at nodes that don’t need to be examined

Don’t know whether $h$ is better or worse than $a$
Pruning example 1

**BACKWARD-INDUCTION** and **MINIMAX** both look at nodes that don’t need to be examined

![Game tree diagram]

Still don’t know whether \( h \) is better or worse than \( a \)
Pruning example 1

BACKWARD-INDUCTION and MINIMAX both look at nodes that don’t need to be examined

$h$ is worse than $a$
Alpha-beta pruning

Start a minimax search at node $c$

Let $\alpha = \text{biggest lower bound on any ancestor of } f$

\[ \alpha = \max(-2, 4, 0) = 4 \text{ in the example} \]

If the game reaches $f$, Max will get utility $\leq 3$

To reach $f$, the game must go through $d$

But if the game reaches $d$, Max can get utility $\geq 4$ by moving off of the path to $f$

So the game will never reach $f$

We can stop trying to compute $u^*(f)$, because it can’t affect $u^*(c)$

This is called an *alpha cutoff*
Alpha-beta pruning

Start a minimax search at node $a$

Let $\beta =$ smallest upper bound on any ancestor of $d$

$\beta = \min(5, -2, 3) = -2$ in the example

If the game reaches $d$, Max will get utility $\geq 0$

To reach $d$, the game must go through $b$

But if the game reaches $b$, Min can make Max’s utility $\leq -2$ by moving off of the path to $d$

So the game will never reach $d$

We can stop trying to compute $u^*(d)$, because it can’t affect $u^*(a)$

This is called a beta cutoff
The alpha-beta algorithm

function $\text{ALPHA-BETA}(x, d, \alpha, \beta)$ returns an estimate of $x$’s utility value

inputs: $x$, current state

$d$, maximum search depth

$\alpha$ and $\beta$, lower and upper bounds on ancestors’ values

if $x$ is a terminal state then return $\text{Max}$’s payoff at $x$

else if $d = 0$ then return $e(x)$

else if it is $\text{Max}$’s move at $x$ then

$v \leftarrow -\infty$

for every child $y$ of $x$ do

$v \leftarrow \max(v, \text{ALPHA-BETA}(y, d-1, \alpha, \beta))$

if $v \geq \beta$ then return $v$

$\alpha \leftarrow \max(\alpha, v)$

else

$v \leftarrow \infty$

for every child $y$ of $x$ do

$v \leftarrow \min(v, \text{ALPHA-BETA}(y, d-1, \alpha, \beta))$

if $v \leq \alpha$ then return $v$

$\beta \leftarrow \min(\alpha, v)$

return $v$
α-β pruning example

\[ a = -\infty \]
\[ \beta = \infty \]

\[ a = -\infty \]
\[ \beta = \infty \]
**α-β pruning example**

\[ \alpha = -\infty \quad \beta = \infty \]

\[ \alpha = -\infty \quad \beta = \infty \]

\[ 7 \]

\[ c \]

\[ d \]

\[ e \]

\[ f \]

\[ g \]

\[ h \]

\[ i \]

\[ k \]

\[ l \]

\[ m \]

\[ \ldots \]
α-β pruning example
\( \alpha - \beta \) pruning example

\[
\begin{align*}
\alpha &= -\infty \\
\beta &= \infty \\
\end{align*}
\]
\( \alpha - \beta \) pruning example

\[
\begin{align*}
\alpha &= -\infty \quad \beta = \infty \\
a &= 7 \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*}
\]

\[
\begin{align*}
\alpha &= 7 \\
\beta &= \infty
\end{align*}
\]
$\alpha$-$\beta$ pruning example

\begin{itemize}
  \item $a = -\infty$
  \item $b = \infty$
  \item $c = \infty$
  \item $d = \infty$
  \item $e = \infty$
  \item $f = \infty$
  \item $g = \infty$
  \item $h = \infty$
  \item $i = \infty$
  \item $j = \infty$
  \item $k = \infty$
  \item $l = \infty$
\end{itemize}

\begin{itemize}
  \item $m = \infty$
\end{itemize}

Nau: Game Theory 30
\(\alpha - \beta\) pruning example
\( \alpha - \beta \) pruning example

\[
\begin{align*}
\alpha &= -\infty \\
\beta &= \infty \\
\alpha &= 7 \\
\beta &= \infty \\
\alpha &= 7 \\
\beta &= 8 \\
\alpha &= -\infty \\
\beta &= \infty
\end{align*}
\]
\(\alpha - \beta\) pruning example
\( \alpha - \beta \) pruning example

\[
\begin{array}{c}
\alpha = -\infty \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]

\[
\begin{array}{c}
\alpha = 7 \\
\beta = \infty
\end{array}
\]
Properties of $\text{ALPHA-BETA}$

$\text{ALPHA-BETA}$ is a simple example of reasoning about which computations are relevant (a form of $\text{metareasoning}$)

$\diamondsuit$ if $\alpha \leq \text{MINIMAX}(x, d) \leq \beta$,  
then $\text{ALPHA-BETA}(x, d, \alpha, \beta)$ returns $\text{MINIMAX}(x, d)$

$\diamondsuit$ if $\text{MINIMAX}(x, d) \leq \alpha$,  
then $\text{ALPHA-BETA}(x, d, \alpha, \beta)$ returns a value $\leq \alpha$

$\diamondsuit$ if $\text{MINIMAX}(x, d) \geq \beta$,  
then $\text{ALPHA-BETA}(x, d, \alpha, \beta)$ returns a value $\geq \beta$

Consequently,

$\diamondsuit$ If $\alpha = -\infty$ and $\beta = \infty$,  
then $\text{ALPHA-BETA}(x, d, \alpha, \beta)$ returns $\text{MINIMAX}(x, d)$

$\diamondsuit$ If $\alpha = -\infty$, $\beta = \infty$, and $d = \infty$,  
then $\text{ALPHA-BETA}(x, d, \alpha, \beta)$ returns $u^*(x)$
Properties of \textbf{ALPHA-BETA}

Good move ordering can enable us to prune more nodes

Best case is if
- ♦ at nodes where it’s Max’s move, children are largest-value first
- ♦ at nodes where it’s Min’s move, children are smallest-value first

In this case, ALPHA-BETA’s time complexity is \(O(b^{h/2})\)
  \(\Rightarrow\) doubles the solvable depth

Worst case is the reverse
- ♦ at nodes where it’s Max’s move, children are smallest-value first
- ♦ at nodes where it’s Min’s move, children are largest-value first

In this case, ALPHA-BETA will visit every node of depth \(\leq d\)
Hence time complexity is the same as MINIMAX: \(O(b^h)\)
Deeper lookahead (i.e., larger depth bound $d$) usually gives better decisions.

Exceptions do exist, and we’ll discuss them in the next session:

“Pathological” games in which deeper lookahead gives worse decisions
But such games are rare

Suppose we have 100 seconds, explore $10^4$ nodes/second
$\Rightarrow 10^6 \approx 35^{8/2}$ nodes per move
$\Rightarrow$ ALPHA-BETA reaches depth 8 $\Rightarrow$ pretty good chess program

Some modifications that can improve the accuracy or computation time:

node ordering (see next slide)
quiescence search
biasing
transposition tables
thinking on the opponent’s time

...
Node ordering

Recall that I said:

Best case is if

◊ at nodes where it’s Max’s move, children are largest first
◊ at nodes where it’s Min’s move, children are smallest first

In this case time complexity = \( O(b^{h/2}) \) ⇒ doubles the solvable depth

Worst case is the reverse

How to get closer to the best case:

◊ Every time you expand a state \( s \), apply \texttt{Eval} to its children
◊ When it’s Max’s move, sort the children in order of largest \texttt{Eval} first
◊ When it’s Min’s move, sort the children in order of smallest \texttt{Eval} first
Quiescence search and biasing

♦ In a game like checkers or chess, where the evaluation is based greatly on material pieces,
  The evaluation is likely to be inaccurate if there are pending captures

♦ Search deeper to reach a position where there aren’t pending captures
  Evaluations will be more accurate here

♦ But that creates another problem
  You’re searching some paths to an even depth, others to an odd depth
  Paths that end just after your opponent’s move will look worse than paths that end just after your move

♦ To compensate, add or subtract a number called the “biasing factor”
Transposition tables

Often there are multiple paths to the same state (i.e., the state space is a really graph rather than a tree)

Idea:
◊ when you compute $s$’s minimax value, store it in a hash table
◊ visit $s$ again $\Rightarrow$ retrieve its value rather than computing it again

The hash table is called a **transposition table**

Problem: far too many states to store all of them

Store some of the states, rather than all of them

Try to store the ones that you’re most likely to need
Thinking on the opponent’s time

Current state $a$

Children $b, c$

Use alpha-beta to estimate their minimax values

Move to the largest, $c$

Consider your estimates of $f$ and $g$’s minimax values

Your opponent is likely to move to $f$ since its value is smaller

Do a minimax search below $f$ while waiting for the opponent to move

If he/she moves to $f$ then you’ve already done a lot of the work of figuring out your next move
Game-tree search in practice

**Checkers**: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994.

Checkers was *solved* in April 2007: from the standard starting position, both players can guarantee a draw with perfect play. This took $10^{14}$ calculations over 18 years. Checkers has a search space of size $5 \times 10^{20}$.

**Chess**: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

**Othello**: human champions refuse to compete against computers, who are too good.

**Go**: until recently, human champions didn’t compete against computers because the computers were too bad. But that has changed . . .
Game-tree search in the game of go

A game tree’s size grows exponentially with both its depth and its branching factor.

Go is huge:
- branching factor $\approx 200$
- game length $\approx 250$ to $300$ moves
- number of paths in the game tree $\approx 10^{525}$ to $10^{620}$

Much too big for a normal game tree search.

Comparison:
- Number of atoms in universe: about $10^{80}$
- Number of particles in universe: about $10^{87}$
Game-tree search in the game of go

During the past couple years, go programs have gotten much better

Main reason: **Monte Carlo roll-outs**

Basic idea: do a minimax search of a randomly selected subtree

At each node that the algorithm visits,

◊ It randomly selects some of the children
  There are heuristics for deciding how many

◊ Calls itself recursively on these, ignores the others
Forward pruning in chess

Back in the 1970s, some similar ideas were tried in chess.

The approach was called forward pruning.
Main difference: select the children heuristically rather than randomly.
It didn’t work as well as brute-force alpha-beta, so people abandoned it.

Why does a similar idea work so much better in go?
Summary

If a game is two-player zero-sum,
then maximin and minimax are the same

If the game also is perfect-information,
only need to look at pure strategies

If the game also is sequential, deterministic, and finite,
then can do a game-tree search
   minimax values, alpha-beta pruning

In sufficiently complicated games, perfection is unattainable
⇒ must approximate: limited search depth, static evaluation function

In games that are even more complicated, further approximation is needed
⇒ Monte Carlo roll-outs