15-466
Computer Game Programming

Board Games

Maxim Likhachev
Robotics Institute
Carnegie Mellon University
There Are Still Board Games…
Classes of Board Games

• Two players vs. more than two players (e.g., chess vs. poker)

• Perfect information vs. imperfect information (e.g., chess vs. poker)
  - everybody knows ALL the whole game state

• Zero-sum vs. non-zero-sum (e.g., chess vs. maximizing wealth by cooperating)
  - zero-sum refers to “what your opponent looses, you gain” (gain=-loss)

What class do most board games belong to?
Game Tree

- For Two-opponent Zero-sum Perfect Information Game

Game tree for Tic-Tac-Toe:

- At leaves: +1 if the leaf is a win for X, -1 if the leaf is a win for O

Indicates that other unshown options exist

from “Artificial Intelligence for Games” by I. Millington & J. Funge
Game Tree

• For Two-opponent Zero-sum Perfect Information Game

Abstract game tree:

from “Artificial Intelligence for Games” by I. Millington & J. Funge
Game Tree

• For Two-opponent Zero-sum Perfect Information Game
• If branches can merge (can arrive to the same state via multiple paths), then the game has transpositions
  - no longer a tree

*Example of games with transpositions?*
Minimax for Games without Transposition

• Recursive approach to computing the value of the game assuming we and the opponent are acting optimally

GetMinimaxValue(state s at level d)
  if d = maximum depth, then return evaluation score of s
  if state s is a leaf state, then return the score of s
  if d is even, then
    return MAX_{s’ \in \text{succ}(s)} (GetMinimaxValue (s’ at level d+1))
  if d is odd, then
    return MIN_{s’ \in \text{succ}(s)} (GetMinimaxValue (s’ at level d+1))

What do we do with these values afterwards?
Minimax for Games without Transposition

- Recursive approach to computing the value of the game assuming we and the opponent are acting optimally

```plaintext
GetMinimaxValue(state s at level d)
  if d = maximum depth, then return evaluation score of s
  if state s is a leaf state, then return the score of s
  if d is even, then
    return MAX_{s' in succ(s)} (GetMinimaxValue (s’ at level d+1))
  if d is odd, then
    return MIN_{s' in succ(s)} (GetMinimaxValue (s’ at level d+1))
```

Anything wrong with this assumption?
Negamax for Games without Transposition

• Recursive approach to computing the value of the game assuming we and the opponent are acting optimally.

GetNegaMaxValue(state s at level d)

  if \( d = \text{maximum depth} \), then return \((-1)^d\) * evaluation score of s

  if state s is a leaf state, then return \((-1)^d\) * score of s

  return \( \max_{s' \in \text{succ}(s)} (-1 \times \text{GetNegaMaxValue (s’ at level d+1)}) \)

Avoids alternating min-max operations

from “Artificial Intelligence for Games” by I. Millington & J. Funge
Alpha-Beta Pruning for Minimax

- Alpha $\alpha$: lower bound on the optimal score
- Beta $\beta$: upper bound on the optimal score
- No need to evaluate any branch whose value is guaranteed to be outside of ($\alpha$, $\beta$) range
Alpha-Beta Pruning for Minimax

- Alpha $\alpha$: lower bound on the optimal score
- Beta $\beta$: upper bound on the optimal score
- No need to evaluate any branch whose value is guaranteed to be outside of $(\alpha, \beta)$ range

$$\begin{align*}
5 & \quad (-\infty; \infty) \\
\text{max} & \\
\end{align*}$$

$$\begin{align*}
\text{min} & \\
5 & \quad (-\infty; \infty) \\
\text{max} & \\
\end{align*}$$

$$\begin{align*}
\text{min} & \\
4 & \quad (-\infty; \infty) \\
\text{max} & \\
\end{align*}$$

$$\begin{align*}
\text{max} & \\
5 & \quad (-\infty; \infty) \\
\text{max} & \\
3 & \quad (-\infty; \infty) \\
\text{max} & \\
5 & \quad (-\infty; \infty) \\
\text{max} & \\
3 & \quad (-\infty; \infty) \\
\text{max} & \\
9 & \quad (-\infty; \infty) \\
\text{max} & \\
9 & \quad (-\infty; \infty) \\
\text{max} & \\
4 & \quad (-\infty; \infty) \\
\text{max} & \\
4 & \quad (-\infty; \infty) \\
\text{max} & \\
9 & \quad (-\infty; \infty) \\
\text{max} & \\
10 & \quad (-\infty; \infty) \\
\text{max} & \\
\end{align*}$$
Alpha-Beta Pruning for Minimax

- Alpha $\alpha$: lower bound on the optimal score
- Beta $\beta$: upper bound on the optimal score
- No need to evaluate any branch whose value is guaranteed to be outside of ($\alpha$, $\beta$) range

```
\begin{array}{c}
\text{max} \\
5 \quad (-\infty; \infty) \\
\text{min} \\
5 \quad (-\infty; \infty) \\
\text{max} \\
5 \quad (-\infty; \infty) \\
\text{max} \\
5 \quad (-\infty; \infty) \\
\text{max} \\
5 \quad (-\infty; \infty) \\
\text{min} \\
9 \quad (5; \infty) \\
\text{max} \\
3 \quad (5; \infty) \\
\text{max} \\
8 \quad (5; \infty) \\
\end{array}
```
Alpha-Beta Pruning for Minimax

• Alpha $\alpha$: lower bound on the optimal score
• Beta $\beta$: upper bound on the optimal score
• No need to evaluate any branch whose value is guaranteed to be outside of ($\alpha$, $\beta$) range
Alpha-Beta Pruning for Minimax

- Alpha $\alpha$: lower bound on the optimal score
- Beta $\beta$: upper bound on the optimal score
- No need to evaluate any branch whose value is guaranteed to be outside of $(\alpha, \beta)$ range

![Decision Tree Diagram]

- Maxim Likhachev
- Carnegie Mellon University
Alpha-Beta Pruning for Minimax

- Alpha $\alpha$: lower bound on the optimal score
- Beta $\beta$: upper bound on the optimal score
- No need to evaluate any branch whose value is guaranteed to be outside of ($\alpha$, $\beta$) range

5
\(\begin{align*} &\text{max} \\
&\text{min} \\
&\text{max} \\
&\text{min} \\
&\text{max} \\
&\text{min} \\
&\text{max} \\
&\text{min} \\
&\text{max} \\
&\text{min} \\
&\text{max} \\
&\text{min} \\
&\text{max} \\
&\text{min} \end{align*} \)

(-\infty; \infty)

(-\infty; \infty)

(-\infty; \infty)

(\infty; \infty)

(\infty; \infty)

(\infty; \infty)

(\infty; \infty)

(\infty; \infty)

(\infty; \infty)

(\infty; \infty)

(\infty; \infty)

(\infty; \infty)

(\infty; \infty)
Alpha-Beta Pruning for Minimax

• Alpha α: lower bound on the optimal score
• Beta β: upper bound on the optimal score
• No need to evaluate any branch whose value is guaranteed to be outside of (α, β) range
Alpha-Beta Pruning for Minimax

• Alpha $\alpha$: lower bound on the optimal score
• Beta $\beta$: upper bound on the optimal score
• No need to evaluate any branch whose value is guaranteed to be outside of ($\alpha$, $\beta$) range

\begin{align*}
\alpha &\leq \text{lower bound on the optimal score} \\
\beta &\leq \text{upper bound on the optimal score} \\
\text{No need to evaluate any branch whose value is guaranteed to be outside of } (\alpha, \beta) &\text{ range}
\end{align*}
Alpha-Beta Pruning for Minimax

- Alpha $\alpha$: lower bound on the optimal score
- Beta $\beta$: upper bound on the optimal score
- No need to evaluate any branch whose value is guaranteed to be outside of $(\alpha, \beta)$ range

Tree diagram with values and bounds:

- Max $5$ with range $(-\infty; \infty)$
- Min $5$ with range $(-\infty; 5)$
- Max $9$ with range $8; 5$
- Min $4$ with range $5; \infty$
- Max $10$ with range $5; \infty$
Alpha-Beta Pruning for Minimax

• Alpha $\alpha$: lower bound on the optimal score
• Beta $\beta$: upper bound on the optimal score
• No need to evaluate any branch whose value is guaranteed to be outside of $(\alpha, \beta)$ range

GetMinimaxValueWithAB(state $s$ at level $d$, $a$, $b$)

  if $d = \text{maximum depth}$, then return evaluation score of $s$
  if state $s$ is a leaf state, then return the score of $s$
  iterate over all successors $s'$ in succ($s$)
    if $d$ is even, then
      $a = \text{MAX}(a, \text{GetMinimaxValue} (s' \text{ at level } d+1, a, b))$
      if $a \geq b$ return $a$ //beta cut-off
    if $d$ is odd, then
      $b = \text{MIN}(b, \text{GetMinimaxValue} (s' \text{ at level } d+1, a, b))$
      if $a \geq b$ return $b$ //alpha cut-off
Alpha-Beta Pruning for Minimax

- Alpha $\alpha$: lower bound on the optimal score
- Beta $\beta$: upper bound on the optimal score
- No need to evaluate any branch whose value is guaranteed to be outside of ($\alpha$, $\beta$) range

GetMinimaxValueWithAB(state $s$ at level $d$, $a$, $b$)

- if $d = \text{maximum depth}$, then return evaluation score of $s$
- if state $s$ is a leaf state, then return the score of $s$
- iterate over all successors $s'$ in succ($s$)
  - if $d$ is even, then
    - $a = \text{MAX}(a, \text{GetMinimaxValue} (s' \text{ at level } d+1, a, b))$
    - if $a \geq b$ return $a$ //beta cut-off
  - if $d$ is odd, then
    - $b = \text{MIN}(b, \text{GetMinimaxValue} (s' \text{ at level } d+1, a, b))$
    - if $a \geq b$ return $b$ //alpha cut-off

What are the initial parameter values?
Alpha-Beta Pruning for Minimax

• Alpha $\alpha$: lower bound on the optimal score
• Beta $\beta$: upper bound on the optimal score
• No need to evaluate any branch whose value is guaranteed to be outside of $(\alpha, \beta)$ range

GetMinimaxValueWithAB(state s at level d, a, b)
  if $d =$ maximum depth, then return evaluation score of $s$
  if state $s$ is a leaf state, then return the score of $s$
  iterate over all successors $s'$ in succ($s$)
    if $d$ is even, then
      $a = \text{MAX}(a, \text{GetMinimaxValue} (s' \text{ at level } d+1, a, b))$
      if $a \geq b$ return $a$  //beta cut-off
    if $d$ is odd, then
      $b = \text{MIN}(b, \text{GetMinimaxValue} (s' \text{ at level } d+1, a, b))$
      if $a \geq b$ return $b$  //alpha cut-off
Alpha-Beta Pruning for Negamax

• Alpha $\alpha$: lower bound on the optimal score
• Beta $\beta$: upper bound on the optimal score

GetNegaMaxValue(state $s$ at level $d$, $a$, $b$)

  if $d = \text{maximum depth}$, then return $(-1)^d \times \text{evaluation score of } s$
  if state $s$ is a leaf state, then return $(-1)^d \times \text{score of } s$
  iterate over all successors $s'$ in $\text{succ}(s)$
  
  $a = \text{MAX}(a, -1 \times \text{GetNegaMaxValue} \ (s' \text{ at level } d+1, -b, -a))$

  if $a \geq b$ return $a$ //alpha-beta cut-off

from “Artificial Intelligence for Games” by I. Millington & J. Funge
Alpha-Beta Pruning for Negamax

- Alpha α: lower bound on the optimal score
- Beta β: upper bound on the optimal score

GetNegaMaxValue(state s at level d, a, b)

if d = maximum depth, then return \((-1)^d \times \text{evaluation score of } s\)

if state s is a leaf state, then return \((-1)^d \times \text{score of } s\)

iterate over all successors s’ in s:

\[ a = \text{MAX}(a, -1 \times \text{GetNegaMaxValue}(s’ at level d+1, -b, -a)) \]

if \(a \geq b\) return \(a\) // alpha-beta cut-off

What is the main computational bottleneck in both minimax & negamax? What are the ways to speed it up?

from “Artificial Intelligence for Games” by I. Millington & J. Funge
Speeding Up

• Picking “right” ordering for Alpha-Beta Minimax/Negamax

• Artificially narrowing the alpha-beta window (aspiration search)

• Use full-window for the first move, and then windows whose width varies as a result of the first move scores (negascout)
• Picking “right” ordering for Alpha-Beta Minimax/Negamax

• Artificially narrowing the alpha-beta window (aspiration search)

• Use full-window for the first move, and then windows whose width varies as a result of the first move scores (negascout)

Apparently, a combination is extremely popular in board games such as Chess, Checkers, etc.
Games with Transpositions

• If branches can merge (can arrive to the same state via multiple paths), then the game has transpositions - no longer a tree

Can we use the same Minimax/Negamax/Alpha-Beta searches?
Games with Transpositions

• If branches can merge (can arrive to the same state via multiple paths), then the game has transpositions - no longer a tree

Can we use the same Minimax/Negamax/Alpha-Beta searches?

Yes

How can we improve their efficiency?
Games with Transpositions

- If branches can merge (can arrive to the same state via multiple paths), then the game has transpositions - no longer a tree

Can we use the same Minimax/Negamax/Alpha-Beta searches?  
Yes

How can we improve their efficiency?  
Use transposition tables to avoid duplicate work
Games with Transpositions

• Transposition tables:
  - *maintain values for computed board configurations in memory and look them up/insert new ones during search*
  - let us avoid evaluating the same board configurations multiple times
  - let us avoid computing the minimax value of the sub-tree for the same board configurations multiple times
Games with Transpositions

• Transposition tables:
  - maintain values for computed board configurations in memory and look them up/insert new ones during search
  - let us avoid evaluating the same board configurations multiple times
  - let us avoid computing the minimax value of the sub-tree for the same board configurations multiple times

How do you store them in memory given the HUGE # of all possible configurations?
Games with Transpositions

• Transposition tables:
  - *maintain values for computed board configurations in memory and look them up/insert new ones during search*
  - let us avoid evaluating the same board configurations multiple times
  - let us avoid computing the minimax value of the sub-tree for the same board configurations multiple times

*How do you store them in memory given the HUGE # of all possible configurations?*

*Hash table*
Games with Transpositions

- Transposition tables:
  - maintain values for computed board configurations in memory and look them up/insert new ones during search
  - let us avoid evaluating the same board configurations multiple times
  - let us avoid computing the minimax value of the sub-tree for the same board configurations multiple times

How do you store them in memory given the HUGE # of all possible configurations?

Hash table

How do you keep the hash tables from growing too large?
Games with Transpositions

- **Transposition tables:**
  - maintain values for computed board configurations in memory and look them up/insert new ones during search
  - let us avoid evaluating the same board configurations multiple times
  - let us avoid computing the minimax value of the sub-tree for the same board configurations multiple times

How do you store them in memory given the HUGE # of all possible configurations?

How do you keep the hash tables from growing too large?

You can also store the transposition table from previous searches and work on improving it while your opponent thinks (typical trick used in commercial board games)!