function ALPHA-BETA-SEARCH(state) returns an action
inputs: state, current state in game
\[ v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \]
return an action in ACTIONS(state) with value \( v \)

function MAX-VALUE(state, \( \alpha \), \( \beta \)) returns a utility value
if TERMINAL-TEST(state) then return \( \text{UTILITY}(state) \)
\[ v \leftarrow -\infty \]
for \( a \) in ACTIONS(state) do
\[ v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{Result}(s,a), \alpha, \beta)) \]
if \( v \geq \beta \) then return \( v \)
\[ \alpha \leftarrow \text{MAX}(\alpha, v) \]
return \( v \)

function MIN-VALUE(state, \( \alpha \), \( \beta \)) returns a utility value
if TERMINAL-TEST(state) then return \( \text{UTILITY}(state) \)
\[ v \leftarrow +\infty \]
for \( a,s \) in SUCCESSORS(state) do
\[ v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta)) \]
if \( v \leq \alpha \) then return \( v \)
\[ \beta \leftarrow \text{MIN}(\beta, v) \]
return \( v \)
Game-playing AIs: Games and Adversarial Search

AIMA 5.1-5.5,
AIMA 16.1-16.3
Games: Outline of Unit

Part I: Games as Search
- Motivation
- Game-playing AI successes
- Game Trees
- Evaluation Functions

Part II: Adversarial Search
- The Minimax Rule
- Alpha-Beta Pruning
Ratings of human & computer chess champions

AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol

DeepMind's artificial intelligence astonishes fans to defeat human opponent and offers evidence computer software has mastered a major challenge.

Google DeepMind’s AlphaGo program triumphed in its final game against South Korean Go grandmaster Lee Sedol to win the series 4-1, providing further evidence of the landmark achievement for an artificial intelligence program.

Lee started Tuesday’s game strongly, taking advantage of an early mistake by AlphaGo. But in the end, Lee was unable to hold off a comeback by his opponent, which won a narrow victory.
The Simplest Game Environment

- **Multiagent**
- **Static:** No change while an agent is deliberating
- **Discrete:** A finite set of percepts and actions
- **Fully observable:** An agent's sensors give it the complete state of the environment.
- **Strategic:** The next state is determined by the current state and the action executed by the agent and the actions of one other agent.
Key properties of our sample games

1. Two players alternate moves
2. Zero-sum: one player’s loss is another’s gain
3. Clear set of legal moves
4. Well-defined outcomes (e.g. win, lose, draw)

- Examples:
  - Chess, Checkers, Go,
  - Mancala, Tic-Tac-Toe, Othello …
More complicated games

- Most card games (e.g. Hearts, Bridge, etc.) and Scrabble
  - Stochastic, not deterministic
  - Not fully observable: lacking in perfect information
- Real-time strategy games, e.g. Warcraft
  - Continuous rather than discrete
  - No pause between actions, don’t take turns
- Cooperative games
Pac-Man

https://youtu.be/-CbyAk3Sn9I
Formalizing the Game setup

1. Two players: MAX and MIN; MAX moves first.
2. MAX and MIN take turns until the game is over.
3. Winner gets award, loser gets penalty.

• Games as search:
  • Initial state: e.g. board configuration of chess
  • Successor function: list of (move,state) pairs specifying legal moves.
  • Terminal test: Is the game finished?
  • Utility function: Gives numerical value of terminal states. e.g. win (+∞), lose (-∞) and draw (0)
  • MAX uses search tree to determine next move.
How to Play a Game by Searching

• General Scheme
  1. Consider all legal successors to the current state (‘board position’)
  2. Evaluate each successor board position
  3. Pick the move which leads to the best board position.
  4. After your opponent moves, repeat.

• Design issues
  1. Representing the ‘board’
  2. Representing legal next boards
  3. Evaluating positions
  4. Looking ahead
Hexapawn: A very simple Game

- Hexapawn is played on a 3x3 chessboard

- Only standard pawn moves:
  1. A pawn moves forward one square onto an empty square
  2. A pawn “captures” an opponent pawn by moving diagonally forward one square, if that square contains an opposing pawn. The opposing pawn is removed from the board.
Hexapawn: A very simple Game

- Hexapawn is played on a 3x3 chessboard

- Player $P_1$ wins the game against $P_2$ when:
  - One of $P_1$’s pawns reaches the far side of the board, or
  - $P_2$ cannot move because no legal move is possible.
  - $P_2$ has no pawns left.

(Invented by Martin Gardner in 1962, with learning “program” using match boxes. Reprinted in “The Unexpected Hanging..)
Hexapawn: Three Possible First Moves

White moves
Game Trees

- Represent the game problem space by a tree:
  - Nodes represent ‘board positions’; edges represent legal moves.
  - Root node is the first position in which a decision must be made.
Hexapawn: Simplified Game Tree for 2 Moves

White to move

Black to move

White to move
Adversarial Search
Battle of Wits

https://www.youtube.com/watch?v=rMz7JBRbmNo
MAX & MIN Nodes: An egocentric view

- Two players: MAX, MAX’s opponent MIN
- *All play is computed from MAX’s vantage point.*
- When MAX moves, MAX attempts to MAXimize MAX’s outcome.
- When MAX’s opponent moves, they attempt to MINimize MAX’s outcome.

WE TYPICALLY ASSUME MAX MOVES FIRST:

- Label the root (level 0) MAX
- Alternate MAX/MIN labels at each successive tree level *(ply).*
- *Even levels* represent turns for MAX
- *Odd levels* represent turns for MIN
Game Trees

- Represent the game problem space by a tree:
  - Nodes represent ‘board positions’; edges represent legal moves.
  - Root node is the first position in which a decision must be made.

- Evaluation function $f$ assigns real-number scores to `board positions’ without reference to path

- Terminal nodes represent ways the game could end, labeled with the desirability of that ending (e.g. win/lose/draw or a numerical score)
Evaluation functions: $f(n)$

- Evaluates how good a ‘board position’ is
- Based on **static features** of that board alone
- Zero-sum assumption lets us use one function to describe goodness for both players.
  - $f(n)>0$ if MAX is winning in position $n$
  - $f(n)=0$ if position $n$ is tied
  - $f(n)<0$ if MIN is winning in position $n$
- Build using expert knowledge,
  - Tic-tac-toe: $f(n)=($# of 3 lengths open for MAX$)-($# open for MIN$)$

(AIMA 5.4.1)
A Partial Game Tree for Tic-Tac-Toe

\[ f(n) = 8 - 5 = 3 \]

\[ f(n) = 6 - 5 = 1 \]

\[ f(n) = 6 - 3 = 3 \]

\[ f(n) = 6 - 4 = 2 \]

\[ f(n) = 6 - 2 = 4 \]

\[ f(n) = 2 \]

\[ f(n) = 3 \]

\[ f(n) = 4 \]

\[ f(n) = 2 \]

\[ f(n) = 3 \]

\[ f(n) = 2 \]

\[ f(n) = 3 \]

\[ f(n) = 0 \]

\[ f(n) = + \infty \]

\[ f(n) = - \infty \]

\[ f(n) = \# of potential three-lines for X \]

\[ \# of potential three-line for O \]

\[ \infty \]

\[ 0 \]

\[ + \infty \]
Chess Evaluation Functions

- Claude Shannon argued for a chess evaluation function in a 1950 paper

- Alan Turing defined function in 1948:
  \[ f(n) = (\text{sum of A’s piece values}) - (\text{sum of B’s piece values}) \]

- More complex: weighted sum of positional features:
  \[ \sum w_i \text{feature}_i(n) \]

- Deep Blue had >8000 features

### Pieces values for a simple Turing-style evaluation function often taught to novice chess players

<table>
<thead>
<tr>
<th>Piece</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pawn</td>
<td>1.0</td>
</tr>
<tr>
<td>Knight</td>
<td>3.0</td>
</tr>
<tr>
<td>Bishop</td>
<td>3.25</td>
</tr>
<tr>
<td>Rook</td>
<td>5.0</td>
</tr>
<tr>
<td>Queen</td>
<td>9.0</td>
</tr>
</tbody>
</table>

**Positive:** rooks on open files, knights in closed positions, control of the center, developed pieces

**Negative:** doubled pawns, wrong-colored bishops in closed positions, isolated pawns, pinned pieces

*Examples of more complex features*
Some Chess Positions and their Evaluations

White to move
\[ f(n) = (9+3) - (5+5+3.25) = -1.25 \]

So, considering our opponent’s possible responses would be wise.
The Minimax Rule (AIMA 5.2)
The Minimax Rule: “Don’t play hope chess”

Idea: Make the best move for MAX assuming that MIN always replies with the best move for MIN

Easily computed by a recursive process

- The backed-up value of each node in the tree is determined by the values of its children:
  - For a MAX node, the backed-up value is the maximum of the values of its children (i.e. the best for MAX)
  - For a MIN node, the backed-up value is the minimum of the values of its children (i.e. the best for MIN)
The Minimax Procedure

Until game is over:

1. Start with the current position as a MAX node.
2. Expand the game tree a fixed number of ply.
3. Apply the evaluation function to the leaf positions.
4. Calculate back-up values bottom-up.
5. Pick the move assigned to MAX at the root.
6. Wait for MIN to respond.
2-ply Example: Backing up values

This is the move selected by minimax

Evaluation function value
Adversarial Search (Minimax)

- **Minimax search:**
  - A state-space search tree
  - Players alternate turns
  - Compute each node’s **minimax value**: the best achievable utility against a rational (optimal) adversary

```
\[
\begin{align*}
\text{max} & : 5 \\
\text{min} & : 2, 5, 6 \\
\end{align*}
\]
```

**Terminal values:** part of the game

Minimax values: computed recursively
Minimax Implementation

```python
def max_value(state):
    initialize v = -\infty
    for each successor of state:
        v = max(v, min_value(successor))
    return v

def min_value(state):
    initialize v = +\infty
    for each successor of state:
        v = min(v, max_value(successor))
    return v
```
Minimax Implementation

def max-value(state):
    if the state is a terminal state:
        return the state’s utility
    initialize $v = -\infty$
    for each successor of state:
        $v = \max(v, \min-value(successor))$
    return $v$

def min-value(state):
    if the state is a terminal state:
        return the state’s utility
    initialize $v = +\infty$
    for each successor of state:
        $v = \min(v, \max-value(successor))$
    return $v$
def value(state):
    if the state is a terminal state: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v

def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v

def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
def min-value(state):
    if the state is a terminal state:
        return the state’s utility
    initialize $v = +\infty$
    for each successor of state:
        $v = \min(v, \text{max-value(successor)})$
    return $v$

def max-value(state):
    if the state is a terminal state:
        return the state’s utility
    initialize $v = -\infty$
    for each successor of state:
        $v = \max(v, \text{min-value(successor)})$
    return $v$
What if MIN does not play optimally?

1. Definition of optimal play for MAX assumes MIN plays optimally:
   - *Maximizes worst-case outcome* for MAX.
   - (Classic game theoretic strategy)

2. But if MIN does not play optimally, MAX will do even better.
   - This theorem is not hard to prove
Comments on Minimax Search

- Depth-first search with fixed number of ply $m$ as the limit.
  - $O(b^m)$ time complexity – As usual!
  - $O(bm)$ space complexity

- Performance will depend on
  - the quality of the static evaluation function (expert knowledge)
  - depth of search (computing power and search algorithm)

- Differences from normal state space search
  - Looking to make one move only, despite deeper search
  - No cost on arcs – costs from backed-up static evaluation
  - MAX can’t be sure how MIN will respond to his moves

- Minimax forms the basis for other game tree search algorithms.
Alpha-Beta Pruning (AIMA 5.3)
Alpha-Beta Pruning

- A way to improve the performance of the Minimax Procedure
- Basic idea: "If you have an idea which is surely bad, don’t take the time to see how truly awful it is" ~ Pat Winston

- We don’t need to compute the value at this node.
- No matter what this is, it won’t change the value of the root node.
## Alpha-Beta Pruning

- During Minimax, keep track of two additional values:
  - $\alpha$: MAX’s current *lower* bound on MAX’s outcome
  - $\beta$: MIN’s current *upper* bound on MIN’s outcome

- MAX will never allow a move that could lead to a worse score (for MAX) than $\alpha$
- MIN will never allow a move that could lead to a better score (for MAX) than $\beta$

- Therefore, stop evaluating a branch whenever:
  - When evaluating a MAX node: a value $v \geq \beta$ is backed-up
    - MIN will never select that MAX node
  - When evaluating a MIN node: a value $v \leq \alpha$ is found
    - MAX will never select that MIN node
**Alpha-Beta Implementation**

\[\text{\textbf{def min-value(state, }\alpha, \beta)\text{:}}\]
\[
\text{initialize } v = +\infty \\
\text{for each successor of state:} \\
\text{\quad } v = \min(v, \text{value(successor, }\alpha, \beta)) \\
\text{\quad if } v \leq \beta \text{ return } v \\
\text{\quad } \alpha = \max(\alpha, v) \\
\text{return } v
\]

\[\text{\textbf{def max-value(state, }\alpha, \beta)\text{:}}\]
\[
\text{initialize } v = -\infty \\
\text{for each successor of state:} \\
\text{\quad } v = \max(v, \text{value(successor, }\alpha, \beta)) \\
\text{\quad if } v \geq \beta \text{ return } v \\
\text{\quad } \beta = \min(\beta, v) \\
\text{return } v
\]

\(\alpha\): MAX’s best option on path to root
\(\beta\): MIN’s best option on path to root
Alpha-Beta Pruning Properties

- This pruning has **no effect** on minimax value computed for the root!

- **Values of intermediate nodes might be wrong**
  - Important: children of the root may have the wrong value
  - So the most naïve version won’t let you do action selection

- **Good child ordering improves effectiveness of pruning**

- **With “perfect ordering”:**
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…

- This is a simple example of **metareasoning** (computing about what to compute)
Alpha-Beta Pruning

- Based on observation that for all viable paths utility value $f(n)$ will be $\alpha \leq f(n) \leq \beta$

- Initially, $\alpha = -\infty$, $\beta = \infty$

- As the search tree is traversed, the possible utility value window shrinks as $\alpha$ increases, $\beta$ decreases
Alpha-Beta Pruning

- Whenever the current ranges of alpha and beta no longer overlap, it is clear that the current node is a dead end
Games and Adversarial Search II
Alpha-Beta Pruning (AIMA 5.3)

Some slides adapted from Richard Lathrop, USC/ISI, CS 271
Review: The Minimax Rule

Idea: Make the best move for MAX assuming that MIN always replies with the best move for MIN

1. Start with the current position as a MAX node.
2. Expand the game tree a fixed number of ply.
3. Apply the evaluation function to all leaf positions.
4. Calculate back-up values bottom-up:
   - For a MAX node, return the maximum of the values of its children (i.e. the best for MAX)
   - For a MIN node, return the minimum of the values of its children (i.e. the best for MIN)
5. Pick the move assigned to MAX at the root
6. Wait for MIN to respond and REPEAT FROM 1
2-ply Example: Backing up values

This is the move selected by minimax

New point: Actually calculated by DFS!
Minimax Algorithm

function MINIMAX-DECISION(state) returns an action
inputs: state, current state in game
v ← MAX-VALUE(state)
return an action in SUCCESSORS(state) with value v

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

function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← ∞
for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s))
return v
Alpha-Beta Pruning

• A way to improve the performance of the Minimax Procedure

• Basic idea: “If you have an idea which is surely bad, don’t take the time to see how truly awful it is” ~ Pat Winston

• Assuming left-to-right tree traversal:
  • We don’t need to compute the value at this node!
  • No matter what it is it can’t effect the value of the root node.
Alpha-Beta Pruning II

- During Minimax, keep track of two additional values:
  - $\alpha$: current *lower* bound on MAX’s outcome
  - $\beta$: current *upper* bound on MIN’s outcome

- MAX will never choose a move that could lead to a worse score (for MAX) than $\alpha$
- MIN will never choose a move that could lead to a better score (for MAX) than $\beta$

- Therefore, stop evaluating a branch whenever:
  - When evaluating a MAX node: a value $v \geq \beta$ is backed-up
    — MIN will never select that MAX node
  - When evaluating a MIN node: a value $v \leq \alpha$ is found
    — MAX will never select that MIN node
Alpha-Beta Pruning IIIa

- Based on observation that for all viable paths utility value $f(n)$ will be $\alpha \leq f(n) \leq \beta$

- Initially, $\alpha = -\infty$, $\beta = \infty$

- As the search tree is traversed, the possible utility value window shrinks as $\alpha$ increases, $\beta$ decreases
Alpha-Beta Pruning IIIb

- Whenever the current ranges of alpha and beta no longer overlap ($\alpha \geq \beta$), it is clear that the current node is a dead end, so it can be pruned.
Alpha-beta Algorithm: In detail

- Depth first search (usually bounded, with static evaluation)
  - only considers nodes along a single path from root at any time

\[
\begin{align*}
\beta & \\
\downarrow & \\
\alpha &= \text{current lower bound on MAX's outcome} \\
& \quad \text{(initially, } \alpha = -\infty) \\
\beta &= \text{current upper bound on MIN's outcome} \\
& \quad \text{(initially, } \beta = +\infty) \\
\end{align*}
\]

- Pass current values of \(\alpha\) and \(\beta\) **down** to child nodes during search.
- Update values of \(\alpha\) and \(\beta\) during search:
  - MAX updates \(\alpha\) at MAX nodes
  - MIN updates \(\beta\) at MIN nodes
- Prune remaining branches at a node whenever \(\alpha \geq \beta\)
When to Prune

Prune whenever $\alpha \geq \beta$.

- Prune below a Max node when its $\alpha$ value becomes $\geq$ the $\beta$ value of its ancestors.
  - **Max nodes update** $\alpha$ based on children’s returned values.
  - Idea: Player MIN at node above won’t pick that value anyway, since MIN can force a worse value.

- Prune below a Min node when its $\beta$ value becomes $\leq$ the $\alpha$ value of its ancestors.
  - **Min nodes update** $\beta$ based on children’s returned values.
  - Idea: Player MAX at node above won’t pick that value anyway; she can do better.
Pseudocode for Alpha-Beta Algorithm

function ALPHA-BETA-SEARCH(state) returns an action
inputs: state, current state in game
\[ \nu \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \]
return an action in ACTIONS(state) with value \( \nu \)
An Alpha-Beta Example

Do DF-search until first leaf

\[ \alpha, \beta, \text{ initial values} \]

\[ \alpha = -\infty \]

\[ \beta = +\infty \]

\[ \alpha, \beta, \text{ passed to kids} \]

\[ \alpha = -\infty \]

\[ \beta = +\infty \]
MIN updates β, based on kids
MIN updates $\beta$, based on kids.
No change.
Alpha-Beta Example (continued)

MAX updates $\alpha$, based on kids.

$\alpha = 3$

$\beta = +\infty$

3 is returned as node value.
Alpha-Beta Example (continued)

\[ a = 3 \]
\[ b = +\infty \]

\[ \alpha = 3 \]
\[ \beta = +\infty \]

\( \alpha, \beta, \text{passed to kids} \)

\[ \alpha = 3 \]
\[ \beta = +\infty \]
Alpha-Beta Example (continued)

\[ a = 3 \]
\[ b = +\infty \]

**MIN updates \( \beta \), based on kids.**

\[ \alpha = 3 \]
\[ \beta = 2 \]
Alpha-Beta Example (continued)

\[
\begin{align*}
\alpha &= 3 \\
\beta &= +\infty
\end{align*}
\]

\[\alpha \geq \beta, \text{ so prune.}\]
MAX updates $\alpha$, based on kids.
No change.

2 is returned as node value.
Alpha-Beta Example (continued)

\[ a = 3 \]
\[ b = +\infty \]

\[ \alpha = 3 \]
\[ \beta = +\infty \]

\[ \alpha, \beta, \text{passed to kids} \]

\[ \alpha = 3 \]
\[ \beta = +\infty \]
Alpha-Beta Example (continued)

\[
\beta = 14
\]

MIN updates $\beta$, based on kids.
Alpha-Beta Example (continued)

MIN updates $\beta$, based on kids.

$\beta = +\infty$

$\alpha = 3$

$\beta = 5$

$\alpha = 3$

$\beta = +\infty$

$\beta = 5$
Alpha-Beta Example (continued)

\[ \alpha = 3 \]
\[ \beta = +\infty \]

2 is returned as node value.
Max now makes it’s best move, as computed by Alpha-Beta
### Alpha-Beta Algorithm Pseudocode

**function** ALPHA-BETA-SEARCH(*state*) **returns** an action

**inputs:** *state*, current state in game

\[ \nu \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty) \]

**return** an action in ACTIONS(*state*) with value \( \nu \)

**function** MAX-VALUE(*state*, \( \alpha \), \( \beta \)) **returns** a utility value

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

\[ \nu \leftarrow -\infty \]

**for** \( a \) in ACTIONS(*state*) **do**

\[ \nu \leftarrow \text{MAX}(\nu, \text{MIN-VALUE} \text{(Result}(s,a), \alpha, \beta)) \]

**if** \( \nu \geq \beta \) **then return** \( \nu \)

\[ \alpha \leftarrow \text{MAX}(\alpha, \nu) \]

**return** \( \nu \)
Alpha-Beta Algorithm II

function MIN-VALUE(state, \( \alpha \), \( \beta \)) returns a utility value

if TERMINAL-TEST(state) then return UTILITY(state)

\( v \leftarrow +\infty \)

for \( a, s \) in SUCCESSORS(state) do

\( v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s, \alpha, \beta)) \)

if \( v \leq \alpha \) then return \( v \)

\( \beta \leftarrow \text{MIN}(\beta, v) \)

return \( v \)
Effectiveness of Alpha-Beta Pruning

- Guaranteed to compute same root value as Minimax
- **Worst case:** no pruning, same as Minimax (O(b^d))
- **Best case:** when each player’s best move is the first option examined, examines only O(b^{d/2}) nodes, allowing to search twice as deep!
When best move is the first examined, examines only $O(b^{d/2})$ nodes….

- So: run Iterative Deepening search, sort by value returned on last iteration.
- So: expand captures first, then threats, then forward moves, etc.

- $O(b^{d/2})$ is the same as having a branching factor of $\sqrt{b}$,
  - Since $(\sqrt{b})^d = b^{d/2}$
  - e.g., in chess go from $b \sim 35$ to $b \sim 6$

- For Deep Blue, alpha-beta pruning reduced the average branching factor from 35-40 to 6, as expected, doubling search depth
Real systems use a few more tricks

- Expand the proposed solution a little farther
  - Just to make sure there are no surprises
- Learn better board evaluation functions
  - E.g., for backgammon
- Learn model of your opponent
  - E.g., for poker
- Do stochastic search
  - E.g., for go
Next time:
Expectimax and Utilities

Read AIMA Chapter 16.1-16.3