Theory of Computer Games: Selected Advanced Topics

Tsan-sheng Hsu

http://www.iis.sinica.edu.tw/~tshsu
Abstract

- Some advanced research issues.
  - The graph history interaction (GHI) problem.
  - Opponent models.
  - Searching chance nodes.
  - Proof-number search.
Graph history interaction problem

- The graph history interaction (GHI) problem [Campbell 1985]:
  - In a game graph, a position can be visited by more than one path from a starting position.
  - The value of the position depends on the path visiting it.
    - It can be win, loss or draw for Chinese chess.
    - It can only be draw for Western chess and Chinese dark chess.
    - It can only be loss for Go.

- In the transposition table, you record the value of a position, but not the path leading to it.
  - Values computed from rules on repetition cannot be used later on.
  - It takes a huge amount of storage to store all the paths visiting it.

- This is a very difficult problem to be solved in real time [Wu et al '05].
GHI problem – example

• Assume the one causes loops loses the game.
GHI problem – example

- Assume the one causes loops loses the game.
- $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ is loss because of rules of repetition.
  - Memorized $J$ as a loss position.
GHI problem – example

- Assume the one causes loops loses the game.
- $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ is loss because of rules of repetition. ▶ Memorized $J$ as a loss position.
- $A \rightarrow B \rightarrow D \rightarrow H$ is a win. Hence $D$ is win.
GHI problem – example

• Assume the one causes loops loses the game.

• $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ is loss because of rules of repetition.
  ▷ Memorized $J$ as a loss position.

• $A \rightarrow B \rightarrow D \rightarrow H$ is a win. Hence $D$ is win.

• $A \rightarrow B \rightarrow E$ is a loss. Hence $B$ is loss.
GHI problem – example

• Assume the one causes loops loses the game.
• $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ is loss because of rules of repetition.
  ▷ Memorized $J$ as a loss position.
• $A \rightarrow B \rightarrow D \rightarrow H$ is a win. Hence $D$ is win.
• $A \rightarrow B \rightarrow E$ is a loss. Hence $B$ is loss.
• $A \rightarrow C \rightarrow F \rightarrow J$ is loss because $J$ is recorded as loss.
• $A$ is loss because both branches lead to loss.
GHI problem – example

• Assume the one causes loops loses the game.
• $A \to B \to D \to G \to I \to J \to D$ is loss because of rules of repetition. ▷ Memorized $J$ as a loss position.

• $A \to B \to D \to H$ is a win. Hence $D$ is win.
• $A \to B \to E$ is a loss. Hence $B$ is loss.
• $A \to C \to F \to J$ is loss because $J$ is recorded as loss.
• $A$ is loss because both branches lead to loss.
• However, $A \to C \to F \to J \to D \to H$ is a win.
Comments

- Using DFS to search the above game graph from left first or from right first produces two different results.
- Position $A$ is actually a win position.
  - Problem: memorize $J$ is a loss is only valid when the path leading to it causes a loop.
- Storing the path leading to a position in a transposition table requires too much memory.
- It is still a research problem to use a more efficient data structure.
Opponent models

- In a normal alpha-beta search, it is assumed that you and the opponent use the same strategy.
  - What is good to you is bad to the opponent and vice versa!
  - Hence we can reduce a minimax search to a NegaMax search.
  - This is normally true when the game ends, but may not be true in the middle of the game.

- What will happen when there are two strategies or evaluating functions $f_1$ and $f_2$ so that
  - for some positions $p$, $f_1(p)$ is better than $f_2(p)$
    - “better” means closer to the real value $f(p)$
  - for some positions $q$, $f_2(q)$ is better than $f_1(q)$

- If you are using $f_1$ and you know your opponent is using $f_2$, what can be done to take advantage of this information.
  - This is called OM (opponent model) search [Carmel and Markovitch 1996].
    - In a MAX node, use $f_1$.
    - In a MIN node, use $f_2$. 
Opponent models – comments

- **Comments:**
  - Need to know your opponent’s model precisely or to have some knowledge about your opponent.
  - How to learn the opponent model on-line or off-line?
  - When there are more than 2 possible opponent strategies, use a probability model (PrOM search) to form a strategy.
Search with chance nodes

- **Chinese dark chess**
  - Two player, zero sum, **complete information**
  - **Perfect information**
  - **Stochastic**
  - There is a **chance** node during searching [Ballard 1983].
    - The value of a chance node is a distribution, not a fixed value.

- **Previous work**
  - Alpha-beta based [Ballard 1983]
  - Monte-Carlo based [Lancoto et al 2013]
Example (1/3)

- It's black turn and black has 6 different possible legal moves including 4 of them being moving its elephant and two flipping moves at a1 or a8.
  - It is difficult for black to secure a win by moving its elephant in all 3 possible directions or capturing the red pawn at left.
Example (2/3)

- If black flips a1, then it becomes one of the 2 following cases.
  - If a1 is black cannon, then it is difficult for red to win.
  - If a1 is black king, then it is difficult for black to lose.
- If black flips \( a8 \), then it becomes one of the 2 following cases.
  - If \( a8 \) is black cannon, then red cannon captures it immediately and results in a black lose.
  - If \( a8 \) is black king, then red cannon captures it immediately and results in a black lose.
Basic ideas for searching chance nodes

- Assume a chance node \( x \) has a score probability distribution function \( Pr(*) \) with the range of possible outcomes from 1 to \( N \) where \( N \) is a positive integer.
  - For each possible outcome \( i \), we need to compute \( \text{score}(i) \).
  - The expected value \( E = \sum_{i=1}^{N} \text{score}(i) \times Pr(x = i) \).
  - The minimum value is \( m = \min_{i=1}^{N} \{ \text{score}(i) \mid Pr(x = i) > 0 \} \).
  - The maximum value is \( M = \max_{i=1}^{N} \{ \text{score}(i) \mid Pr(x = i) > 0 \} \).

Example: open game in Chinese dark chess.
  - For the first ply, \( N = 14 \times 32 \).
    - Using symmetry, we can reduce it to 7*8.
  - We now consider the chance node of flipping the piece at the cell a1.
    - \( N = 14 \).
    - Assume \( x = 1 \) means a black King is revealed and \( x = 8 \) means a red King is revealed.
    - Then \( \text{score}(1) = \text{score}(8) \) since the first player owns the revealed king no matter its color is.
    - \( Pr(x = 1) = Pr(x = 8) = 1/14 \).
Bounds in a chance node

- Assume the various possibilities of a chance node is evaluated one by one in the order that at the end of phase $i$, the $i$th choice is evaluated.
  - Assume $v_{\text{min}} \leq \text{score}(i) \leq v_{\text{max}}$.

- What are the lower and upper bounds, namely $m_i$ and $M_i$, of the expected value of the chance node immediately after the end of phase $i$?
  - $i = 0$.
    - $m_0 = v_{\text{min}}$
    - $M_0 = v_{\text{max}}$
  - $i = 1$, we first compute $\text{score}(1)$, and then know
    - $m_1 \geq \text{score}(1) \cdot \Pr(x = 1) + v_{\text{min}} \cdot (1 - \Pr(x = 1))$, and
    - $M_1 \leq \text{score}(1) \cdot \Pr(x = 1) + v_{\text{max}} \cdot (1 - \Pr(x = 1))$.
  - \ldots
  - $i = i^*$, we have computed $\text{score}(1), \ldots, \text{score}(i^*)$, and then know
    - $m_{i^*} \geq \sum_{i=1}^{i^*} \text{score}(i) \cdot \Pr(x = i) + v_{\text{min}} \cdot (1 - \sum_{i=1}^{i^*} \Pr(x = i))$, and
    - $M_{i^*} \leq \sum_{i=1}^{i^*} \text{score}(i) \cdot \Pr(x = i) + v_{\text{max}} \cdot (1 - \sum_{i=1}^{i^*} \Pr(x = i))$. 
Changes of bounds: uniform case (1/2)

- Assume the search window entering a chance node with $N = c$ choices is $[\alpha, \beta]$.
  - For simplicity, let’s assume $Pr_i = \frac{1}{c}$, for all $i$, and the evaluated value of the $i$th choice is $v_i$.

- The value of a chance node after the first $i$ choices are explored can be expressed as
  - an expected value $E_i = \frac{\text{vsum}_i}{i}$;
    - $\text{vsum}_i = \sum_{j=1}^{i} v_j$
    - This value is returned only when all choices are explored.
    $\Rightarrow$ The expected value of an un-explored child shouldn’t be $\frac{v_{\min} + v_{\max}}{2}$.
  - a range of possible values $[m_i, M_i]$.
    - $m_i = \frac{\sum_{j=1}^{i} v_j + v_{\min} \cdot (c - i)}{c}$
    - $M_i = \frac{\sum_{j=1}^{i} v_j + v_{\max} \cdot (c - i)}{c}$

- Invariants:
  - $E_i \in [m_i, M_i]$
  - $E_N = m_N = M_N$
Changes of bounds: uniform case (2/2)

- Let $m_i$ and $M_i$ be the current lower and upper bounds, respectively, of the expected value of this chance node immediately after the evaluation of the $i$th node.
  
  \[ m_i = \left( \sum_{j=1}^{i-1} v_j + v_i + v_{\min} \cdot (c - i) \right) / c \]
  
  \[ M_i = \left( \sum_{j=1}^{i-1} v_j + v_i + v_{\max} \cdot (c - i) \right) / c \]

- How to incrementally update $m_i$ and $M_i$:
  
  \[ m_0 = v_{\min} \]
  
  \[ M_0 = v_{\max} \]
  
  \[ m_i = m_{i-1} + (v_i - v_{\min}) / c \]
  
  \[ M_i = M_{i-1} + (v_i - v_{\max}) / c \]

- The current search window is $[\alpha, \beta]$.
  
  - No more searching is needed when
    
    \[ m_i \geq \beta, \text{ chance node cut off I}; \]
    
    \Rightarrow The lower bound found so far is good enough.
    
    \Rightarrow Similar to a beta cutoff.
    
    \Rightarrow The returned value is $m_i$.
    
    \[ M_i \leq \alpha, \text{ chance node cut off II}. \]
    
    \Rightarrow The upper bound found so far is bad enough.
    
    \Rightarrow Similar to an alpha cutoff.
    
    \Rightarrow The returned value is $M_i$. 
Chance node cut off

- **When** $m_i \geq beta$, **chance node cut off I**,
  - which means \((\sum_{j=1}^{i-1} v_j + v_i + v_{min} \cdot (c - i))/c \geq beta\)
  - \(\Rightarrow v_i \geq B_{i-1} = c \cdot beta - (\sum_{j=1}^{i-1} v_j - v_{min} \cdot (c - i))\)

- **When** $M_i \leq alpha$, **chance node cut off II**,
  - which means \((\sum_{j=1}^{i-1} v_j + v_i + v_{max} \cdot (c - i))/c \leq alpha\)
  - \(\Rightarrow v_i \leq A_{i-1} = c \cdot alpha - (\sum_{j=1}^{i-1} v_j - v_{max} \cdot (c - i))\)

- Hence set the window for searching the $i$th choice to be $[A_{i-1}, B_{i-1}]$ which means no further search is needed if the result is not within this window.

- **How to incrementally update** $A_i$ and $B_i$?
  - $A_0 = c \cdot (alpha - v_{max}) + v_{max}$
  - $B_0 = c \cdot (beta - v_{min}) + v_{min}$
  - $A_i = A_{i-1} + v_{max} - v_i$
  - $B_i = B_{i-1} + v_{min} - v_i$
Algorithm: Chance\_Search

- Algorithm $F3.1'(\text{position } p, \text{ value } \alpha, \text{ value } \beta)$
  
  // max node
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$, then return $f(p)$
  - else begin
    - $m := -\infty$
    - for $i := 1$ to $b$ do
      - begin
        - if $p_i$ is to play a chance node $n$
          then $t := Star1\_F3.1'(p_i, n, \max\{\alpha, m\}, b)$
        - else $t := G3.1'(p_i, \max\{\alpha, m\}, \beta)$
        - if $t > m$ then $m := t$
        - if $m \geq \beta$ then return($m$) // beta cut off
      - end
    - end
  - end;
  - return $m$
Algorithm: Chance_Search

**Algorithm Star1_F3.1′(position p, node n, value alpha, value beta)**

- // a chance node n with equal probability choices \( k_1, \ldots, k_c \)
- determine the possible values of the chance node n to be \( k_1, \ldots, k_c \)
- \( A_0 = c \cdot (alpha - v_{max}) + v_{max}, \quad B_0 = c \cdot (beta - v_{min}) + v_{min}; \)
- \( m_0 = v_{min}, \quad M_0 = v_{max} \) // current lower and upper bounds
- \( vsum = 0; \) // current sum of expected values
- for \( i = 1 \) to \( c \) do
  begin
    ▶ let \( p_i \) be the position of assigning \( k_i \) to n in p;
    ▶ \( t := G3.1'(p_i,\max\{A_{i-1},v_{min}\},\min\{B_{i-1},v_{max}\}) \)
    ▶ \( m_i = m_{i-1} + (t - v_{min})/c, \quad M_i = M_{i-1} + (t - v_{max})/c; \)
    ▶ if \( t \geq B_{i-1} \) then return \( m_i; \) // failed high, chance node cut off I
    ▶ if \( t \leq A_{i-1} \) then return \( M_i; \) // failed low, chance node cut off II
    ▶ \( vsum += t; \)
    ▶ \( A_i = A_{i-1} + v_{max} - t, \quad B_i = B_{i-1} + v_{min} - t; \)
  end
- return \( vsum/c; \)
Example: Chinese dark chess

- **Assumption:**
  - The range of the scores of Chinese dark chess is $[-10, 10]$ inclusive, $alpha = -10 \text{ and } beta = 10$.
  - $N = 7$.
  - $Pr(x = i) = 1/N = 1/7$.

- **Calculation:**
  - $i = 0$,
    - $m_0 = -10$.
    - $M_0 = 10$.
  - $i = 1 \text{ and if } score(1) = -2$, then
    - $m_1 = -2 * 1/7 + -10 * 6/7 = -62/7 \approx -8.86$.
    - $M_1 = -2 * 1/7 + 10 * 6/7 = 58/7 \approx 8.26$.
  - $i = 1 \text{ and if } score(1) = 3$, then
    - $m_1 = 3 * 1/7 + -10 * 6/7 = -57/7 \approx -8.14$.
    - $M_1 = 3 * 1/7 + 10 * 6/7 = 63/7 = 9$. 
General case

- Assume the $i$th choice happens with a chance $w_i/c$ where $c = \sum_{i=1}^{N} w_i$ and $N$ is the total number of choices.

  - $m_0 = v_{\text{min}}$
  - $M_0 = v_{\text{max}}$

  - $m_i = (\sum_{j=1}^{i-1} w_j \cdot v_j + w_i \cdot v_i + v_{\text{min}} \cdot (c - \sum_{j=1}^{i} w_j))/c$
    - $m_i = m_{i-1} + (w_i/c) \cdot (v_i - v_{\text{min}})$

  - $M_i = (\sum_{j=1}^{i-1} w_j \cdot v_j + w_i \cdot v_i + v_{\text{max}} \cdot (c - \sum_{j=1}^{i} w_j))/c$
    - $M_i = M_{i-1} + (w_i/c) \cdot (v_i - v_{\text{max}})$

  - $A_0 = (c/w_1) \cdot (\alpha - v_{\text{max}}) + v_{\text{max}}$
  - $B_0 = (c/w_1) \cdot (\beta - v_{\text{min}}) + v_{\text{min}}$

  - $A_{i-1} = (c \cdot \alpha - (\sum_{j=1}^{i-1} w_j \cdot v_j - v_{\text{max}} \cdot (c - \sum_{j=1}^{i} w_j))/w_i$
    - $A_i = (w_i/w_{i+1}) \cdot (A_{i-1} - v_i) + v_{\text{max}}$

  - $B_{i-1} = (c \cdot \beta - (\sum_{j=1}^{i-1} w_j \cdot v_j - v_{\text{min}} \cdot (c - \sum_{j=1}^{i} w_j))/w_i$
    - $B_i = (w_i/w_{i+1}) \cdot (B_{i-1} - v_i) + v_{\text{min}}$
Comments

- We illustrate the ideas using a fail soft version of the alpha-beta algorithm.
  - Original and fail hard version have a simpler logic in maintaining the search interval.
  - The semantic of comparing an exact returning value with an expected returning value is something that needs careful thinking.
  - May want to pick a chance node with a lower expected value but having a hope of winning, not one with a slightly higher expected value but having no hope of winning when you are in disadvantageous.
  - May want to pick a chance node with a lower expected value but having no chance of losing, not one with a slightly higher expected value but having a chance of losing when you are in advantage.

- Need to revise algorithms carefully when dealing with the original, fail hard or NegaScout version.
  - What does it mean to combine bounds from a fail hard version?

- Exist other improvements by considering better move orderings involving chance nodes.
How to use these bounds

- The lower and upper bounds of the expected score can be used to do alpha-beta pruning.
  - Nicely fit into the alpha-beta search algorithm.

- Can do better by not searching the DFS order.
  - It is not necessary to search completely the subtree of \( x = 1 \) first, and then start to look at the subtree of \( x = 2 \).
  - Assume it is a MAX chance node, e.g., the opponent takes a flip.
    - Knowing some value \( v'_1 \) of a subtree for \( x = 1 \) gives an upper bound, i.e., \( \text{score}(1) \geq v'_1 \).
    - Knowing some value \( v'_2 \) of a subtree for \( x = 2 \) gives another upper bound, i.e., \( \text{score}(2) \geq v'_2 \).
    - These bounds can be used to make the search window further narrower.

- For Monte-Carlo based algorithm, we need to use a sparse sampling algorithm to efficiently estimate the expected value of a chance node [Kearn et al 2002].
Proof number search

- 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 descendent node if its value is 1, then the value of $u$ is 1.
- A **most disproving node** for a node $u$: a descendent 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.
- The definition implies a bottom-up ordering.
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 value\( (u) \) is unknown, then disproof\( (u) \) is cost of evaluating \( u \).
  - If value\( (u) \) is 1, then disproof\( (u) = \infty \).
  - If value\( (u) \) is 0, then 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

proof number, disproof number

proof number, disproof number
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 $\min\{\text{proof}(\text{root}), \text{disproof}(\text{root})\}$.
  - 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 $\text{proof}(\text{root}) = 0$ or $\text{disproof}(\text{root}) = 0$, then we are done, otherwise
    - $\text{proof}(\text{root}) \leq \text{disproof}(\text{root})$: we try to prove it.
    - $\text{proof}(\text{root}) > \text{disproof}(\text{root})$: we try to disprove it.

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

```
  a
 /\  
 b  c
 / \
 d  e  f  g  h
```

18 ? ? 10 ?
Illustration

\[
\begin{align*}
 &a \\
 &\quad b \\
 &\quad\quad d \quad e \\
 &\quad\quad v \leq 18? \quad \quad \quad \quad v \leq 18? \\
 &\quad\quad 18 \quad ? \\
 &\quad\quad g \quad h \\
 &\quad\quad 10 \quad ?
\end{align*}
\]
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,
    \[
    \text{disproof}_v(u) = \min_{i=1}^{i=b} \text{disproof}_v(u_i).
    \]
Revised PN-search($v$): algorithm

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

- $u \leftarrow \text{root}; \{ \ast \text{ find the leaf to prove or disprove } \ast \} $
  - if we try to prove, then
    - while $u$ is not a leaf do
      - if $u$ is a MAX node, then
        - $u \leftarrow$ leftmost child of $u$ with the smallest non-zero $\text{proof}_v$ number;
      - if current is a MIN node, then
        - $u \leftarrow$ leftmost child of $u$ with a non-zero $\text{proof}_v$ number;
  
  - if we try to disprove, then
    - while $u$ is not a leaf do
      - if $u$ is a MAX node, then
        - $u \leftarrow$ leftmost child of $u$ with a non-zero $\text{disproof}_v$ number;
      - if current is a MIN node, then
        - $u \leftarrow$ leftmost child of $u$ with the smallest non-zero $\text{disproof}_v$ 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 $\leftarrow \lfloor (\text{low} + \text{high})/2 \rfloor$
  - PN-search(mid)
  - if it is disproved, then high $\leftarrow$ mid $-$ 1
  - else if it is proved, then low $\leftarrow$ mid
Comments

- Can be used to construct opening books.
- Appear 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.
  - Performances of most previous algorithms depend heavily on whether good move orderings can be found.
- Searching the “easiest” branch may not give you the best performance.
  - Performance depends on the value of each internal node.
- 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 (1/2)


References and further readings (2/2)

- Bruce W. Ballard The *-minimax search procedure for trees containing chance nodes Artificial Intelligence, Volume 21, Issue 3, September 1983, Pages 327-350
- Kearns, Michael; Mansour, Yishay; Ng, Andrew Y. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. Machine Learning, 2002, 49.2-3: 193-208.