CS 4700: Foundations of Artificial Intelligence

Spring 2020
Prof. Haym Hirsh

Lecture 9
February 12, 2020
Reminder: Technology Policy

No technology except for first four rows of left and right sides
Jupyter Notebooks Postponed

“Primer”: Friday 5pm in Gates G01
Friday Feb 14 5pm Gates G01
Covers:
Jupyter Notebooks and .ipynb files
Google Colab (online Jupyter Notebook environment)
Useful Python features

Bring your laptop
This Friday’s Lecture **Preponed**

To be given
Today 3:30-4:20
Gates G01

Will be recorded
and available on course website
(must watch between Wed and following Mon)

(Please come to the live version!)
Position-play Value

Each white piece has a certain position-play contribution and so does the black king. These must all be added up to give the position-play value.

For a Q, R, B, or Kt, count—
(a) The square root of the number of moves the piece can move from the position, counting a capture as two moves, and noting getting that the king must not be left in check.
(b) (If not a Q) 1·0 if it is defended, and an additional 0·5 if two times defended.
For a K, count—
(c) For moves other than castling as (a) above.
(d) It is then necessary to make some allowance for the vulnerability of the K. This can be done by assuming it to be replaced by a friendly Q on the same square, estimating as in (a), but subtracting instead of adding.
(e) Count 1·0 for the possibility of castling later not being lost by moves of K or rooks, a further 1·0 if castling could take place on the next move, and yet another 1·0 for the actual performance of castling.
For a P, count—
(f) 0·2 for each rank advanced.
(g) 0·3 for being defended by at least one piece (not P).
For the black K, count—
(h) 1·0 for the threat of checkmate.
(i) 0·5 for check.

We can now state the rule for play as follows. The move chosen must have the greatest possible value, and, consistent with this, the greatest possible position-play value. If this condition admits of
1. INTRODUCTION

This paper is concerned with the problem of constructing a computing routine or "program" for a modern general purpose computer which will enable it to play chess. Although perhaps of no practical importance, the question is of theoretical interest, and it is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance. Some possibilities in this direction are:

(1) Machines for designing filters, equalizers, etc.
(2) Machines for designing relay and switching circuits.
XXII. Programming a Computer for Playing Chess\textsuperscript{1}

By CLAUDE E. SHANNON

\[
f(P) = 200(K-K') + 9(Q-Q') + 5(R-R') + 3(B-B'+N-N') + (P-P') - 0.5(D-D'+S-S'+I-I') + 0.1(M-M') + ... \]

This paper is concerned with the problem of constructing a computing routine or "program" for a modern general purpose computer which will enable it to play chess.

Although perhaps of no practical importance, the question is of theoretical interest, and it is hoped that a satisfactory solution of this problem will act as a wedge in attacking other problems of a similar nature and of greater significance. Some possibilities in this direction are:

(1) Machines for designing filters, equalizers, etc.
(2) Machines for designing relay and switching circuits.
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Alpha-Beta Pruning: Implementation

My Turn
\( f(s) = \text{Max of successors} \)

My Opponent’s Turn
\( f(s) = \text{Min of successors} \)
Minimax Algorithm with Alpha-Beta Pruning

maxturn(s,ops,a,b): {my turn}

if cutoff(s,depth) then return f(s)
else
  val ← -∞;
  foreach o ∈ ops
    val’ ← minturn(apply(s,o),ops,a,b);
    if val’ > val then
      val ← val’;
      bestop ← o;
  if val ≥ b then return val;
  a ← max(a,val)
return val

minturn(s,ops,a,b): {opponent’s turn}

if cutoff(s,depth) then return f(s)
else
  val ← +∞;
  foreach o ∈ ops
    val’ ← maxturn(apply(s,o),ops,a,b);
    if val’ < val then
      val ← val’;
      bestop ← o;
  if val ≤ a then return val;
  b ← min(b,val)
return val

Initial call:
• If I go first: maxturn(initial-state,ops,-∞,+∞)
• If opponent goes first: minturn(initial-state,ops,-∞,+∞)
Alpha-Beta Pruning: Implementation

\[(\alpha, \beta)\]
\[(-\infty, +\infty)\]

My Turn
\[f(s) = \text{Max of successors}\]

My Opponent’s Turn
\[f(s) = \text{Min of successors}\]
Alpha-Beta Pruning: Implementation

(best I can force thus far, worst my opponent can force thus far)

($\alpha$, $\beta$)

($-\infty$, $+\infty$)

My Turn
$f(s) = \text{Max of successors}$

My Opponent’s Turn
$f(s) = \text{Min of successors}$
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Minimax Algorithm with Alpha-Beta Pruning

maxturn(s,ops,a,b):  {my turn}

if cutoff(s,depth) then return f(s)
else
    val ← -∞;
    foreach o ∈ ops
        val’ ← minturn(apply(s,o),ops,a,b);
        if val’ > val then
            val ← val’;
            bestop ← o;
    if val ≥ b then return val;
    a ← max(a,val)
return val

minturn(s,ops,a,b):  {opponent’s turn}

if cutoff(s,depth) then return f(s)
else
    val ← +∞;
    foreach o ∈ ops
        val’ ← maxturn(apply(s,o),ops,a,b);
        if val’ < val then
            val ← val’;
            bestop ← o;
    if val ≤ a then return val;
    b ← min(b,val)
return val

Initial call:
• If I go first: maxturn(initial-state,ops,-∞,+∞)
• If opponent goes first: minturn(initial-state,ops,-∞,+∞)
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Minimax Algorithm with Alpha-Beta Pruning

\[
\text{maxturn}(s, \text{ops}, a, b): \quad \{\text{my turn}\}
\]

if cutoff\((s, \text{depth})\) then return \(f(s)\)
else
  \[\text{val} \leftarrow -\infty;\]
  foreach \(o \in \text{ops}\)
    \[\text{val}' \leftarrow \text{minturn}(\text{apply}(s, o), \text{ops}, a, b);\]
    if \(\text{val}' > \text{val}\) then
      \[\text{val} \leftarrow \text{val}';\]
      \[\text{bestop} \leftarrow o;\]
    if \(\text{val} \geq b\) then return \(\text{val}\);
  \[a \leftarrow \max(a, \text{val})\]
return \(\text{val}\)

\[
\text{minturn}(s, \text{ops}, a, b): \quad \{\text{opponent’s turn}\}
\]

if cutoff\((s, \text{depth})\) then return \(f(s)\)
else
  \[\text{val} \leftarrow +\infty;\]
  foreach \(o \in \text{ops}\)
    \[\text{val}' \leftarrow \text{maxturn}(\text{apply}(s, o), \text{ops}, a, b);\]
    if \(\text{val}' < \text{val}\) then
      \[\text{val} \leftarrow \text{val}';\]
      \[\text{bestop} \leftarrow o;\]
    if \(\text{val} \leq a\) then return \(\text{val}\);
  \[b \leftarrow \min(b, \text{val})\]
return \(\text{val}\)

Initial call:
- If I go first: \(\text{maxturn}(\text{initial-state}, \text{ops}, -\infty, +\infty)\)
- If opponent goes first: \(\text{minturn}(\text{initial-state}, \text{ops}, -\infty, +\infty)\)
Alpha-Beta Pruning: Implementation

My Turn

\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn

\[ f(s) = \text{Min of successors} \]
Minimax Algorithm with Alpha-Beta Pruning

\[\text{maxturn}(s,\text{ops},a,b): \quad \text{\{my turn\}}\]

if \(\text{cutoff}(s,\text{depth})\) then return \(f(s)\)
else
    \(\text{val} \leftarrow -\infty;\)
    \(\text{foreach } o \in \text{ops}\)
    \(\text{val}' \leftarrow \text{minturn}(\text{apply}(s,o)\),\text{ops},a,b);\)
    if \(\text{val}' > \text{val}\) then
        \(\text{val} \leftarrow \text{val}';\)
        \(\text{bestop} \leftarrow o;\)
    if \(\text{val} \geq b\) then return \(\text{val}\);
    \(a \leftarrow \max(a,\text{val})\)
return \(\text{val}\)

\[\text{minturn}(s,\text{ops},a,b): \quad \text{\{opponent’s turn\}}\]

if \(\text{cutoff}(s,\text{depth})\) then return \(f(s)\)
else
    \(\text{val} \leftarrow +\infty;\)
    \(\text{foreach } o \in \text{ops}\)
    \(\text{val}' \leftarrow \text{maxturn}(\text{apply}(s,o)\),\text{ops},a,b);\)
    if \(\text{val}' < \text{val}\) then
        \(\text{val} \leftarrow \text{val}';\)
        \(\text{bestop} \leftarrow o;\)
    if \(\text{val} \leq a\) then return \(\text{val}\);
    \(b \leftarrow \min(b,\text{val})\)
return \(\text{val}\)

Initial call:
• If I go first: \(\text{maxturn}(\text{initial-state},\text{ops},-\infty,+,\infty)\)
• If opponent goes first: \(\text{minturn}(\text{initial-state},\text{ops},-\infty,+,\infty)\)
Alpha-Beta Pruning: Implementation

My Turn
\( f(s) = \text{Max of successors} \)

My Opponent’s Turn
\( f(s) = \text{Min of successors} \)
Minimax Algorithm
with Alpha-Beta Pruning

maxturn(s,ops,a,b): {my turn}
    if cutoff(s,depth) then return f(s)
    else
        val ← -∞;
        foreach o ∈ ops
            val’ ← minturn(apply(s,o),ops,a,b);
            if val’ > val then
                val ← val’;
                bestop ← o;
        if val ≥ b then return val;
        a ← max(a,val)
    return val

minturn(s,ops,a,b): {opponent’s turn}
    if cutoff(s,depth) then return f(s)
    else
        val ← +∞;
        foreach o ∈ ops
            val’ ← maxturn(apply(s,o),ops,a,b);
            if val’ < val then
                val ← val’;
                bestop ← o;
        if val ≤ a then return val;
        b ← min(b,val)
    return val

Initial call:
• If I go first: maxturn(initial-state,ops,-∞,+∞)
• If opponent goes first: minturn(initial-state,ops,-∞,+∞)
Alpha-Beta Pruning: Implementation

My Turn
\( f(s) = \text{Max of successors} \)

My Opponent’s Turn
\( f(s) = \text{Min of successors} \)
Alpha-Beta Pruning: Implementation

My Turn
\( f(s) = \text{Max of successors} \)

My Opponent’s Turn
\( f(s) = \text{Min of successors} \)
Alpha-Beta Pruning: Implementation

My Turn
f(s) = Max of successors

My Opponent’s Turn
f(s) = Min of successors

Diagram:

- 

My Turn
f(s) = Max of successors

My Opponent’s Turn
f(s) = Min of successors
Minimax Algorithm with Alpha-Beta Pruning

\[
\text{maxturn}(s, \text{ops}, a, b): \quad \{\text{my turn}\}
\]

\[
\text{if cutoff}(s, \text{depth}) \\text{then return } f(s)
\]

\[
\text{else}
\]

\[
\text{val} \leftarrow -\infty;
\]

\[
\text{foreach } o \in \text{ops}
\]

\[
\text{val}' \leftarrow \text{minturn}(\text{apply}(s, o), \text{ops}, a, b);
\]

\[
\text{if val}' > \text{val} \text{ then}
\]

\[
\text{val} \leftarrow \text{val}';
\]

\[
\text{bestop} \leftarrow o;
\]

\[
\text{if val} \geq b \text{ then return val} ;
\]

\[
a \leftarrow \max(a, \text{val})
\]

\[
\text{return val}
\]

\[
\text{minturn}(s, \text{ops}, a, b): \quad \{\text{opponent’s turn}\}
\]

\[
\text{if cutoff}(s, \text{depth}) \\text{then return } f(s)
\]

\[
\text{else}
\]

\[
\text{val} \leftarrow +\infty;
\]

\[
\text{foreach } o \in \text{ops}
\]

\[
\text{val}' \leftarrow \text{maxturn}(\text{apply}(s, o), \text{ops}, a, b);
\]

\[
\text{if val}' < \text{val} \text{ then}
\]

\[
\text{val} \leftarrow \text{val}';
\]

\[
\text{bestop} \leftarrow o;
\]

\[
\text{if val} \leq a \text{ then return val} ;
\]

\[
b \leftarrow \min(b, \text{val})
\]

\[
\text{return val}
\]

Initial call:

- If I go first: \( \text{maxturn}(\text{initial-state}, \text{ops}, -\infty, +\infty) \)
- If opponent goes first: \( \text{minturn}(\text{initial-state}, \text{ops}, -\infty, +\infty) \)
Alpha-Beta Pruning: Implementation

My Turn
f(s) = Max of successors

My Opponent’s Turn
f(s) = Min of successors

(-∞, +∞)

(-∞, -5)

-5
+4

(-∞, -5)
Minimax Algorithm with Alpha-Beta Pruning

Maxturn($s, ops, a, b$): \{my turn\}

- if cutoff($s, depth$) then return f($s$)
- else
  - val $\leftarrow -\infty$
  - foreach $o \in ops$
    - val’ $\leftarrow$ minturn(apply($s$, $o$), $ops, a, b$);
    - if val’ > val then
      - val $\leftarrow$ val’;
      - bestop $\leftarrow o$;
    - if val $\geq b$ then return val;
  - a $\leftarrow$ max(a, val)
- return val

Minturn($s, ops, a, b$): \{opponent’s turn\}

- if cutoff($s, depth$) then return f($s$)
- else
  - val $\leftarrow +\infty$
  - foreach $o \in ops$
    - val’ $\leftarrow$ maxturn(apply($s$, $o$), $ops, a, b$);
    - if val’ < val then
      - val $\leftarrow$ val’;
      - bestop $\leftarrow o$;
    - if val $\leq a$ then return val;
  - b $\leftarrow$ min(b, val)
- return val

Initial call:
- If I go first: maxturn(initial-state, ops, -\infty, +\infty)
- If opponent goes first: minturn(initial-state, ops, -\infty, +\infty)
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Minimax Algorithm with Alpha-Beta Pruning

maxturn(s, ops, a, b): \{my turn\}

if cutoff(s, depth) then return f(s)
else
  val ← -∞;
  foreach o ∈ ops
    val’ ← minturn(apply(s, o), ops, a, b);
    if val’ > val then
      val ← val’;
      bestop ← o;
  if val ≥ b then return val;
  a ← max(a, val)
return val

minturn(s, ops, a, b): \{opponent’s turn\}

if cutoff(s, depth) then return f(s)
else
  val ← +∞;
  foreach o ∈ ops
    val’ ← maxturn(apply(s, o), ops, a, b);
    if val’ < val then
      val ← val’;
      bestop ← o;
  if val ≤ a then return val;
  b ← min(b, val)
return val

Initial call:
• If I go first: maxturn(initial-state, ops, -∞, +∞)
• If opponent goes first: minturn(initial-state, ops, -∞, +∞)
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Minimax Algorithm with Alpha-Beta Pruning

\[
\text{maxturn}(s, \text{ops}, a, b): \quad \{\text{my turn}\}
\]

\[
\begin{align*}
\text{if } \text{cutoff}(s, \text{depth}) \text{ then return } & f(s) \\
\text{else} & \\
\text{val} & \leftarrow -\infty; \\
\text{foreach } o \in \text{ops} & \\
\text{val}' & \leftarrow \text{minturn}(\text{apply}(s, o), \text{ops}, a, b); \\
\text{if } \text{val}' > \text{val} & \text{ then} \\
\text{val} & \leftarrow \text{val}'; \\
\text{bestop} & \leftarrow o; \\
\text{if } \text{val} \geq b & \text{ then return } \text{val}; \\
\text{a} & \leftarrow \max(a, \text{val}) \\
\text{return val}
\end{align*}
\]

\[
\text{minturn}(s, \text{ops}, a, b): \quad \{\text{opponent’s turn}\}
\]

\[
\begin{align*}
\text{if } \text{cutoff}(s, \text{depth}) \text{ then return } & f(s) \\
\text{else} & \\
\text{val} & \leftarrow +\infty; \\
\text{foreach } o \in \text{ops} & \\
\text{val}' & \leftarrow \text{maxturn}(\text{apply}(s, o), \text{ops}, a, b); \\
\text{if } \text{val}' < \text{val} & \text{ then} \\
\text{val} & \leftarrow \text{val}'; \\
\text{bestop} & \leftarrow o; \\
\text{if } \text{val} \leq a & \text{ then return } \text{val}; \\
\text{b} & \leftarrow \min(b, \text{val}) \\
\text{return val}
\end{align*}
\]

Initial call:
- If I go first: \( \text{maxturn} (\text{initial-state}, \text{ops}, -\infty, +\infty) \)
- If opponent goes first: \( \text{minturn} (\text{initial-state}, \text{ops}, -\infty, +\infty) \)
Alpha-Beta Pruning: Implementation

\[ f(s) = \begin{cases} \text{Max of successors} & \text{My Turn} \\ \text{Min of successors} & \text{My Opponent's Turn} \end{cases} \]
# Minimax Algorithm with Alpha-Beta Pruning

\[
\text{maxturn}(s, \text{ops, } a, b): \quad \{\text{my turn}\}
\]

- If cutoff\((s, \text{depth})\) then return \(f(s)\)
- Else
  - \(val \leftarrow -\infty\);
  - foreach \(o \in \text{ops}\)
    - \(val' \leftarrow \text{minturn}(\text{apply}(s, o), \text{ops, } a, b)\);
    - if \(val' > val\) then
      - \(val \leftarrow val'\);
      - \(\text{bestop} \leftarrow o\);
    - if \(val \geq b\) then return \(val\);
  - \(a \leftarrow \max(a, val)\)

return \(val\)

\[
\text{minturn}(s, \text{ops, } a, b): \quad \{\text{opponent’s turn}\}
\]

- If cutoff\((s, \text{depth})\) then return \(f(s)\)
- Else
  - \(val \leftarrow +\infty\);
  - foreach \(o \in \text{ops}\)
    - \(val' \leftarrow \text{maxturn}(\text{apply}(s, o), \text{ops, } a, b)\);
    - if \(val' < val\) then
      - \(val \leftarrow val'\);
      - \(\text{bestop} \leftarrow o\);
    - if \(val \leq a\) then return \(val\);
  - \(b \leftarrow \min(b, val)\)

return \(val\)

Initial call:
- If I go first: \(\text{maxturn}(\text{initial-state, ops, } -\infty, +\infty)\)
- If opponent goes first: \(\text{minturn}(\text{initial-state, ops, } -\infty, +\infty)\)
Alpha-Beta Pruning: Implementation

My Turn
\( f(s) = \text{Max of successors} \)

My Opponent’s Turn
\( f(s) = \text{Min of successors} \)
Minimax Algorithm
with Alpha-Beta Pruning

maxturn(s,ops,a,b):
  {my turn}
  if cutoff(s,depth) then return f(s)
  else
    val ← -∞;
    foreach o ∈ ops
      val’ ← minturn(apply(s,o),ops,a,b);
      if val’ > val then
        val ← val’;
        bestop ← o;
    if val ≥ b then return val;
    a ← max(a,val)
  return val

minturn(s,ops,a,b):
  {opponent's turn}
  if cutoff(s,depth) then return f(s)
  else
    val ← +∞;
    foreach o ∈ ops
      val’ ← maxturn(apply(s,o),ops,a,b);
      if val’ < val then
        val ← val’;
        bestop ← o;
    if val ≤ a then return val;
    b ← min(b,val)
  return val

Initial call:
• If I go first: maxturn(initial-state,ops,-∞,+∞)
• If opponent goes first: minturn(initial-state,ops,-∞,+∞)
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Minimax Algorithm with Alpha-Beta Pruning

\textbf{maxturn}(s,ops,a,b): \quad \{\text{my turn}\}

\begin{align*}
\text{if cutoff}(s,\text{depth}) & \text{ then return } f(s) \\
\text{else} & \\
& \text{val } \leftarrow -\infty; \\
& \text{foreach } o \in \text{ops} \\
& \quad \text{val' } \leftarrow \text{minturn(apply}(s,o),\text{ops,a,b}); \\
& \quad \text{if val' } > \text{ val then} \\
& \quad \quad \text{val } \leftarrow \text{val'}; \\
& \quad \quad \text{bestop } \leftarrow o; \\
& \quad \text{if val } \geq b \text{ then return val}; \\
& a \leftarrow \max(a,\text{val}) \\
& \text{return val}
\end{align*}

\textbf{minturn}(s,ops,a,b): \quad \{\text{opponent’s turn}\}

\begin{align*}
\text{if cutoff}(s,\text{depth}) & \text{ then return } f(s) \\
\text{else} & \\
& \text{val } \leftarrow +\infty; \\
& \text{foreach } o \in \text{ops} \\
& \quad \text{val' } \leftarrow \text{maxturn(apply}(s,o),\text{ops,a,b}); \\
& \quad \text{if val' } < \text{ val then} \\
& \quad \quad \text{val } \leftarrow \text{val'}; \\
& \quad \quad \text{bestop } \leftarrow o; \\
& \quad \text{if val } \leq a \text{ then return val}; \\
& b \leftarrow \min(b,\text{val}) \\
& \text{return val}
\end{align*}

Initial call:
- If I go first: \text{maxturn}(\text{initial-state},\text{ops},-\infty,+:\infty)
- If opponent goes first: \text{minturn}(\text{initial-state},\text{ops},-\infty,+:\infty)
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Alpha-Beta Pruning: Implementation

My Turn
\( f(s) = \text{Max of successors} \)

My Opponent’s Turn
\( f(s) = \text{Min of successors} \)
Minimax Algorithm with Alpha-Beta Pruning

maxturn(s,ops,a,b): {my turn}
  if cutoff(s,depth) then return f(s)
  else
    val ← -∞;
    foreach o ∈ ops
      val’ ← minturn(apply(s,o),ops,a,b);
      if val’ > val then
        val ← val’;
        bestop ← o;
      if val ≥ b then return val;
    a ← max(a,val)
  return val

minturn(s,ops,a,b): {opponent’s turn}
  if cutoff(s,depth) then return f(s)
  else
    val ← +∞;
    foreach o ∈ ops
      val’ ← maxturn(apply(s,o),ops,a,b);
      if val’ < val then
        val ← val’;
        bestop ← o;
      if val ≤ a then return val;
    b ← min(b,val)
  return val

Initial call:
• If I go first: maxturn(initial-state,ops,-∞,+∞)
• If opponent goes first: minturn(initial-state,ops,-∞,+∞)
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Minimax Algorithm with Alpha-Beta Pruning

maxturn(s,ops,a,b): {my turn}  
if cutoff(s,depth) then return f(s)  
else  
val ← -∞;  
foreach o ∈ ops  
val’ ← minturn(apply(s,o),ops,a,b);  
if val’ > val then  
val ← val’;  
bestop ← o;  
if val ≥ b then return val;  
a ← max(a,val)  
return val

minturn(s,ops,a,b): {opponent’s turn}  
if cutoff(s,depth) then return f(s)  
else  
val ← +∞;  
foreach o ∈ ops  
val’ ← maxturn(apply(s,o),ops,a,b);  
if val’ < val then  
val ← val’;  
bestop ← o;  
if val ≤ a then return val;  
b ← min(b,val)  
return val

Initial call:  
• If I go first: maxturn(initial-state,ops,-∞,+∞)  
• If opponent goes first: minturn(initial-state,ops,-∞,+∞)
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Alpha-Beta Pruning: Implementation

My Turn
\[ f(s) = \text{Max of successors} \]

My Opponent’s Turn
\[ f(s) = \text{Min of successors} \]
Alpha-Beta Pruning: Implementation

And so on
Improving Minimax Search

For vanilla two-player zero-sum games:
Improving Minimax Search

For vanilla two-player zero-sum games:

- Ordering (or pruning) the operators to try at each node:
Improving Minimax Search

For vanilla two-player zero-sum games:

• Ordering (or pruning) the operators to try at each node:
  • Use $f(s)$
  • Learn from game play (can be simple statistics on move outcomes)
Improving Minimax Search

For vanilla two-player zero-sum games:

• Ordering (or pruning) the operators to try at each node:
  • Use \( f(s) \)
  • Learn from game play (can be simple statistics on move outcomes)

• Dealing with the “horizon effect” – the effect of a move is past depth-bound “horizon”:
Improving Minimax Search

For vanilla two-player zero-sum games:

- Ordering (or pruning) the operators to try at each node:
  - Use $f(s)$
  - Learn from game play (can be simple statistics on move outcomes)

- Dealing with the “horizon effect” – the effect of a move is past depth-bound “horizon”:
  - Don’t use fixed depth, go deeper when appropriate (for example, “quiescence”)
Improving Minimax Search

For vanilla two-player zero-sum games:

• Ordering (or pruning) the operators to try at each node:
  • Use $f(s)$
  • Learn from game play (can be simple statistics on move outcomes)

• Dealing with the “horizon effect” – the effect of a move is past depth-bound “horizon”:
  • Don’t use fixed depth, go deeper when appropriate (for example, “quiescence”)

• Memorize “book openings”
Improving Minimax Search

For vanilla two-player zero-sum games:

• Ordering (or pruning) the operators to try at each node:
  • Use $f(s)$
  • Learn from game play (can be simple statistics on move outcomes)

• Dealing with the “horizon effect” – the effect of a move is past depth-bound “horizon”:
  • Don’t use fixed depth, go deeper when appropriate (for example, “quiescence”)

• Memorize “book openings”

• Table lookup for endgames: can precompute a lot
Improving Minimax Search

For vanilla two-player zero-sum games:

• Ordering (or pruning) the operators to try at each node:
  • Use f(s)
  • Learn from game play (can be simple statistics on move outcomes)

• Dealing with the “horizon effect” – the effect of a move is past depth-bound “horizon”:
  • Don’t use fixed depth, go deeper when appropriate (for example, “quiescence”)

• Memorize “book openings”

• Table lookup for endgames: can precompute a lot

• ...
Improving Minimax Search

Going beyond vanilla two-player zero-sum games:
Improving Minimax Search

Going beyond vanilla two-player zero-sum games:

• Random game elements (stochastic games) – example: dice
Improving Minimax Search

Going beyond vanilla two-player zero-sum games:

• Random game elements (stochastic games) – example: dice
  • Repeated simulation
• Learning evaluation functions from book games and self-play
Improving Minimax Search

Going beyond vanilla two-player zero-sum games:

• Random game elements (stochastic games) – example: dice
  • Repeated simulation

• Learning evaluation functions from book games and self-play
  • Example: learn \( w_i \)'s if function looks like

\[
f(s) = w_1f_1(s) + w_2f_2(s) + \cdots + w_nf_n(s) = \sum_{i=1}^{n} w_if_i(s)
\]
Some Studies in Machine Learning
Using the Game of Checkers

Arthur L. Samuel

Abstract: Two machine-learning procedures have been investigated in some detail using the game of checkers. Enough work has been done to verify the fact that a computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program. Furthermore, it can learn to do this in a remarkably short period of time (8 or 10 hours of machine-playing time) when given only the rules of the game, a sense of direction, and a redundant and incomplete list of parameters which are thought to have something to do with the game, but whose correct signs and relative weights are unknown and unspecified. The principles of machine learning verified by these experiments are, of course, applicable to many other situations.

Introduction

The studies reported here have been concerned with the programming of a digital computer to behave in a way which, if done by human beings or animals, would be described as involving the process of learning. While this is not the place to dwell on the importance of machine-learning procedures, or to discourse on the philosophical aspects of the so-called problem of learning, present investigations have led to the development of a machine for playing a game of checkers. The first attempt to write such a program (1950) was made by Newell and Shaw. This program, which is described below, is very different from the first version of the game program. It involves a method that can be called self-learning. The machine plays itself, improving its play by learning. The program is a book game, since it was written in a book style, with the features of a book game such as self-play.

The second procedure requires some programming for each new application. This procedure is called the self-learning method. The self-learning method should lead to the development of general-purpose learning machines. A comparison between the size of the switching nets that can be reasonably constructed or simulated at the present time and the size of the neural nets used by animals, suggests that we have a long way to go before we obtain practical devices. The second procedure requires some programming for each new application.
A Parallel Network that Learns to Play Backgammon

G. Tesauro*
Center for Complex Systems Research, University of Illinois
at Urbana-Champaign, 508 S. Sixth St., Champaign,
IL 61820, U.S.A.

T.J. Sejnowski**
Biophysics Department, The Johns Hopkins University,
Baltimore, MD 21218, U.S.A.

ABSTRACT
A class of connectionist networks is described that has learned to play backgammon at an intermediate-to-advanced level. The networks were trained by back-propagation learning on a large set of sample positions evaluated by a human expert. In actual match play against humans and conventional computer programs, the networks have demonstrated substantial ability to generalize on the basis of expert knowledge of the game. This is possibly the most complex domain yet studied with connectionist learning. New techniques were needed to overcome problems due to the scale and complexity of the task. These include techniques for intelligent design of training set examples and efficient coding schemes, and procedures for escaping from local minima. We suggest how these techniques might be used in applications of network learning to general large-scale, difficult "real-world" problem domains.
Improving Minimax Search

Going beyond vanilla two-player zero-sum games:

• Random game elements (stochastic games) – example: dice
  • Repeated simulation

• Learning evaluation functions from book games and self-play
  • Example: learn $w_i$’s if function looks like
    \[ f(s) = w_1f_1(s) + w_2f_2(s) + \cdots + w_nf_n(s) = \sum_{i=1}^{n} w_if_i(s) \]

• Games of hidden information – example: other players’ hands
Improving Minimax Search

Going beyond vanilla two-player zero-sum games:

• Random game elements (stochastic games) – example: dice
  • Repeated simulation

• Learning evaluation functions from book games and self-play
  • Example: learn $w_i$’s if function looks like
    \[
    f(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)
    \]

• Games of hidden information – example: other players’ hands
  • Infer distribution over hidden information
  • Repeated simulation
Improving Minimax Search

Going beyond vanilla two-player zero-sum games:
• Random game elements (stochastic games) – example: dice
  • Repeated simulation
• Learning evaluation functions from book games and self-play
  • Example: learn $w_i$’s if function looks like
    \[ f(s) = w_1f_1(s) + w_2f_2(s) + \cdots + w_nf_n(s) = \sum_{i=1}^{n} w_if_i(s) \]
• Games of hidden information – example: other players’ hands
  • Infer distribution over hidden information
  • Repeated simulation

Much of this is captured in Monte Carlo Tree Search (coming later)
Next Topic:
Knowledge Representation and Reasoning
Textbook Chapter 7
Knowledge Representation and Reasoning

• Agents:
  • Represent what they know about the world
  • Use inference to derive new information

Sometimes called the “Logicist Approach” to AI
Wumpus World

Get points for taking gold
Die if in the same square as pit or wumpus
Can move Up, Down, Left, Right
Sensors:
• Stench: Wumpus is 1 away
• Breeze: Pit is 1 away
• Glitter: Gold is in current room
• States: [x, y, Stench?, Breeze?, Glitter?]
  • Initial state: [1,1,False,False,False]
Wumpus World

Definition of world (not known to agent)

What the agent knows

State = [1, 1, False, False, False]
Wumpus World

Definition of world (not known to agent)

OK means “safe” (won’t die)

What the agent knows
Wumpus World
Wumpus World
Wumpus World

- No stench => no Wumpus in adjacent squares
- No breeze => no pit in adjacent squares
Wumpus World

No glitter => no gold here
Wumpus World

Wumpus moves Right
Wumpus World

State = [2,1,False,True,False]
Wumpus World
Wumpus World

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>PIT</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>PIT</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>START</td>
<td></td>
<td>PIT</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,4</td>
<td>2,4</td>
<td>3,4</td>
<td>4,4</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>3,3</td>
<td>4,3</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
<td>P?</td>
<td>3,2</td>
</tr>
<tr>
<td>1,1</td>
<td>OK</td>
<td>2,1</td>
<td>B OK</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>3,1</td>
<td>P?</td>
</tr>
<tr>
<td></td>
<td>OK</td>
<td>4,1</td>
<td></td>
</tr>
</tbody>
</table>
Wumpus World
Wumpus World
Wumpus World
Knowledge Representation and Reasoning

• Agents:
  • Represent what they know about the world
  • Use inference to derive new information
Knowledge Representation and Reasoning

• Agents:
  • Represent what they know about the world
  • Use inference to derive new information

Focus here will be on formal logic