Alpha-Beta Pruning: Algorithm and Analysis

Tsan-sheng Hsu

http://www.iis.sinica.edu.tw/~tshsu
**Introduction**

- **Alpha-beta pruning** is the standard searching procedure used for 2-person perfect-information zero sum games.

- **Definitions:**
  - A *position* $p$.
  - The *value* of a position $p$, $f(p)$, is a numerical value computed from evaluating $p$.
    - Value is computed from the root player’s point of view.
    - Positive values mean in favor of the root player.
    - Negative values mean in favor of the opponent.
    - Since it is a zero sum game, thus from the opponent’s point of view, the value can be assigned $-f(p)$.
  - A *terminal position*: a position whose value can be known.
    - A position where win/loss/draw can be concluded.
    - A position where some constraints are met.
  - A *position* $p$ has $d$ legal moves $p_1, p_2, \ldots, p_d$. 
From the root, number a node in a search tree by a sequence of integers \(a.b.c.d\ldots\)
- Meaning from the root, you first take the \(a\)th branch, then the \(b\)th branch, and then the \(c\)th branch, and then the \(d\)th branch \ldots
- The root is specified as an empty sequence.
- The depth of a node is the length of the sequence of integers specifying it.

This is called “Dewey decimal system.”
Mini-max formulation:

- \( F'(p) = \begin{cases} 
  f(p) & \text{if } d = 0 \\
  \max \{G'(p_1), \ldots, G'(p_d)\} & \text{if } d > 0 
\end{cases} \)

- \( G'(p) = \begin{cases} 
  f(p) & \text{if } d = 0 \\
  \min \{F'(p_1), \ldots, F'(p_d)\} & \text{if } d > 0 
\end{cases} \)

- An indirect recursive formula!
- Equivalent to AND-OR logic.
Algorithm: Mini-max

- **Algorithm** $F'(\text{position } p)$ // max node
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$, then return $f(p)$ else begin
    - $m := -\infty$
    - for $i := 1$ to $d$ do
      - $t := G'(p_i)$
      - if $t > m$ then $m := t$ // find max value
  - end; return $m$

- **Algorithm** $G'(\text{position } p)$ // min node
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$, then return $f(p)$ else begin
    - $m := \infty$
    - for $i := 1$ to $d$ do
      - $t := F'(p_i)$
      - if $t < m$ then $m := t$ // find min value
  - end; return $m$

- A brute-force method to try all possibilities!
Algorithm $F'(\text{position } p)$ // max node

- determine the successor positions $p_1, \ldots, p_d$
- if $d = 0$ // a terminal node
  - or depth reaches the cutoff threshold // from iterative deepening
  - or time is running up // from timing control
  - or some other constraints are met // add knowledge here
  then return $f(p)$ // current board value
  else begin
    $\Delta m := -\infty$ // initial value
    for $i := 1$ to $d$ do // try each child
      begin
        $\Delta t := G'(p_i)$
        if $t > m$ then $m := t$ // find max value
      end
    end
  return $m$
Algorithm \( G'(\text{position } p) \) // min node

- determine the successor positions \( p_1, \ldots, p_d \)
- if \( d = 0 \) // a terminal node
  - or depth reaches the cutoff threshold // from iterative deepening
  - or time is running up // from timing control
  - or some other constraints are met // add knowledge here
  then return \( f(p) \) // current board value
else begin
  \[
  \begin{align*}
  \Delta m &:= \infty \quad // \text{initial value} \\
  \text{for } i &:= 1 \text{ to } d \text{ do } // \text{try each child} \\
  &\text{begin} \\
  &\quad t := F'(p_i) \\
  &\quad \text{if } t < m \text{ then } m := t // \text{find min value} \\
  &\text{end}
  \end{align*}
  \]
end
- return \( m \)
Nega-max formulation:

Let $F(p)$ be the greatest possible value achievable from position $p$ against the optimal defensive strategy.

\[
F(p) = \begin{cases} 
  h(p) & \text{if } d = 0 \\
  \max\{-F(p_1), \ldots, -F(p_d)\} & \text{if } d > 0 
\end{cases}
\]

\[
h(p) = \begin{cases} 
  f(p) & \text{if depth of } p \text{ is 0 or even} \\
  -f(p) & \text{if depth of } p \text{ is odd}
\end{cases}
\]
Algorithm: Nega-max

- Algorithm $F(position \ p)$
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$ // a terminal node
    - or depth reaches the cutoff threshold // from iterative deepening
    - or time is running up // from timing control
    - or some other constraints are met // add knowledge here
  - then return $h(p)$ else
  - begin
    - $m := -\infty$
    - for $i := 1$ to $d$ do
      - begin
        - $t := -F(p_i)$ // recursive call, the returned value is negated
        - if $t > m$ then $m := t$ // always find a max value
      - end
    - end
  - return $m$

- Also a brute-force method to try all possibilities, but with a simpler code.
Intuition for improvements

- **Branch-and-bound:** using information you have so far to cut or prune branches.
  - A branch is cut means we do not need to search it anymore.
  - If you know for sure the value of your result is more than $x$ and the current search result for this branch so far can give you no more than $x$,
    - then there is no need to search this branch any further.

- **Two types of approaches**
  - **Exact algorithms:** through mathematical proof, it is guaranteed that the branches pruned won’t contain the solution.
    - *Alpha-beta pruning: reinvented by several researchers in the 1950’s and 1960’s.*
    - *Scout.*
    - . . .
  - **Approximated heuristics:** with a high probability that the solution won’t be contained in the branches pruned.
    - *Obtain a good estimation on the remaining cost.*
    - *Cut a branch when it is in a very bad position and there is little hope to gain back the advantage.*
Alpha cut-off:
- On a max node
  - Assume you have finished exploring the branch at 1 and obtained the best value from it as bound.
  - You now search the branch at 2 by first searching the branch at 2.1.
  - Assume branch at 2.1 returns a value that is $\leq$ bound.
  - Then no need to evaluate the branch at 2.2 and all later branches of 2, if any, at all.
  - The best possible value for the branch at 2 must be $\leq$ bound.
  - Hence we should take value returned from the branch at 1 as the best possible solution.
**Beta cut-off:**

- **On a min node**
  - Assume you have finished exploring the branch at 1.1 and obtained the best value from it as bound.
  - You now search the branches at 1.2 by first exploring the branch at 1.2.1.
  - Assume the branch at 1.2.1 returns a value that is $\geq$ bound.
  - Then no need to evaluate the branch at 1.2.2 and all later branches of 1.2, if any, at all.
  - The best possible value for the branch at 1.2 is $\geq$ bound.
  - Hence we should take value returned from the branch at 1.1 as the best possible solution.
Deep alpha cut-off

- **For alpha cut-off:**
  - For a min node $u$, the branch of its ancestor (e.g., elder brother of its parent) produces a lower bound $V_l$.
  - The first branch of $u$ produces an upper bound $V_u$ for $v$.
  - If $V_l \geq V_u$, then there is no need to evaluate the second branch and all later branches, of $u$.

- **Deep alpha cut-off:**
  - Def: For a node $u$ in a tree and a positive integer $g$, $\text{Ancestor}(g, u)$ is the direct ancestor of $u$ by tracing the parent’s link $g$ times.
  - When the lower bound $V_l$ is produced at and propagated from $u$’s great grand parent, i.e., $\text{Ancestor}(3, u)$, or any $\text{Ancestor}(2i + 1, u)$, $i \geq 1$.
  - When an upper bound $V_u$ is returned from the a branch of $u$ and $V_l \geq V_u$, then there is no need to evaluate all later branches of $u$.

- We can find similar properties for deep beta cut-off.
Illustration — Deep alpha cut-off

1.1 1.2
V=15
2.1 2.2
\( V \geq 15 \)
2.1.1
2.1.1.1 2.1.1.2
V=7
V \leq 7
V\geq 15

TCG: \( \alpha-\beta \) Pruning, 20131106, Tsan-sheng Hsu ©
Ideas for refinements

- During searching, maintain two values $\alpha$ and $\beta$ so that
  - $\alpha$ is the current lower bound of the possible returned value;
  - $\beta$ is the current upper bound of the possible returned value.

- If during searching, we know for sure $\alpha > \beta$, then there is no need to search any more in this branch.
  - The returned value cannot be in this branch.
  - Backtrack until it is the case $\alpha \leq \beta$.

- The two values $\alpha$ and $\beta$ are called the ranges of the current search window.
  - These values are dynamic.
  - Initially, $\alpha$ is $-\infty$ and $\beta$ is $\infty$. 
Alpha-beta pruning algorithm: Mini-Max

- Algorithm $F_2'(\text{position } p, \text{ value } \alpha, \text{ value } \beta) \ // \ max \ node$
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$, then return $f(p)$ else begin
    - $m := \alpha$
    - for $i := 1$ to $d$ do
      - $t := G_2'(p_i, m, \beta)$
      - if $t > m$ then $m := t$
      - if $m \geq \beta$ then return($m$) // beta cut off
    - end; return $m$

- Algorithm $G_2'(\text{position } p, \text{ value } \alpha, \text{ value } \beta) \ // \ min \ node$
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$, then return $f(p)$ else begin
    - $m := \beta$
    - for $i := 1$ to $d$ do
      - $t := F_2'(p_i, \alpha, m)$
      - if $t < m$ then $m := t$
      - if $m \leq \alpha$ then return($m$) // alpha cut off
    - end; return $m$
Initial call: $F^2'(\text{root},-\infty,\infty)$

- $m = -\infty$

- call $G^2'(\text{node 1},-\infty,\infty)$
  - it is a terminal node
  - return value 15

- $t = 15$;
  - since $t > m$, $m$ is now 15

- call $G^2'(\text{node 2},15,\infty)$
  - call $F^2'(\text{node 2.1},15,\infty)$
  - it is a terminal node; return 10
  - $t = 10$; since $t < \infty$, $m$ is now 10
  - alpha is 15, $m$ is 10, so we have an alpha cut off
  - no need to call $F^2'(\text{node 2.2},15,10)$
  - ...

TCG: α-β Pruning, 20131106, Tsan-sheng Hsu
Alpha-beta pruning algorithm: Nega-max

- **Algorithm** $F_2$(position $p$, value $alpha$, value $beta$)
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$ // a terminal node
    or depth reaches the cutoff threshold // from iterative deepening
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - then return $h(p)$ else
  - begin
    \> $m := alpha$
    \> for $i := 1$ to $d$ do
    \> begin
    \> \> $t := -F_2(p_i, -beta, -m)$
    \> \> if $t > m$ then $m := t$
    \> \> if $m \geq beta$ then return$(m)$ // cut off
    \> end
  - end
  - return $m$
Examples
Lessons from the previous examples

- It looks like for the same tree, different move orderings give very different cut branches.
- It looks like if a node can evaluate a child with the best possible outcome earlier, then it can decide to cut earlier.
  - For a min node, this means to evaluate the child branch that gives the lowest value first.
  - For a max node, this means to evaluate the child branch that gives the highest value first.
Analysis of a possible best case

- **Q:** In the best possible scenario, what branches are cut?

- **Definitions:**
  - A path in a search tree is a sequence of numbers indicating the branches selected in each level using the Dewey decimal system.
  - A position is denoted as a path $a_1.a_2.\cdots.a_\ell$ from the root.
  - A position $a_1.a_2.\cdots.a_\ell$ is **critical** if
    - $a_i = 1$ for all even values of $i$ or
    - $a_i = 1$ for all odd values of $i$.
  - Note: as a special case, the root is critical.

- **Examples:**
  - $2.1.4.1.2$, $1.3.1.5.1.2$, $1.1.1.2.1.1.1.3$ and $1.1$ are critical
  - *Examples: $1.2.1.1.2$ is not critical*
Perfect-ordering tree

- A perfect-ordering tree:

\[ F(a_1 \cdots a_\ell) = \begin{cases} 
    h(a_1 \cdots a_\ell) & \text{if } a_1 \cdots a_\ell \text{ is a terminal} \\
    -F(a_1 \cdots a_\ell \cdot 1) & \text{otherwise}
\end{cases} \]

- The first successor of every non-terminal position gives the best possible value.
Theorem 1

Theorem 1: $F^2$ examines precisely the critical positions of a perfect-ordering tree.

Proof sketch:

- Classify the critical positions, a.k.a. nodes.
  - You must evaluate the first branch from the root to the bottom.
  - Alpha cut off happens at odd-depth nodes as soon as the first branch of this node is evaluated.
  - Beta cut off happens at even-depth nodes as soon as the first branch of this node is evaluated.

- For each type of nodes, try to associate them with the types of pruning occurred.
Types of nodes

Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where $j$ is the least index, if exists, such that $a_j \neq 1$ and $\ell$ is the last index.

- Def: let $IS_1(a_i)$ be a boolean function so that it is 0 if it is not the value 1 and it is 1 if it is.
  - We call this $IS_1$ parity of a number.

- If $j$ exists and $\ell > j$, then
  - $a_{j+1} = 1$ because this position is critical and thus the $IS_1$ parities of $a_j$ and $a_{j+1}$ are different.

- Since this position is critical, if $a_j \neq 1$, then $a_h = 1$ for any $h$ such that $h - j$ is odd.

- We now classify critical nodes into 3 types.
Type 1 nodes

- **Type 1**: the root, or a node with all the $a_i$ are 1;
  - This means $j$ does not exist.
  - Nodes on the leftmost branch.
  - The leftmost child of a type 1 node except the root.
Type 2 nodes

- Classification of critical positions \(a_1.a_2.\cdots.a_j.\cdots.a_\ell\) where \(j\) is the least index such that \(a_j \neq 1\) and \(\ell\) is the last index.

- **type 2**: \(\ell - j\) is zero or even;
  - type 2.1: \(\ell - j = 0\).
    - It is in the form of \(1.1.1.\cdots.1.1.1.a_\ell\) and \(a_\ell \neq 1\).
    - The non-leftmost children of a type 1 node.
  - type 2.2: \(\ell - j > 0\) and is even.
    - It is in the form of \(1.1.\cdots.1.1.a_j.1.a_{j+2}.\cdots.a_{\ell-2}.1.a_\ell\).
    - Note, we will show \(1.1.\cdots.1.1.a_j.1.a_{j+2}.\cdots.a_{\ell-2}.1\) is a type 3 node later.
    - All of the children of a type 3 node.
Type 3 nodes

- Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where $j$ is the least index such that $a_j \neq 1$ and $\ell$ is the last index.

- **type 3:** $\ell - j$ is odd;
  - type 3.1: $\ell = j + 1$.
    - It is of the form $1.1.\cdots.1.a_j.1$
    - The leftmost child of a type 2.1 node.
  - type 3.2: $\ell > j + 1$.
    - It is of the form $1.1.\cdots.1.a_j.1.a_{j+2}.1.\cdots.1.a_{\ell-1}.1$
    - The leftmost child of a type 2.2 node.
Comments

- Nodes of the same have common properties.
- These properties can be used in solving other problems.
  - Efficient parallel processing.
- Main techniques used: you cannot have two consecutive non-1 numbers in the ID of a critical node.
Illustration — critical nodes

1 1 1 1 1 1 * 1 * 1

1 1 1 : 1

1: not 1

*: any

TCG: α-β Pruning, 20131106, Tsan-sheng Hsu ©
Type 2.1 nodes

- Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where $j$ is the least index such that $a_j \neq 1$ and $\ell$ is the last index.

- type 2: $\ell - j$ is zero or even;
  - type 2.1: $\ell - j = 0$.
    - Then $\ell = j$.
    - It is in the form of $1.1.1.\cdots.1.1.1.a_\ell$ and $a_\ell \neq 1$.
    - The non-leftmost children of a type 1 node.
Type 3.1 nodes

- Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where $j$ is the least index such that $a_j \neq 1$ and $\ell$ is the last index.
- type 3: $\ell - j$ is odd;
  - type 3.1: $\ell = j + 1$.

$\triangleright$ It is of the form $1.1.\cdots.1.a_j.1$ and $a_\ell \neq 1$.

$\triangleright$ The leftmost child of a type 2.1 node.
Type 2.2 nodes

- Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where $j$ is the least index such that $a_j \neq 1$ and $\ell$ is the last index.

- type 2: $\ell - j$ is zero or even;
  - type 2.2: $\ell - j > 0$ and is even.
    - The IS1 parties of $a_j$ and $a_{j+1}$ are different.
      $\implies$ Since $a_j \neq 1$, $a_{j+1} = 1$.
    - $(\ell - 1) - j$ is odd:
      $\implies$ The IS1 parties of $a_{\ell-1}$ and $a_j$ are different.
      $\implies$ Since $a_j \neq 1$, $a_{\ell-1} = 1$.
    - It is in the form of $1.1.\cdots.1.1.a_j.1.a_{j+2}.\cdots.a_{\ell-2}.1.a_\ell$.
    - Note, we will show $1.1.\cdots.1.1.a_j.1.a_{j+2}.\cdots.a_{\ell-2}.1$ is a type 3 node later.
    - All of the children of a type 3 node.
Type 3.2 nodes

- Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where $j$ is the least index such that $a_j \neq 1$ and $\ell$ is the last index.

- **type 3:** $\ell - j$ is odd;
  - $a_j \neq 1$ and $\ell - j$ is odd
    - Since this position is critical, the IS1 parities of $a_j$ and $a_\ell$ are different.
      - $\implies a_\ell = 1$
      - $\implies a_{j+1} = 1$
  - It is in the form of
    - 1.1.\cdots.1.a_j.1.a_{j+2}.1.\cdots.1.a_{\ell-1}.1.

- The leftmost child of a **type 2 node**.

- **type 3.1:** $\ell = j + 1$.
  - It is of the form 1.1.\cdots.1.a_j.1
  - The leftmost child of a type 2.1 node.

- **type 3.2:** $\ell > j + 1$.
  - It is of the form 1.1.\cdots.1.a_j.1.a_{j+2}.1.\cdots.1.a_{\ell-1}.1
  - The leftmost child of a type 2.2 node.
Illustration — Types of nodes

- Type 1
- Type 2.1
- Type 2.2
- Type 3.1
- Type 3.2

TCG: α-β Pruning, 20131106, Tsan-sheng Hsu ©
Proof sketch for Theorem 1

Properties (invariants)

- A type 1 position $p$ is examined by calling $F2(p, -\infty, \infty)$
  - $p$’s first successor $p_1$ is of type 1
  - $F(p) = - F(p_1) \neq \pm \infty$
  - $p$’s other successors $p_2, \ldots, p_d$ are of type 2
  - $p_i, i > 1$, are examined by calling $F2(p_i, -\infty, F(p_1))$

- A type 2 position $p$ is examined by calling $F2(p, -\infty, beta)$ where $-\infty < beta \leq F(p)$
  - $p$’s first successor $p_1$ is of type 3
  - $F(p) = - F(p_1)$
  - $p$’s other successors $p_2, \ldots, p_d$ are not examined

- A type 3 position $p$ is examined by calling $F2(p, alpha, \infty)$ where $\infty > alpha \geq F(p)$
  - $p$’s successors $p_1, \ldots, p_d$ are of type 2
  - they are examined by calling $F2(p_1, -\infty, -alpha)$,
    $F2(p_2, -\infty, -\max\{m_1, alpha\}), \ldots,$ $F2(p_i, -\infty, -\max\{m_{i-1}, alpha\})$ where $m_i = F2(p_i, -\infty, -\max\{m_{i-1}, alpha\})$

- Using an inductive argument to prove all and also only critical positions are examined.
Corollary 1: Assume each position has exactly $d$ successors

- The number of positions examined by the alpha-beta procedure on level $i$ is exactly
  \[ d^{\lceil i/2 \rceil} + d^{\lfloor i/2 \rfloor} - 1. \]

Proof:

- There are $d^{\lfloor i/2 \rfloor}$ sequences of the form $a_1 \cdots a_i$ with $1 \leq a_i \leq d$ for all $i$ such that $a_i = 1$ for all odd values of $i$.
- There are $d^{\lceil i/2 \rceil}$ sequences of the form $a_1 \cdots a_i$ with $1 \leq a_i \leq d$ for all $i$ such that $a_i = 1$ for all even values of $i$.
- We subtract 1 for the sequence $1.1.\cdots.1.1$ which are counted twice.

Total number of nodes visited is

\[ \sum_{i=0}^{\ell} d^{\lceil i/2 \rceil} + d^{\lfloor i/2 \rfloor} - 1. \]
Analysis: average case

- **Assumptions:** Let a random game tree be generated in such a way that
  - each position on level $j$ has probability $q_j$ of being nonterminal
  - has an average of $d_j$ successors

- **Properties of the above random game tree**
  - Expected number of positions on level $\ell$ is $d_0 \cdot d_1 \cdots d_{\ell-1}$
  - Expected number of positions on level $\ell$ examined by an alpha-beta procedure assumed the random game tree is perfectly ordered is
    \[
    d_0 q_1 d_2 q_3 \cdots d_{\ell-2} q_{\ell-1} + q_0 d_1 q_2 d_3 \cdots q_{\ell-2} d_{\ell-1} - q_0 q_1 \cdots q_{\ell-1} \text{ if } \ell \text{ is even;}
    \]
    \[
    d_0 q_1 d_2 q_3 \cdots q_{\ell-2} d_{\ell-1} + q_0 d_1 q_2 d_3 \cdots d_{\ell-2} q_{\ell-1} - q_0 q_1 \cdots q_{\ell-1} \text{ if } \ell \text{ is odd}
    \]

- **Proof sketch:**
  - If $x$ is the expected number of positions of a certain type on level $j$, then $x d_j$ is the expected number of successors of these positions, and $x q_j$ is the expected number of “numbered 1” successors.
  - The above numbers equal to those of Corollary 1 when $q_j = 1$ and $d_j = d$ for $0 \leq j < \ell$. 
Perfect ordering is not always best

- Intuitively, we may “think” alpha-beta pruning would be most effective when a game tree is perfectly ordered.
  - That is, when the first successor of every position is the best possible move.
  - This is not always the case!

- Truly optimum order of game trees traversal is not obvious.
When is a branch pruned?

- Assume a node $r$ has two children $u$ and $v$ with $u$ being visited before $v$ using some move ordering.
  - Further assume $u$ produced a new bound $bound$.
- Assume node $v$ has a child $w$.
  - If the value $new$ returned from $w$ can cause a range conflict with $bound$, then branches of $v$ later than $w$ are cut.
- This means as long as the “relative” ordering of $u$ and $v$ are good enough, then we can have some cut-off.
  - There is no need for $r$ to have the best move ordering.
Theorem 2

Theorem 2: Alpha-beta pruning is optimum in the following sense:

- Given any game tree and any algorithm which computes the value of the root position, there is a way to permute the tree by reordering successor positions if necessary;
- so that every terminal position examined by the alpha-beta method under this permutation is examined by the given algorithm.
- Furthermore if the value of the root is not $\infty$ or $-\infty$, the alpha-beta procedure examines precisely the positions which are critical under this permutation.
Variations of alpha-beta search

- Initially, to search a tree with the root \( r \) by calling \( F^2(r, -\infty, +\infty) \).
  - What does it mean to search a tree with the root \( r \) by calling \( F^2(r, \alpha, \beta) \)?
    ▶ To search the tree rooted at \( r \) requiring that the returned value to be within \( \alpha \) and \( \beta \).

- In an alpha-beta search with a pre-assigned window \([\alpha, \beta]\):
  - Failed-high means it returns a value that is larger than or equal to its upper bound \( \beta \).
  - Failed-low means it returns a value that is smaller than or equal to its lower bound \( \alpha \).

- Variations:
  - Brute force Nega-Max version: \( F \)
    ▶ Always finds the correct answer according to the Nega-Max formula.
  - Fail hard alpha-beta cut (Nega-Max) version: \( F^2 \)
  - Fail soft alpha-beta cut (Nega-Max) version: \( F^3 \)
Fail hard version

- Original version.
- Algorithm $F^2(position \ p, \ value \ alpha, \ value \ beta)$
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$ // a terminal node
    - or depth reaches the cutoff threshold // from iterative deepening
    - or time is running up // from timing control
    - or some other constraints are met // add knowledge here
  - then return $h(p)$ else
  - begin
    - $m := alpha$ // hard initial value
    - for $i := 1$ to $d$ do
      - begin
        - $t := -F^2(p_i, -beta, -m)$
        - if $t > m$ then $m := t$ // the returned value is “used”
        - if $m \geq beta$ then return($m$) // cut off
      - end
    - end
  - end
  - return $m$
Properties and comments

**Properties:**
- \(\alpha < \beta\)
- \(F_2(p, \alpha, \beta) = \alpha\) if \(F(p) \leq \alpha\)
- \(F_2(p, \alpha, \beta) = F(p)\) if \(\alpha < F(p) < \beta\)
- \(F_2(p, \alpha, \beta) = \beta\) if \(F(p) \geq \beta\)
- \(F_2(p, -\infty, +\infty) = F(p)\)

**Comments:**
- \(F_2(p, \alpha, \beta)\): find the best possible value according to a nega-max formula for the position \(p\) with the constraints that
  - If \(F(p)\) is less than the lower bound \(\alpha\), then \(F_2(p, \alpha, \beta)\) returns with a value \(\alpha\) from a terminal position whose value is \(\leq \alpha\).
  - If \(F(p)\) is more than the upper bound \(\beta\), then \(F_2(p, \alpha, \beta)\) returns with value \(\beta\) from a terminal terminal position whose value is \(\geq \beta\).

- The meanings of \(\alpha\) and \(\beta\) during searching:
  - For a max node: the current best value is at least \(\alpha\).
  - For a min node: the current best value is at most \(\beta\).

- \(F_2\) always finds a value that is within \(\alpha\) and \(\beta\).
  - The bounds are hard, i.e., cannot be violated.
As long as the value of the leaf node $W$ is less than the current alpha value, the returned value of $A$ will be at least the returned value of $W$. 
Fail soft version

- **Algorithm** $F3(position \ p, \ value \ alpha, \ value \ beta)$
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$ // a terminal node
    - or depth reaches the cutoff threshold // from iterative deepening
    - or time is running up // from timing control
    - or some other constraints are met // add knowledge here
  - then return $h(p)$ else
  - begin
    - $m := -\infty$ // soft initial value
    - for $i := 1$ to $d$ do
      - begin
        - $t := -F3(p_i, -beta, -\max\{m, alpha\})$
        - if $t > m$ then $m := t$ // the returned value is “used”
        - if $m \geq beta$ then return$(m)$ // cut off
      - end
    - end
  - return $m$
Properties and comments

- Properties:
  - $\alpha < \beta$
  - $F_3(p, \alpha, \beta) \leq \alpha$ if $F(p) \leq F_3(p, \alpha, \beta) \leq \alpha$
  - $F_3(p, \alpha, \beta) = F(p)$ if $\alpha < F(p) < \beta$
  - $F_3(p, \alpha, \beta) \geq \beta$ if $F(p) \geq F_3(p, \alpha, \beta) \geq \beta$
  - $F_3(p, -\infty, +\infty) = F(p)$

- $F_3$ finds a “better” value when the value is out of the search window.
  - Better means a tighter bound.
  - The bounds are soft, i.e., can be violated.
  - When it fails high, $F_3$ normally returns a value that is higher than that of $F_2$.
    - Never higher than that of $F$!
  - When it fails low, $F_3$ normally returns a value that is lower than that of $F_2$.
    - Never lower than that of $F$!
Let the value of the leaf node \( W \) be \( u \).

- If \( u < \text{alpha} \), then the branch at \( W \) will have a returned value of at least \( u \).
Comparisons between $F_2$ and $F_3$

- Both versions find the corrected value $v$ if $v$ is within the window $[\alpha, \beta]$.
- Both versions scan the same set of nodes during searching.
  - If the returned value of a subtree is decided by a cut, then $F_2$ and $F_3$ return the same value.
- $F_3$ provides more information when the true value is out of the pre-assigned search window.
  - Can provide a feeling on how bad or good the game tree is.
  - Use this “better” value to guide searching later on.
- $F_3$ saves about 7% of time than that of $F_2$ when a transposition table is used to save and re-use searched results [Fishburn 1983].
  - A transposition table is a data structure to record the results of previous searched results.
  - The entries of a transposition table can be efficiently accessed, i.e., read and write, during searching.
  - Need an efficient addressing scheme, e.g., hash, to translate between a position and its address.
Assume the node $A$ can be reached from the starting position using path $P_1$ and path $P_2$.

- If $W$ is visited first along $P_1$ with a bound of $[4000, 5000]$, and returns a value of 200, then
  - the returned value of $W$, 200, is stored into the transposition table.

- If $A$ is visited again along $P_2$ with a bound of $[400, 500]$, then a better value of previously stored value of $W$ helps to decide whether the subtree rooted at $W$ needs to be searched again.
**F2 and F3: Example (2/2)**

- Fail soft version has a chance to record a better value to be used later when this position is revisited.
  - If A is visited again along $P_2$ with a bound of [400, 500], then
    - it does not need to be searched again, since the previous stored value of $W$ is $-200$.
  - However, if the value of $W$ is 450, then it needs to be searched again.

- The fail hard version does not store the returned value of $W$ after its first visit since this value is less than $\alpha$.
Questions

- What move ordering is good?
  - It may not be good to search the best possible move first.
  - It maybe better to cut off a branch with more nodes first.
- How about the case when the tree is not uniform?
- What is the effect of using iterative-deepening alpha-beta cut off?
- How about the case for searching a game graph instead of a game tree?
  - Can some nodes be visited more than once?
References and further readings

