Game Playing State-of-the-Art


Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

Go: Human champions are being beaten. In go, classically use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

Pacman
Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!
Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame.  
2007: Checkers solved!

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame.  
2007: Checkers solved! 

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic. 

Go: Human champions are being beaten. In go, \( b > 300 \)! Classically use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.
Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame.
2007: Checkers solved!

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

Go: Human champions are being beaten. In go, \( b > 300 \)! Classically use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

Pacman
Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

Go: Human champions are being beaten. In go, \( b > 300 \)! Classically use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

Pacman
Behavior from Computation

Behavior from Computation


Demo: mysterypacman(L6D1)
Video of Demo Mystery Pacman
Adversarial Games

Types of Games

Types of Games

Many different kinds of games!

Many different kinds of games!

Axes:

Types of Games


Many different kinds of games!

Axes:
- Deterministic or stochastic?
Types of Games


Many different kinds of games!

Axes:
- Deterministic or stochastic?
- One, two, or more players?
Types of Games


Many different kinds of games!

Axes:
- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
Types of Games

Many different kinds of games!

Axes:
- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information (can you see the state)?
Types of Games


Many different kinds of games!

Axes:
- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information (can you see the state)?
Types of Games


Many different kinds of games!

Axes:
- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information (can you see the state)?

Want algorithms for calculating a strategy (policy) which recommends a move from each state.
Deterministic Games

Deterministic Games


Many possible formalizations, one is:
- States: S (start at s0)
Deterministic Games


Many possible formalizations, one is:
- States: $S$ (start at $s_0$)
- Players: $P = \{1...N\}$
Deterministic Games


Many possible formalizations, one is:
- States: $S$ (start at $s_0$)
- Players: $P = \{1 \ldots N\}$ (usually take turns).
Deterministic Games


Many possible formalizations, one is:
- States: $S$ (start at $s_0$)
- Players: $P = \{1\ldots N\}$ (usually take turns).
- Actions: $A$ (may depend on player / state)
Deterministic Games

Many possible formalizations, one is:
- States: $S$ (start at $s_0$)
- Players: $P = \{1\ldots N\}$ (usually take turns).
- Actions: $A$ (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$. 

Many possible formalizations, one is:

- States: $S$ (start at $s_0$)
- Players: $P = \{1...N\}$ (usually take turns).
- Actions: $A$ (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$.
- Terminal Test: $S \rightarrow \{t, f\}$
Deterministic Games


Many possible formalizations, one is:

- States: \( S \) (start at \( s_0 \))
- Players: \( P = \{1...N\} \) (usually take turns).
- Actions: \( A \) (may depend on player / state)
- Transition Function: \( S \times A \rightarrow S \).
- Terminal Test: \( S \rightarrow \{t, f\} \)
- Terminal Utilities: \( S \times P \rightarrow R \).
Many possible formalizations, one is:

- States: $S$ (start at $s_0$)
- Players: $P = \{1\ldots N\}$ (usually take turns).
- Actions: $A$ (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$
- Terminal Test: $S \rightarrow \{t, f\}$
- Terminal Utilities: $S \times P \rightarrow R$. 

Deterministic Games


Many possible formalizations, one is:
- States: $S$ (start at $s_0$)
- Players: $P = \{1\ldots N\}$ (usually take turns).
- Actions: $A$ (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$.
- Terminal Test: $S \rightarrow \{t,f\}$
- Terminal Utilities: $S \times P \rightarrow R$.

Solution for a player is a policy: $S \rightarrow A$. 
Zero-Sum Games

Zero-Sum Games

Zero-Sum Games

Agents have opposite utilities
(values on outcomes)

Zero-Sum Games

Agents have opposite utilities (values on outcomes)

Let us think of a single value that one maximizes and the other minimizes

Zero-Sum Games

Agents have opposite utilities (values on outcomes)
Let's us think of a single value that one maximizes and the other minimizes
Adversarial, pure competition

Zero-Sum Games

- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

General Games

- Agents have independent utilities (values on outcomes)
Zero-Sum Games

Agents have opposite utilities (values on outcomes)
Let us think of a single value that one maximizes and the other minimizes
Adversarial, pure competition

General Games
Agents have independent utilities (values on outcomes)
Cooperation, indifference, competition, and more are all possible
Zero-Sum Games

- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition


General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- More later on non-zero-sum games
Zero-Sum Games

Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

General Games
- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- More later on non-zero-sum games

Adversarial Search
Single-Agent Trees
Value of Game Tree

Value of a State: utility of best achievable outcome from state.
Value of Game Tree

Value of a State: utility of best achievable outcome from state.
Value of Game Tree

Value of a State: utility of best achievable outcome from state.
Value of Game Tree

Value of a State: utility of best achievable outcome from state.
Value of Game Tree

Value of a State: utility of best achievable outcome from state.
Value of Game Tree

Value of a State: utility of best achievable outcome from state.
Value of Game Tree

Value of a State: utility of best achievable outcome from state.

```
| Terminal States: | V(s) = known |
| Non terminal states: | V(s) = max s' ∈ kids(s) V(s') |
```
Value of Game Tree

Value of a State: utility of best achievable outcome from state.
Value of a State: utility of best achievable outcome from state.
Value of a State: utility of best achievable outcome from state.

Terminal States:

Non terminal states:

\[ V(s) = \text{known} \]

\[ V(s) = \max_{s' \in \text{kids}(s)} V(s') \]
Value of Game Tree

Value of a State: utility of best achievable outcome from state.

Non terminal states:
\[ V(s) = \max_{s' \in \text{kids}(s)} V(s') \]

Terminal States:
\[ V(s) = \text{known} \]

Diagram:
- Terminal states have known values.
- Non-terminal states have values calculated as the maximum of all possible child states.
Adversarial Game Tree
Adversarial Game Tree
Adversarial Game Tree
Adversarial Game Tree
Adversarial Game Tree
Adversarial Game Tree
Adversarial Game Tree
Adversarial Game Tree
Adversarial Game Tree
Adversarial Game Tree
Minimax Values

Max states: $V(s) = \max_{s' \in \text{succ}(s)} V(s')$

Min states: $V(s) = \min_{s' \in \text{succ}(s)} V(s')$

Terminal States: $V(s) = \text{known}$
Minimax Values

Min states: $V(s) = \min_{s' \in \text{succ}(s)} V(s')$

Max states: $V(s) = \max_{s' \in \text{succ}(s)} V(s')$

Terminal States: $V(s) = \text{known}$

States Under Agent’s Control.
Minimax Values

Max states: \( V(s) = \max_{s' \in \text{succ}(s)} V(s') \)

Min states: \( V(s) = \min_{s' \in \text{succ}(s)} V(s') \)

Terminal States: \( V(s) = \text{known} \)

States Under Agent’s Control. Max.
Minimax Values

Min states: \( V(s) = \min_{s' \in \text{succ}(s)} V(s') \)

Max states: \( V(s) = \max_{s' \in \text{succ}(s)} V(s') \)

Terminal States: \( V(s) = \text{known} \)

States Under Agent’s Control. Max.

Terminal States.
Minimax Values

Min states: \( V(s) = \min_{s' \in \text{succ}(s)} V(s') \)

Max states: \( V(s) = \max_{s' \in \text{succ}(s)} V(s') \)

Terminal States: \( V(s) = \text{known} \)

States Under Agent’s Control. Max.
Terminal States.
States Under Opponent’s Control.
Minimax Values

Min states: $V(s) = \min_{s' \in \text{succ}(s)} V(s')$

Max states: $V(s) = \max_{s' \in \text{succ}(s)} V(s')$

Terminal States: $V(s) = \text{known}$

States Under Agent’s Control. Max.
Terminal States.
States Under Opponent’s Control. Min.
Tic-Tac-Toe Game Tree
Adversarial Search (Minimax)

Minimax values: computed recursively.

Terminal values: part of the game.
Adversarial Search (Minimax)

Minimax values: computed recursively.

Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers

Terminal values: part of the game.
Minimax values: computed recursively.

Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers
- One player maximizes result

Terminal values: part of the game.
Adversarial Search (Minimax)

Minimax values: computed recursively.

Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Terminal values: part of the game.
Adversarial Search (Minimax)

Minimax values: computed recursively.

Terminal values: part of the game.

Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

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
Adversarial Search (Minimax)

Minimax values: computed recursively.

Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Minimax search:
- A state-space search tree

Terminal values: part of the game.
Adversarial Search (Minimax)

Minimax values: computed recursively.

Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Minimax search:
- A state-space search tree
- Players alternate turns

Terminal values: part of the game.
Adversarial Search (Minimax)

Minimax values: computed recursively.

Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

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

Terminal values: part of the game.
Adversarial Search (Minimax)

Minimax values: computed recursively.

Terminal values: part of the game.

Deterministic, zero-sum games:
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

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
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

def min-value(state):

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

def max-value(state):

def value(state):
  if the state is terminal: 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)
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

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

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

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

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

def value(state):
    if the state is terminal: 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)
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

def min-value(state):
    initialize \( v = +\infty \)
    for each successor \( s \) of \( \text{state} \):
        \( v = \min(v, \text{max-value}(s)) \)
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

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

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

```python
def value(state):
    if the state is terminal: 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)
```
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

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

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

def value(state):
    if the state is terminal: 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)
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

**def min-value(state):**
- initialize \( v = +\infty \)
- for each successor \( s \) of state:
  - \( v = \min(v, \text{max-value}(s)) \)
- return \( v \)

**def max-value(state):**
- initialize \( v = -\infty \)
- for each successor \( s \) of state:
  - \( v = \max(v, \text{min-value}(s)) \)
- return \( v \)
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

**def min-value(state):**
- initialize \( v = +\infty \)
- for each successor \( s \) of state:
  - \( v = \min(v, \max-value(s)) \)
- return \( v \)

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

**def max-value(state):**
- initialize \( v = -\infty \)
- for each successor \( s \) of state:
  - \( v = \max(v, \min-value(s)) \)
- return \( v \)

**def value(state):**
- if the state is terminal: return the state's utility
- if the next agent is MAX: return \( \max-value(state) \)
- if the next agent is MIN: return \( \min-value(state) \)
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

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

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

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

```python
def value(state):
    if the state is terminal: 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)
```
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

```python
def min-value(state):
    initialize v = +\infty
    for each successor s of state:
        v = min(v, max-value(s))
    return v
```

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

```python
def max-value(state):
    initialize v = -\infty
    for each successor s of state:
        v = max(v, min-value(s))
    return v
```
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

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

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

```python

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

def value(state):
    if the state is terminal: 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)
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

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

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

def max-value(state):
    initialize v = -\infty.
    for each successor s of state:
        v = max(v, min-value(s))
    return v

def value(state):
    if the state is terminal: 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)
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

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

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

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

```python
def value(state):
    if the state is terminal: 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)
```
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

**def** min-value(state):
    initialize \( v = +\infty \)
    for each successor \( s \) of state:
        \( v = \min(v, \text{max-value}(s)) \)
    return \( v \)

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

**def** max-value(state):
    initialize \( v = -\infty \)
    for each successor \( s \) of state:
        \( v = \max(v, \text{min-value}(s)) \)
    return \( v \)

def value(state):
    if the state is terminal: return the state’s utility
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

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

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

def value(state):
    if the state is terminal: 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)
```
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

```python
def min_value(state):
    initialize v = +\infty
    for each successor \textit{s} of state:
        v = min(v, max_value(s))
    return v

def max_value(state):
    initialize v = -\infty
    for each successor \textit{s} of state:
        v = max(v, min_value(s))
    return v

def value(state):
    if the state is terminal: 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)
```
Minimax Implementation

\[ V(s) = \min_{s' \in \text{succ}(s)} V(s') \]

\[ V(s') = \max_{s \in \text{succ}(s')} V(s) \]

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

def max-value(state):
    initialize v = -\infty.
    for each successor s of state:
        v = max(v, min-value(s))
    return v

def value(state):
    if the state is terminal: 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)
Minimax Example
Minimax Example
Minimax Example
Minimax Example
Minimax Example

```
5
/  \
3   2
 / \
3 12 8
 / \
4 2
 / \
6 14
 / \
5 20
```
Minimax Efficiency

How efficient is minimax?

Just like (exhaustive) DFS:

- Time: $O(b^m)$
- Space: $O(bm)$

Example: For chess, $b \approx 35$, $m \approx 100$

Exact solution is completely infeasible.

But, do we need to explore the whole tree?
Minimax Efficiency

How efficient is minimax?

- Just like (exhaustive) DFS
Minimax Efficiency

How efficient is minimax?
- Just like (exhaustive) DFS
- Time: $O(b^m)$
Minimax Efficiency

How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$
Minimax Efficiency

How efficient is minimax?
- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

Example: For chess, $b \approx 35, m \approx 100$

Exact solution is completely infeasible.
Minimax Efficiency

How efficient is minimax?
- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

Example: For chess, $b \approx 35, m \approx 100$
- Exact solution is completely infeasible.
How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

Example: For chess, $b \approx 35$, $m \approx 100$

- Exact solution is completely infeasible.
- But, do we need to explore the whole tree?
Minimax Efficiency

How efficient is minimax?
- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

Example: For chess, $b \approx 35$, $m \approx 100$
- Exact solution is completely infeasible.
- But, do we need to explore the whole tree?
Minimax Properties

Minimax values: computed recursively.

Terminal values: part of the game.
Minimax Properties

Minimax values: computed recursively.

Terminal values: part of the game.
Minimax Properties

Minimax values: computed recursively.

Terminal values: part of the game.
Minimax Properties

Minimax values: computed recursively.

Terminal values: part of the game.
Minimax Properties

Minimax values: computed recursively.

Terminal values: part of the game.

Optimal against a perfect player.
Minimax Properties

Minimax values: computed recursively.

Terminal values: part of the game.

Optimal against a perfect player. Otherwise?
Minimax Properties

Minimax values: computed recursively.

Terminal values: part of the game.

Optimal against a perfect player. Otherwise?

Demo: \texttt{minvsexp}(L6D2, L6D3)
Video of Demo Min vs. Exp (Min)
Video of Demo Min vs. Exp (Exp)
Resource Limits
Problem: In realistic games, cannot search to leaves!

Example:
Suppose we have 100 seconds, can explore 10K nodes/sec.
So can check 1M nodes per move.↓↑- reaches about depth 8 – decent chess program

Guarantee of optimal play is gone. More plies makes a BIG difference

Use iterative deepening for an anytime algorithm
Problem: In realistic games, cannot search to leaves!

Guarantee of optimal play is gone.

More plies makes a BIG difference

Use iterative deepening for an anytime algorithm
Resource Limits

Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

![Diagram of a search tree with depth-limited search highlighted]
Resource Limits

Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

- Instead, search to limited depth in the tree.

![Diagram of a tree with limited depth]

Suppose we have 100 seconds, can explore 10K nodes / sec

So can check 1M nodes per move.

↓ , ↑ - reaches about depth 8 – decent chess program

Guarantee of optimal play is gone.

More plies makes a BIG difference

Use iterative deepening for an anytime algorithm
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:
- Suppose we have 100 seconds, can explore 10K nodes / sec
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use 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.
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use 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.
- ↓,↑- reaches about depth 8
  - decent chess program
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use 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.
- ↓,↑- reaches about depth 8
  – decent chess program
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use 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.
- \( \downarrow, \uparrow \) reaches about depth 8
  – decent chess program

Guarantee of optimal play is gone.
Resource Limits

Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use 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.
- ↓, ↑- reaches about depth 8
  - decent chess program

Guarantee of optimal play is gone.
More plies makes a BIG difference
Resource Limits

Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search
- Instead, search to limited depth in the tree.
- Use 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.
- ↓,↑- reaches about depth 8
  - decent chess program

Guarantee of optimal play is gone.
More plies makes a BIG difference
Use iterative deepening for an anytime algorithm
Depth Matters

Evaluation functions are always imperfect. The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters. An important example of the tradeoff between complexity of features and complexity of computation:

Demo: depthlimited (L6, D4), (L6, D5)
Depth Matters

Evaluation functions are always imperfect
Evaluation functions are always imperfect.
The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters.
Depth Matters

Evaluation functions are always imperfect.
The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters.

An important example of the tradeoff between complexity of features and complexity of computation.
Evaluation functions are always imperfect
The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
An important example of the tradeoff between complexity of features and complexity of computation

*Demo: depthlimited(\text{L6D4, L6D5})*
Video of Demo Limited Depth (2)
Video of Demo Limited Depth (10)
Evaluation Functions
Evaluation Functions

Evaluation functions score non-terminals in depth-limited search. An ideal function returns the actual minimax value of the position. In practice, evaluation functions are typically a weighted linear sum of features:

$$ w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) $$

Example: $f_1(s) = (\text{num white queens} - \text{num black queens})$, etc.

[Chess board diagrams showing different scenarios]
Evaluation Functions

Evaluation functions score non-terminals in depth-limited search
Evaluation Functions

Evaluation functions score non-terminals in depth-limited search
Ideal function: returns the actual minimax value of the position
Evaluation Functions

Evaluation functions score non-terminals in depth-limited search.

Ideal function: returns the actual minimax value of the position.

In practice: typically a weighted linear sum of features:

\[ w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s). \]
Evaluation Functions

Evaluation functions score non-terminals in depth-limited search
Ideal function: returns the actual minimax value of the position
In practice: typically weighted linear sum of features:
\[ w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s). \]
Evaluation Functions

Evaluation functions score non-terminals in depth-limited search
Ideal function: returns the actual minimax value of the position
In practice: typically weighted linear sum of features:
\[ w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s). \]
Example: \( f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.
Evaluation for Pacman
Evaluation for Pacman

thrashingd = 2, thrashingd = 2 (fixed evaluation function), smartghosts coordinate (L6D6, 7, 8, 10)
Evaluation for Pacman

thrashingd = 2, thrashingd = 2 (fixed evaluation function), smartghosts coordinate (L6, D6, 7, 8, 10)
Evaluation for Pacman

thrashing_d = 2, thrashing_d = 2 (fixed evaluation function), smartghosts coordinate (L, D, 6, 7, 8, 10)
Evaluation for Pacman

Demo: \( thrashingd = 2, \text{thrashingd} = 2(\text{fixedevaluationfunction}), \text{smartghostscoordinate} = (L_6, D_6, 7, 8, 10) \)
Video of Demo Thrashing (d=2)
Why Pacman Starves

He knows his score will go up by eating the dot now (west, east)

He knows his score will go up just as much by eating the dot later (east, west)

There are no point-scoring opportunities after eating the dot (within the horizon, two here)

Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!
Why Pacman Starves

- He knows his score will go up by eating the dot now (west, east).
- He knows his score will go up just as much by eating the dot later (east, west).
- There are no point-scoring opportunities after eating the dot (within the horizon, two here).
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!
Why Pacman Starves

A danger of replanning agents!

- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!
Why Pacman Starves

- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!
A danger of replanning agents!
- He knows his score will go up by eating the dot now (west, east)
Why Pacman Starves

A danger of replanning agents!

- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
Why Pacman Starves

A danger of replanning agents!
- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
Why Pacman Starves

A danger of replanning agents!
- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!
Why Pacman Starves

A danger of replanning agents!
- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!
Video of Demo Thrashing – Fixed (d=2)
Video of Demo Smart Ghosts (Coordination)
Video of Demo Smart Ghosts (Coordination) – Zoomed In
Game Tree Pruning
Minimax Example
Minimax Example
Minimax Example
Minimax Example
Minimax Example
Minimax Pruning

![Diagram of a minimax pruning tree with nodes labeled 3, 12, 8, 2, 14, 5, and 20.](image)
Minimax Pruning
Minimax Pruning
Minimax Pruning
Minimax Pruning
Minimax Pruning
Minimax Pruning
Alpha-Beta Pruning

Computing MIN-VALUE at some node. Looping over node's children. node's estimate of min is dropping. Who cares about node's value? MAX = "best MAX value on path to root." If node < MAX, than MAX will never choose it. So search "prunes" other children. Symmetric for MAX version.
Alpha-Beta Pruning

General configuration (MIN version)

- Computing MIN-VALUE at some node $n$. 

![Diagram of Alpha-Beta Pruning with MIN version, showing the process of computing MIN-VALUE at a node and pruning unnecessary branches.]
Alpha-Beta Pruning

General configuration (MIN version)
- Computing MIN-VALUE at some node $n$.
- Looping over $n$’s children
Alpha-Beta Pruning

General configuration (MIN version)
- Computing MIN-VALUE at some node \( n \).
- Looping over \( n \)'s children
- \( n \)'s estimate of min is dropping
Alpha-Beta Pruning

General configuration (MIN version)
- Computing MIN-VALUE at some node $n$.
- Looping over $n$’s children
- $n$’s estimate of min is dropping
- Who cares about $n$’s value? MAX
Alpha-Beta Pruning

General configuration (MIN version)
- Computing MIN-VALUE at some node $n$.
- Looping over $n$'s children
- $n$'s estimate of min is dropping
- Who cares about $n$'s value? MAX
- $a = \text{“ best MAX value on path to root.”}
Alpha-Beta Pruning

General configuration (MIN version)
- Computing MIN-VALUE at some node $n$.
- Looping over $n$’s children
- $n$’s estimate of min is dropping
- Who cares about $n$’s value? MAX
- $a = “$ best MAX value on path to root.”
- If $n < a$, than MAX will never choose it.
Alpha-Beta Pruning

General configuration (MIN version)
- Computing MIN-VALUE at some node $n$.
- Looping over $n$’s children
- $n$’s estimate of min is dropping
- Who cares about $n$’s value? MAX
- $a =$ “best MAX value on path to root.”
- If $n < a$, than MAX will never choose it.
  So search “prunes” other children.
General configuration (MIN version)
- Computing MIN-VALUE at some node $n$.
- Looping over $n$’s children
- $n$’s estimate of min is dropping
- Who cares about $n$’s value? MAX
- $a = “$ best MAX value on path to root.”
- If $n < a$, than MAX will never choose it.
  So search “prunes” other children.
Alpha-Beta Pruning

General configuration (MIN version)
- Computing MIN-VALUE at some node $n$.
- Looping over $n$’s children
- $n$’s estimate of min is dropping
- Who cares about $n$’s value? MAX
- $a = \text{“best MAX value on path to root.”}$
- If $n < a$, than MAX will never choose it. So search “prunes” other children.

Symmetric for MAX version.
\( \alpha \) - MAX's best option on path to root.
Alpha-Beta Implementation

\(\alpha\) - MAX’s best option on path to root.
\(\beta\) - MIN’s best option on path to root.
**Alpha-Beta Implementation**

\[ \alpha - \text{MAX's best option on path to root.} \]
\[ \beta - \text{MIN's best option on path to root.} \]

```python
def min-value(state, \alpha, \beta):
    initialize v = +\infty.
    for each successor \( s \) of state:
        v = min(v, value(s, \alpha, \beta))
        if v \leq \alpha return v
    \beta = min(\beta, v)
    return v
```
Alpha-Beta Implementation

\[ \alpha - \text{MAX's best option on path to root.} \]
\[ \beta - \text{MIN's best option on path to root.} \]

def min-value(state, \(\alpha, \beta\)):
    initialize \(v = +\infty\).
Alpha-Beta Implementation

\[ \alpha \] - MAX’s best option on path to root.
\[ \beta \] - MIN’s best option on path to root.

```python
def min-value(state, \alpha, \beta):
    initialize \( v = +\infty \).
    for each successor \( s \) of state:
        \[ v = \min(v, \text{value}(s, \alpha, \beta)) \]
        if \( v \leq \alpha \)
            return \( v \)
    \( \beta = \min(\beta, v) \)
    return \( v \)
```
**Alpha-Beta Implementation**

\( \alpha \) - MAX’s best option on path to root.
\( \beta \) - MIN’s best option on path to root.

```python
def min-value(state, \( \alpha, \beta \)):
    initialize \( v = +\infty \).
    for each successor \( s \) of state:
        \( v = \min(v, \text{value}(s, \alpha, \beta)) \)
```

```python
def max-value(state, \( \alpha, \beta \)):
    initialize \( v = -\infty \).
    for each successor \( s \) of state:
        \( v = \max(v, \text{value}(s, \alpha, \beta)) \)
```

**Alpha-Beta Implementation**

\( \alpha \) - MAX’s best option on path to root.

\( \beta \) - MIN’s best option on path to root.

```python
def min_value(state, \alpha, \beta):
    initialize \( v = + \infty \).
    for each successor \( s \) of state:
        \( v = \min(v, value(s, \alpha, \beta)) \)
        if \( v \leq \alpha \) return \( v \)
```

```python
def max_value(state, \alpha, \beta):
    initialize \( v = - \infty \).
    for each successor \( s \) of state:
        \( v = \max(v, value(s, \alpha, \beta)) \)
        if \( v \leq \alpha \) return \( v \)
        \( \alpha = \min(\alpha, v) \)
    return \( v \)
```
**Alpha-Beta Implementation**

\( \alpha \) - MAX’s best option on path to root.

\( \beta \) - MIN’s best option on path to root.

```python
def min-value(state, \( \alpha \), \( \beta \)):
    initialize \( v = +\infty \).
    for each successor \( s \) of state:
        \( v = \min(v, \text{value}(s, \alpha, \beta)) \)
        if \( v \leq \alpha \) return \( v \)
    \( \alpha = \min(\alpha, v) \)
    return \( v \)
```

```python
def max-value(state, \( \alpha \), \( \beta \)):
    initialize \( v = -\infty \).
    for each successor \( s \) of state:
        \( v = \max(v, \text{value}(s, \alpha, \beta)) \)
        if \( v \leq \alpha \) return \( v \)
    \( \alpha = \min(\alpha, v) \)
    return \( v \)
```
α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state, α, β):
    initialize v = +∞.
    for each successor s of state:
        v = min(v, value(s, α, β))
        if v ≤ α return v
    β = min(β, v)
return v
α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state, α, β):
    initialize v = +∞.
    for each successor s of state:
        v = min(v, value(s, α, β))
        if v ≤ α return v
    β = min(β, v)
    return v

def max-value(state, α, β):
    initialize v = −∞.
    for each successor s of state:
        v = max(v, value(s, α, β))
        if v ≤ α return v
    α = min(α, v)
    return v
Alpha-Beta Implementation

$\alpha$ - MAX’s best option on path to root.
$\beta$ - MIN’s best option on path to root.

```python
def min-value(state, $\alpha$, $\beta$):
    initialize $v = +\infty$.
    for each successor $s$ of state:
        $v = \min(v, \text{value}(s, \alpha, \beta))$
        if $v \leq \alpha$ return $v$
    $\beta = \min(\beta, v)$
    return $v$
def max-value(state, $\alpha$, $\beta$):
    initialize $v = -\infty$.
    for each successor $s$ of state:
        $v = \max(v, \text{value}(s, \alpha, \beta))$
        if $v \leq \alpha$ return $v$
    $\alpha = \min(\alpha, v)$
    return $v$```
**Alpha-Beta Implementation**

$\alpha$ - MAX’s best option on path to root.

$\beta$ - MIN’s best option on path to root.

```python
def min_value(state, \alpha, \beta):
    initialize $v = +\infty$.
    for each successor $s$ of state:
        $v = \min(v, value(s, \alpha, \beta))$
        if $v \leq \alpha$ return $v$
    $\beta = \min(\beta, v)$
    return $v$

def max_value(state, \alpha, \beta):
    initialize $v = -\infty$.
    for each successor $s$ of state:
        $v = \max(v, value(s, \alpha, \beta))$
        if $v \leq \alpha$ return $v$
    $\alpha = \min(\alpha, v)$
    return $v$
```
α - MAX’s best option on path to root.
β - MIN’s best option on path to root.

def min-value(state, α, β):
    initialize v = +∞.
    for each successor s of state:
        v = min(v, value(s, α, β))
        if v ≤ α return v
    β = min(β, v)
    return v

def max-value(state, α, β):
    initialize v = −∞.
    for each successor s of state:
        v = max(v, value(s, α, β))
        if v ≥ α return v
    α = min(α, v)
    return v
Alpha-Beta Implementation

\( \alpha \) - MAX’s best option on path to root.

\( \beta \) - MIN’s best option on path to root.

```python
def min-value(state, \( \alpha \), \( \beta \)):
    initialize \( v = +\infty \).
    for each successor \( s \) of state:
        \( v = \min(v, \text{value}(s, \alpha, \beta)) \)
        if \( v \leq \alpha \) return \( v \)
        \( \beta = \min(\beta, v) \)
    return \( v \)
def max-value(state, \( \alpha \), \( \beta \)):
    initialize \( v = -\infty \).
    for each successor \( s \) of state:
        \( v = \max(v, \text{value}(s, \alpha, \beta)) \)
        if \( v \leq \alpha \) return \( v \)
    \( \alpha = \min(\alpha, v) \)
    return \( v \)
```
**Alpha-Beta Implementation**

\( \alpha \) - MAX’s best option on path to root.

\( \beta \) - MIN’s best option on path to root.

```python
def min_value(state, \( \alpha \), \( \beta \)):
    initialize \( v = +\infty \).
    for each successor \( s \) of state:
        \( v = \min(v, \text{value}(s, \alpha, \beta)) \)
        if \( v \leq \alpha \) return \( v \)
    \( \beta = \min(\beta, v) \)
    return \( v \)

def max_value(state, \( \alpha \), \( \beta \)):
    initialize \( v = -\infty \).
    for each successor \( s \) of state:
        \( v = \max(v, \text{value}(s, \alpha, \beta)) \)
        if \( v \leq \alpha \) return \( v \)
    \( \alpha = \min(\alpha, v) \)
    return \( v \)
```
**Alpha-Beta Implementation**

\(\alpha\) - MAX’s best option on path to root.

\(\beta\) - MIN’s best option on path to root.

```python
def min_value(state, \(\alpha\), \(\beta\)):
    initialize \(v = +\infty\).
    for each successor \(s\) of state:
        \(v = \min(v, \text{value}(s, \alpha, \beta))\)
        if \(v \leq \alpha\) return \(v\)
    \(\beta = \min(\beta, v)\)
    return \(v\)

def max_value(state, \(\alpha\), \(\beta\)):
    initialize \(v = -\infty\).
    for each successor \(s\) of state:
        \(v = \max(v, \text{value}(s, \alpha, \beta))\)
        if \(v \leq \alpha\) return \(v\)
    \(\alpha = \min(\alpha, v)\)
    return \(v\)
```

Alpha-Beta Pruning Properties

This pruning has no effect on minimax value computed for the root! Values of intermediate nodes might be wrong. Important: root's children may be wrong. Most naive version not for action selection. Good child ordering improves effectiveness. 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).
This pruning has no effect on minimax value computed for the root!

```
max

+---+---+---+
| 10| 10|  0 |
```

Important: root's children may be wrong → most naive version not for action selection

Good child ordering improves effectiveness

With "perfect ordering":

\[
\text{Time complexity drops to } O\left(\frac{b^m}{2}\right)
\]

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 Properties

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

Values of intermediate nodes might be wrong

- Important: root’s children may be wrong
Alpha-Beta Pruning Properties

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

Values of intermediate nodes might be wrong
- Important: root’s children may be wrong
- → most naive version not for action selection
Alpha-Beta Pruning Properties

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

Values of intermediate nodes might be wrong

- Important: root’s children may be wrong
- → most naive version not for action selection
This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong
- Important: root’s children may be wrong
- → most naive version not for action selection

Good child ordering improves effectiveness
This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong
- Important: root’s children may be wrong
- \( \rightarrow \) most naive version not for action selection

Good child ordering improves effectiveness

With “perfect ordering”:
- Time complexity drops to \( O(b^{m/2}) \)
Alpha-Beta Pruning Properties

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

Values of intermediate nodes might be wrong
- Important: root’s children may be wrong
- $\rightarrow$ most naive version not for action selection

Good child ordering improves effectiveness

With “perfect ordering”:
- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth!
Alpha-Beta Pruning Properties

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

Values of intermediate nodes might be wrong
- Important: root’s children may be wrong
- → most naive version not for action selection

Good child ordering improves effectiveness

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

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

Values of intermediate nodes might be wrong
- Important: root’s children may be wrong
- $\rightarrow$ most naive version not for action selection

Good child ordering improves effectiveness

With “perfect ordering”:
- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth!
- Full search of, e.g. chess, is still hopeless...
This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong
- Important: root’s children may be wrong
- → most naive version not for action selection

Good child ordering improves effectiveness

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 Quiz
Next Time: Uncertainty!