Scout, NegaScout and Proof-Number Search

Tsan-sheng Hsu

徐讙昇

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu
It looks like alpha-beta pruning is the best we can do for a generic searching procedure.
- What else can be done generically?
- Alpha-beta pruning follows basically the “intelligent” searching behaviors used by human when domain knowledge is not involved.
- Can we find some other “intelligent” behaviors used by human during searching?

Intuition: MAX node.
- Suppose we know currently we have a way to gain at least 300 points at the first branch.
- If there is an efficient way to know the second branch is at most gaining 300 points, then there is no need to search the second branch in detail.
  - Alpha-beta cut algorithm is one way to make sure of this exactly.
  - Is there a way to search a tree approximately?
  - Is searching approximately faster than searching exactly?

Similar intuition holds for a MIN node.
**SCOUT procedure**

- It may be possible to verify whether the value of a branch is greater than a value $v$ or not in a way that is faster than knowing its exact value [Judea Pearl 1980].

**High level idea:**

- While searching a branch $T_i$ of a MAX node, if we have already obtained a lower bound $v_l$.
  - First TEST whether it is possible for $T_i$ to return something greater than $v_l$.
  - If FALSE, then there is no need to search $T_i$. This is called **fails the test**.
  - If TRUE, then search $T_i$. This is called **passes the test**.

- While searching a branch $T_j$ of a MIN node, if we have already obtained an upper bound $v_u$.
  - First TEST whether it is possible for $T_j$ to return something smaller than $v_u$.
  - If FALSE, then there is no need to search $T_j$. This is called **fails the test**.
  - If TRUE, then search $T_j$. This is called **passes the test**.
How to TEST $> v$

procedure TEST(position $p$, condition $>\), value $v$)
// test whether the value of the branch at $p$ is $> v$

determine the successor positions $p_1, \ldots, p_b$ of $p$

if $b = 0$, then // terminal

$\triangleright$ if $f(p) > v$ then // $f()$: evaluating function
$\triangleright$ return TRUE
$\triangleright$ else return FALSE

if $p$ is a MAX node, then

$\bullet$ for $i := 1$ to $b$ do

$\triangleright$ if TEST($p_i, >\), v$) is TRUE, then
return TRUE // succeed if a branch is $> v$

$\bullet$ return FALSE // fail only if all branches $\leq v$

if $p$ is a MIN node, then

$\bullet$ for $i := 1$ to $b$ do

$\triangleright$ if TEST($p_i, >\), v$) is FALSE, then
return FALSE // fail if a branch is $\leq v$

$\bullet$ return TRUE // succeed only if all branches are $> v$
Illustration of TEST

TCG: Scout, NegaScout, PN-search, 20151204, Tsan-sheng Hsu ©
How to TEST — Discussions

- Sometimes it may be needed to test for “\(\geq v\)”, “\(< v\)” or “\(\leq v\)”.

\[
\begin{align*}
\text{TEST}(p, >, v) \text{ is TRUE} & \equiv \text{TEST}(p, \leq, v) \text{ is FALSE} \\
\text{TEST}(p, >, v) \text{ is FALSE} & \equiv \text{TEST}(p, \leq, v) \text{ is TRUE} \\
\text{TEST}(p, <, v) \text{ is TRUE} & \equiv \text{TEST}(p, \geq, v) \text{ is FALSE} \\
\text{TEST}(p, <, v) \text{ is FALSE} & \equiv \text{TEST}(p, \geq, v) \text{ is TRUE}
\end{align*}
\]

- **Practical consideration:**
  - Set a depth limit and evaluate the position’s value when the limit is reached.
Main SCOUT procedure

Algorithm SCOUT(position $p$)

- determine the successor positions $p_1, \ldots, p_b$
- if $b = 0$, then return $f(p)$
  else $v = SCOUT(p_1)$ // SCOUT the first branch
- if $p$ is a MAX node
  • for $i := 2$ to $b$ do
    ▷ if TEST($p_i$, $>$, $v$) is TRUE, // TEST first for the rest of the branches
        then $v = SCOUT(p_i)$ // find the value of this branch if it can be $> v$
- if $p$ is a MIN node
  • for $i := 2$ to $b$ do
    ▷ if TEST($p_i$, $<$, $v$) is TRUE, // TEST first for the rest of the branches
        then $v = SCOUT(p_i)$ // find the value of this branch if it can be $< v$
- return $v$
How to TEST $< v$

procedure TEST(position $p$, condition $<$, value $v$)
// test whether the value of the branch at $p$ is $< v$

- determine the successor positions $p_1, \ldots, p_b$ of $p$
- if $b = 0$, then // terminal
  > if $f(p) < v$ then // $f()$: evaluating function
  > > return TRUE
  > > else return FALSE
- if $p$ is a MAX node, then
  > for $i := 1$ to $b$ do
  > > if TEST($p_i$, $<$, $v$) is FALSE, then
  > > > return FALSE // fail if a branch is $\geq v$
  > > return TRUE // succeed only if all branches $< v$
- if $p$ is a MIN node, then
  > for $i := 1$ to $b$ do
  > > if TEST($p_i$, $<$, $v$) is TRUE, then
  > > > return TRUE // succeed if a branch is $< v$
  > > return FALSE // fail only if all branches are $\geq v$
Note that $v$ is the current best value at any moment.

**MAX node:**
- For any $i > 1$, if $\text{TEST}(p_i, >, v)$ is TRUE,
  - then the value returned by $\text{SCOUT}(p_i)$ must be greater than $v$.
- We say the $p_i$ passes the test if $\text{TEST}(p_i, >, v)$ is TRUE.

**MIN node:**
- For any $i > 1$, if $\text{TEST}(p_i, <, v)$ is TRUE,
  - then the value returned by $\text{SCOUT}(p_i)$ must be smaller than $v$.
- We say the $p_i$ passes the test if $\text{TEST}(p_i, <, v)$ is TRUE.
TEST which is called by SCOUT may visit less nodes than that of alpha-beta.

- Assume \( TEST(p, >, 5) \) is called by the root after the first branch is evaluated.
  - It calls \( TEST(K, >, 5) \) which skips \( K \)'s second branch.
  - \( TEST(p, >, 5) \) is FALSE, i.e., fails the test, after returning from the 3rd branch.
  - No need to do SCOUT for the branch \( p \).
- Alpha-beta needs to visit \( K \)'s second branch.
SCOUT may pay many visits to a node that is cut off by alpha-beta.
Number of nodes visited (1/4)

- For TEST to return TRUE for a subtree $T$, it needs to evaluate at least
  - one child for a MAX node in $T$, and
  - and all of the children for a MIN node in $T$.
  - If $T$ has a fixed branching factor $b$ and uniform depth $b$, the number of nodes evaluated is $\Omega(b^{\ell/2})$ where $\ell$ is the depth of the tree.

- For TEST to return FALSE for a subtree $T$, it needs to evaluate at least
  - one child for a MIN node in $T$, and
  - and all of the children for a MAX node in $T$.
  - If $T$ has a fixed branching factor $b$ and uniform depth $b$, the number of nodes evaluated is $\Omega(b^{\ell/2})$. 
**Number of nodes visited (2/4)**

- **Assumptions:**
  - Assume a full complete $b$-ary tree with depth $\ell$ where $\ell$ is even.
  - The depth of the root, which is a MAX node, is 0.

- **The total number of nodes in the tree is** $\frac{b^{\ell+1} - 1}{b-1}$.

- $H_1$: the minimum number of nodes visited by TEST when it returns TRUE.

\[
H_1 = 1 + 1 + b + b + b^2 + b^2 + b^3 + b^3 + \cdots + b^{\ell/2-1} + b^{\ell/2-1} + b^{\ell/2}
\]

\[
= 2 \cdot (b^0 + b^1 + \cdots + b^{\ell/2}) - b^{\ell/2}
\]

\[
= 2 \cdot \frac{b^{\ell/2+1} - 1}{b-1} - b^{\ell/2}
\]
Number of nodes visited (3/4)

- Assumptions:
  - Assume a full complete $b$-ary tree with depth $\ell$ where $\ell$ is even.
  - The depth of the root, which is a MAX node, is 0.
- $H_2$: the minimum number of nodes visited by alpha-beta.

$$
H_2 = \sum_{i=0}^{\ell} (b^\lceil i/2 \rceil + b^\lfloor i/2 \rfloor - 1)
$$

$$
= \sum_{i=0}^{\ell} b^\lceil i/2 \rceil + \sum_{i=0}^{\ell} b^\lfloor i/2 \rfloor - (\ell + 1)
$$

$$
= \sum_{i=0}^{\ell} b^\lceil i/2 \rceil + H_1 - (\ell + 1)
$$

$$
= (1 + b + b + \cdots + b^{\ell/2-1} + b^{\ell/2} + b^{\ell/2}) + H_1 - (\ell + 1)
$$

$$
= (H_1 - 1 + b^{\ell/2} - b^{\ell/2-1}) + H_1 - (\ell + 1)
$$

$$
= 2 \cdot H_1 + b^{\ell/2} - b^{\ell/2-1} - (\ell + 2)
$$

$$
\sim (2.x) \cdot H_1
$$
Number of nodes visited (4/4)
Comparisons

- When the first branch of a node has the best value, then TEST scans the tree fast.
  - The best value of the first $(i-1)$ branches is used to test whether the $i$th branch needs to be searched exactly.
  - If the value of the first $(i-1)$ branches of the root is better than the value of $i$th branch, then we do not have to evaluate exactly for the $i$th branch.

- Compared to alpha-beta pruning whose cut off comes from bounds of search windows.
  - It is possible to have some cut-off for alpha-beta as long as there are some relative move orderings are “good.”
    ▶ The moving orders of your children and the children of your ancestor who is odd level up decide a cut-off.

  - The search bound is updated during the searching.
    ▶ Sometimes, a deep alpha-beta cut-off occurs because a bound found from your ancestor a distance away.
A node may be visited more than once.
- First visit is to TEST.
- Second visit is to SCOUT.
  ▶ During SCOUT, it may be TESTed with a different value.
- Q: Can information obtained in the first search be used in the second search?

SCOUT is a recursive procedure.
- A node in a branch that is not the first child of a node with a depth of \( \ell \).
  ▶ Note that the depth of the root is defined to be 0.
  ▶ Every ancestor of you may initiate a TEST to visit you.
  ▶ It can be visited \( \ell \) times by TEST.
Performance of SCOUT (2/2)

- Show great improvements on $depth > 3$ for games with small branching factors.
  - It traverses most of the nodes without evaluating them preciously.
  - Few subtrees remained to be revisited to compute their exact mini-max values.

- Experimental data on the game of Kalah show [UCLA Tech Rep UCLA-ENG-80-17, Noe 1980]:
  - SCOUT favors “skinny” game trees, that are game trees with high depth-to-width ratios.
  - On $depth = 5$, it saves over 40% of time.
  - Maybe bad for games with a large branching factor.
  - Move ordering is very important.
    - The first branch, if is good, offers a great chance of pruning further branches.
Alpha-beta revisited

- **In an alpha-beta search with a 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 \(\alpha\).

- **Null or Zero window search**:
  - Using alpha-beta search with the window \([m, m+1]\).
  - The result can be either failed-high or failed-low.
  - Failed-high means the return value is at least \(m + 1\).
    - ▶ **Equivalent to** \(TEST(p, >, m)\) **is true.**
  - Failed-low means the return value is at most \(m\).
    - ▶ **Equivalent to** \(TEST(p, >, m)\) **is false.**
Alpha-Beta + Scout

- Intuition:
  - Try to incorporate SCOUT and alpha-beta together.
  - The searching window of alpha-beta if properly set can be used as TEST in SCOUT.
  - Using a searching window is better than using a single bound as in SCOUT.
  - Can also apply alpha-beta cut if it applies.

- Modifications to the SCOUT algorithm:
  - Traverse the tree with two bounds as the alpha-beta procedure does.
    - A searching window.
    - Use the current best bound to guide the value used in TEST.
  - Use a fail soft version to get a better result when the returned value is out of the window.
Algorithm $F4'(position \ p, \ value \ alpha, \ value \ beta, \ integer \ depth)$

- determine the successor positions $p_1, \ldots, p_b$
- if $b = 0$ // a terminal node
  - or depth = 0 // depth is the remaining depth to search
  - or time is running up // from timing control
  - or some other constraints are met // apply heuristic here
- then return $f(p)$ else
  begin
  - $m := -\infty$ // $m$ is the current best lower bound; fail soft
  - $m := \max\{m, G4'(p_1, \alpha, \beta, \text{depth} - 1)\}$ // the first branch
    - if $m \geq \beta$ then return($m$) // beta cut off
  - for $i := 2$ to $b$ do
  -  9: $t := G4'(p_i, m, m + 1, \text{depth} - 1)$ // null window search
  -  10: if $t > m$ then // failed-high
     11: if (depth < 3 or $t \geq \beta$)
     12: then $m := t$
     13: else $m := G4'(p_i, t, \beta, \text{depth} - 1)$ // re-search
  - 14: if $m \geq \beta$ then return($m$) // beta cut off
  end
- return $m$
The NegaScout Algorithm – MiniMax (2/2)

- Algorithm $G4'(\text{position } p, \text{ value } \alpha, \text{ value } \beta, \text{ integer } \text{depth})$

  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // a terminal node
    - or $\text{depth} = 0$ // $\text{depth}$ is the remaining depth to search
    - or time is running up // from timing control
    - or some other constraints are met // apply heuristic here
  - then return $f(p)$ else
  - begin
    - ▶ $m = \infty$ // $m$ is the current best upper bound; fail soft
    - $m := \min\{m, F4'(p_1, \alpha, \beta, \text{depth} - 1)\}$ // the first branch
      - if $m \leq \alpha$ then return($m$) // alpha cut off
    - ▶ for $i := 2$ to $b$ do
      - ▶ 9: $t := F4'(p_i, m - 1, m, \text{depth} - 1)$ // null window search
      - ▶ 10: if $t < m$ then // failed-low
        - ▶ 11: if ($\text{depth} < 3$ or $t \leq \alpha$)
        - ▶ 12: then $m := t$
        - ▶ 13: else $m := F4'(p_i, \alpha, t, \text{depth} - 1)$ // re-search
      - ▶ 14: if $m \leq \alpha$ then return($m$) // alpha cut off
    - end
  - return $m$
NegaScout – MiniMax version (1/2)
The NegaScout Algorithm

- Use Nega-MAX format.
- Algorithm $F_4(position \ p, \ value \ \alpha, \ value \ \beta, \ integer \ depth)$
  
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // a terminal node
    - or $depth = 0$ // $depth$ is the remaining depth to search
    - or time is running up // from timing control
    - or some other constraints are met // apply heuristic here
  - then return $h(p)$ else
    - $m := -\infty$ // the current lower bound; fail soft
    - $n := \beta$ // the current upper bound
    - for $i := 1$ to $b$ do
      - 9: $t := -F_4(p_i, -n, -\max\{\alpha, m\}, depth - 1)$
      - 10: if $t > m$ then
        - 11: if ($n = \beta$ or $depth < 3$ or $t \geq \beta$)
        - 12: then $m := t$
        - 13: else $m := -F_4(p_i, -\beta, -t, depth - 1)$ // re-search
      - 14: if $m \geq \beta$ then return$(m)$ // cut off
      - 15: $n := \max\{\alpha, m\} + 1$ // set up a null window
  - return $m$
Search behaviors (1/3)

- If the depth is enough or it is a terminal position, then stop searching further.
  - Return $h(p)$ as the value computed by an evaluation function.
  - Note:
    $$h(p) = \begin{cases} f(p) & \text{if depth of } p \text{ is 0 or even} \\ -f(p) & \text{if depth of } p \text{ is odd} \end{cases}$$

- Fail soft version.

- For the first child $p_1$, search using the normal alpha beta window.
  - line 9: normal window for the first child
  - the initial value of $m$ is $-\infty$, hence $-\max\{\alpha, m\} = -\alpha$
    ▶ $m$ is the current best value
  - that is, searching with the normal window $[\alpha, \beta]$
Search behaviors (2/3)

- For the second child and beyond \( p_i, i > 1 \), first perform a null window search for testing whether \( m \) is the answer.
  - line 9: a null-window of \([n - 1, n]\) searches for the second child and beyond where \( n = \max\{\alpha, m\} + 1 \).
    - \( m \) is best value obtained so far
    - \( \alpha \) is the previous lower bound
    - \( m \)'s value will be first set at line 12 because \( n = \beta \)
    - The value of \( n \) is set at line 15.
  - line 11:
    - \( n = \beta \): we are at first iteration.
    - \( \text{depth} < 3 \): on a smaller depth subtree, i.e., depth at most 2, NegaScout always returns the best value.
    - \( t \geq \beta \): we have obtained a good enough value from the failed-soft version to guarantee a beta cut.
For the second child and beyond $p_i, i > 1$, first perform a null window search for testing whether $m$ is the answer.

- **line 11**: on a smaller depth subtree, i.e., depth at most 2, NegaScout always returns the best value.
  - Normally, no need to do alpha-beta or any enhancement on very small subtrees.
  - The overhead is too large on small subtrees.

- **line 13**: re-search when the null window search fails high.
  - The value of this subtree is at least $t$.
  - This means the best value in this subtree is more than $m$, the current best value.
  - This subtree must be re-searched with the the window $[t, \beta]$.

- **line 14**: the normal pruning from alpha-beta.
Example for NegaScout
Refinements

- When a subtree is re-searched, it is best to use information on the previous search to speed up the current search.
  - Restart from the position that the value $t$ is returned.
- Maybe want to re-search using the normal alpha-beta procedure.
- $F_4$ runs much better with a good move ordering and transposition tables.
  - Order the moves in a priority list.
  - Reduce the number of re-searches.
Performances

- Experiments done on a uniform random game tree [Reinefeld 1983].
  - Normally superior to alpha-beta when searching game trees with branching factors from 20 to 60.
  - Shows about 10 to 20% of improvement.
Comments

- Incooperating both SCOUT and alpha-beta.
- Used in state-of-the-art game search engines.
- The first search, though maybe unsuccessful, can provide useful information in the second search.
  - Information can be stored and then be reused.
Ideas for new search methods

- Consider the case of a 2-player game tree with either 0 or 1 on the leaves.
  - win, or not win which is lose or draw;
  - lose, or not lose which is win or draw;
  - Call this a binary valued game tree.

- If the game tree is known as well as the values of some leaves are known, can you make use of this information to search this game tree faster?
  - The value of the root is either 0 or 1.
  - If a branch of the root returns 1, then we know for sure the value of the root is 1.
  - The value of the root is 0 only when all branches of the root returns 0.
  - An AND-OR game tree search.
Which node to search next?

- **A most proving node** for a node $u$: a node if its value is 1, then the value of $u$ is 1.
- **A most disproving node** for a node $u$: a node if its value is 0, then the value of $u$ is 0.
Proof or Disproof Number

- Assign a **proof number** and a **disproof number** to each node $u$ in a binary valued game tree.
  - $\text{proof}(u)$: the minimum number of leaves needed to visited in order for the value of $u$ to be 1.
  - $\text{disproof}(u)$: the minimum number of leaves needed to visited in order for the value of $u$ to be 0.
Proof Number: Definition

- **u is a leaf:**
  - If \( \text{value}(u) \) is unknown, then \( \text{proof}(u) \) is the cost of evaluating \( u \).
  - If \( \text{value}(u) \) is 1, then \( \text{proof}(u) = 0 \).
  - If \( \text{value}(u) \) is 0, then \( \text{proof}(u) = \infty \).

- **u is an internal node with all of the children \( u_1, \ldots, u_b \):**
  - if \( u \) is a MAX node,
    \[
    \text{proof}(u) = \min_{i=1}^{i=b} \text{proof}(u_i);
    \]
  - if \( u \) is a MIN node,
    \[
    \text{proof}(u) = \sum_{i=1}^{i=b} \text{proof}(u_i).
    \]
Disproof Number: Definition

- **u is a leaf:**
  - If \( \text{value}(u) \) is unknown, then \( \text{disproof}(u) \) is cost of evaluating \( u \).
  - If \( \text{value}(u) \) is 1, then \( \text{disproof}(u) = \infty \).
  - If \( \text{value}(u) \) is 0, then \( \text{disproof}(u) = 0 \).

- **u is an internal node with all of the children \( u_1, \ldots, u_b \):**
  - if \( u \) is a MAX node,
    \[
    \text{disproof}(u) = \sum_{i=1}^{i=b} \text{disproof}(u_i);
    \]
  - if \( u \) is a MIN node,
    \[
    \text{disproof}(u) = \min_{i=1}^{i=b} \text{disproof}(u_i).
    \]
Illustrations
How to use these Numbers

- If the numbers are known in advance, then from the root, we search a child $u$ with the value equals to $\min\{\text{proof}(\text{root}), \text{disproof}(\text{root})\}$.
  - Then we find a path from the root towards a leaf recursively as follows,
    - if we try to prove it, then pick a child with the least proof number for a MAX node, and pick any node that has a chance to be proved for a MIN node.
    - if we try to disprove it, then pick a child with the least disproof number for a MIN node, and pick any node that has a chance to be disproved for a MAX node.

- Assume each leaf takes a lot of time to evaluate.
  - For example, the game tree represents an open game tree or an endgame tree.
  - Depends on the results we have so far, pick the next leaf to prove or disprove.

- Need to be able to update these numbers on the fly.
PN-search: algorithm

- **loop**: Compute or update proof and disproof numbers for each node in a bottom up fashion.
  - If proof(root) = 0 or disproof(root) = 0, then we are done, otherwise
    - proof(root) ≤ disproof(root): we try to prove it.
    - proof(root) > disproof(root): we try to disprove it.

- \( u \leftarrow \text{root}; \) \{ * find the leaf to prove or disprove * \}
  - if we try to prove, then
    - while \( u \) is not a leaf do
      - if \( u \) is a MAX node, then
        \( u \leftarrow \text{leftmost child of } u \) with the smallest non-zero proof number;
      - if current is a MIN node, then
        \( u \leftarrow \text{leftmost child of } u \) with a non-zero proof number;
  - if we try to disprove, then
    - while \( u \) is not a leaf do
      - if \( u \) is a MAX node, then
        \( u \leftarrow \text{leftmost child of } u \) with a non-zero disproof number;
      - if current is a MIN node, then
        \( u \leftarrow \text{leftmost child of } u \) with the smallest non-zero disproof number;

- Prove or disprove \( u \); go to loop;
Multi-Valued game Tree

- The values of the leaves may not be binary.
  - Assume the values are non-negative integers.
  - Note: it can be in any finite countable domain.
- Revision of the proof and disproof numbers.
  - $\text{proof}_v(u)$: the minimum number of leaves needed to visited in order for the value of $u$ to $\geq v$.
    - $\triangleright \text{proof}(u) \equiv \text{proof}_1(u)$.
  - $\text{disproof}_v(u)$: the minimum number of leaves needed to visited in order for the value of $u$ to $< v$.
    - $\triangleright \text{disproof}(u) \equiv \text{disproof}_1(u)$. 
Illustration
Multi-Valued Proof Number

- **u** is a leaf:
  - If \( \text{value}(u) \) is unknown, then \( \text{proof}_v(u) \) is cost of evaluating \( u \).
  - If \( \text{value}(u) \geq v \), then \( \text{proof}_v(u) = 0 \).
  - If \( \text{value}(u) < v \), then \( \text{proof}_v(u) = \infty \).

- **u** is an internal node with all of the children \( u_1, \ldots, u_b \):
  - if \( u \) is a MAX node,
    \[
    \text{proof}_v(u) = \min_{i=1}^{i=b} \text{proof}_v(u_i);
    \]
  - if \( u \) is a MIN node,
    \[
    \text{proof}_v(u) = \sum_{i=1}^{i=b} \text{proof}_v(u_i).\]
Multi-valued Disproof Number

- **\( u \) is a leaf:**
  - If value(\( u \)) is unknown, then disproof\(_v(u)\) is cost of evaluating \( u \).
  - If value(\( u \)) \( \geq v \), then disproof\(_v(u)\) = \( \infty \).
  - If value(\( u \)) < \( v \), then disproof\(_v(u)\) = 0.

- **\( u \) is an internal node with all of the children \( u_1, \ldots, u_b \):**
  - if \( u \) is a MAX node,
    \[
    \text{disproof}_v(u) = \sum_{i=1}^{i=b} \text{disproof}_v(u_i);
    \]
  - if \( u \) is a MIN node,
    \[
    \text{disproof}_v(u) = \min_{i=1}^{i=b} \text{disproof}_v(u_i).
    \]

TCG: Scout, NegaScout, PN-search, 20151204, Tsan-sheng Hsu ©
Revised PN-search(ν): algorithm

- **loop:** Compute or update proof_ν and disproof_ν numbers for each node in a bottom up fashion.
  - If proof_ν(root) = 0 or disproof_ν(root) = 0, then we are done, otherwise
    - proof_ν(root) ≤ disproof_ν(root): we try to prove it.
    - proof_ν(root) > disproof_ν(root): we try to disprove it.

- u ← root; {※ find the leaf to prove or disprove ※}
  - if we try to prove, then
    - while u is not a leaf do
      - if u is a MAX node, then
        - u ← leftmost child of u with the smallest non-zero proof_ν number;
      - if current is a MIN node, then
        - u ← leftmost child of u with a non-zero proof_ν number;
  - if we try to disprove, then
    - while u is not a leaf do
      - if u is a MAX node, then
        - u ← leftmost child of u with a non-zero disproof_ν number;
      - if current is a MIN node, then
        - u ← leftmost child of u with the smallest non-zero disproof_ν number;

- Prove or disprove u; go to loop;
Multi-valued PN-search: algorithm

- When the values of the leaves are not binary, use an open value binary search to find an upper bound of the value.
  - Set the initial value of $v$ to be 1.
  - **loop:** `PN-search(v)`
    - Prove the value of the search tree is $\geq v$ or disprove it by showing it is $< v$.
  - If it is proved, then double the value of $v$ and go to **loop** again.
  - If it is disproved, then the true value of the tree is between $\lfloor v/2 \rfloor$ and $v - 1$.
  - { * Use a binary search to find the exact returned value of the tree. * }
- $low \leftarrow \lfloor v/2 \rfloor$; $high \leftarrow v - 1$;
- **while** $low \leq high$ **do**
  - if $low = high$, then return $low$ as the tree value
  - $mid \leftarrow \lfloor (low + high)/2 \rfloor$
  - `PN-search(mid)`
  - if it is disproved, then $high \leftarrow mid - 1$
  - else if it is proved, then $low \leftarrow mid
Comments

- Appears to be good for searching certain types of game trees.
  - Find the easiest way to prove or disprove a conjecture.
  - A dynamic strategy depends on work has been done so far.
- Performance has nothing to do with move ordering.
  - Performance of most previous algorithms depends heavily on whether a good move ordering can be found.
- Searching the “easiest” branch may not give you the best performance.
  - Performance depends on the value of each internal nodes.
- Commonly used in verifying conjectures, e.g., first-player win.
  - Partition the opening moves in a tree-like fashion.
  - Try to the “easiest” way to prove or disprove the given conjecture.
- Take into consideration the fact that some nodes may need more time to process than the other nodes.
References and further readings

