Theory of Computer Games: Selected Advanced Topics

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

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

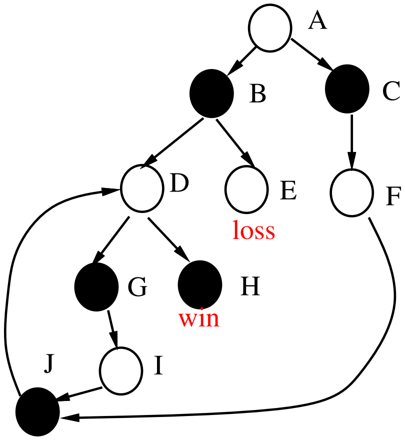
Abstract

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

Graph history interaction problem

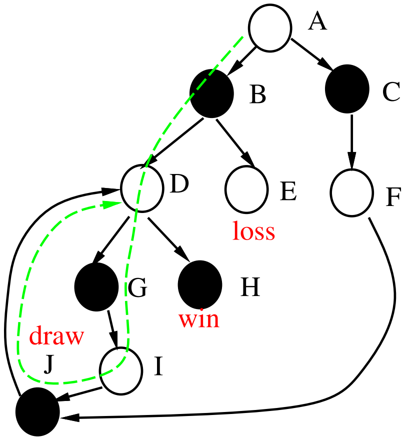
- The graph history interaction (GHI) problem [Campbell 1985]:
  - In a game graph, a position can be visited by more than one paths from a starting position.
  - The value of the position depends on the path visiting it.
    - ▷ *It can be win, loss or draw for Chinese chess.*
    - ▷ *It can only be 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] [Kishimoto and Müller '04].

GHI: when loop draws



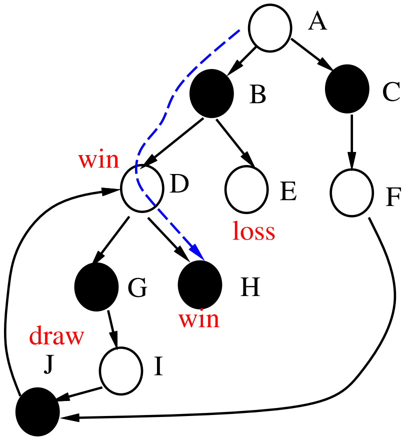
- **Assume if the game falls into a loop, then it is a draw.**

GHI: when loop draws



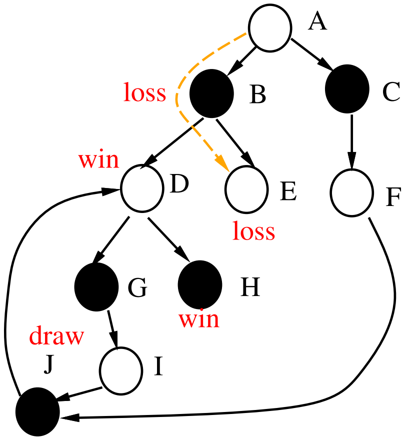
- **Assume if the game falls into a loop, then it is a draw.**
- $A \to B \to D \to G \to I \to J \to D$ **is draw by rules of repetition.**
  - ▷ ***Memorized*** J ***as a draw position.***

GHI: when loop draws



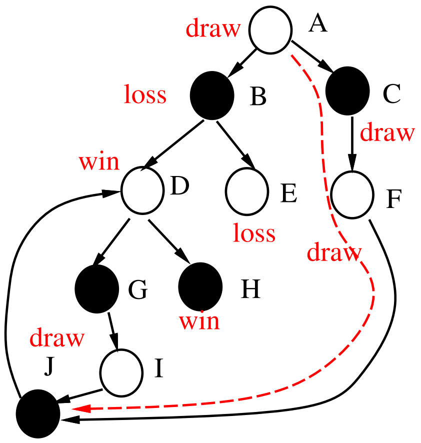
- **Assume if the game falls into a loop, then it is a draw.**
- $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ **is draw by rules of repetition.**
  - ▷ ***Memorized J as a draw position.***
- $A \rightarrow B \rightarrow D \rightarrow H$ **is a win. Hence D is win.**

GHI: when loop draws



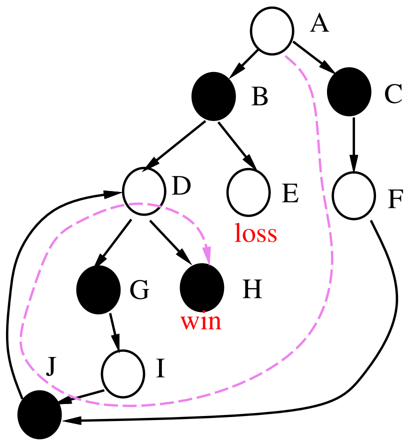
- **Assume if the game falls into a loop, then it is a draw.**
- $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ **is draw by rules of repetition.**
    - ▷ ***Memorized J as a draw position.***
- $A \rightarrow B \rightarrow D \rightarrow H$ **is a win. Hence** D **is win.**
- $A \rightarrow B \rightarrow E$ **is a loss. Hence** B **is loss.**

GHI: when loop draws



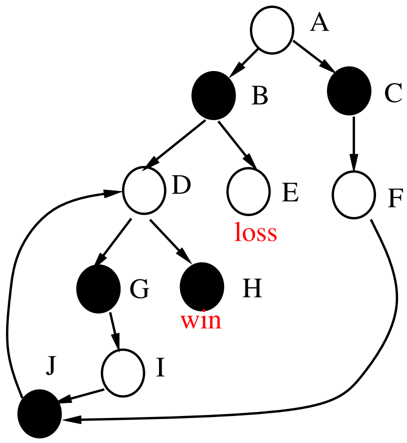
- **Assume if the game falls into a loop, then it is a draw.**
- $A \to B \to D \to G \to I \to J \to D$ **is draw by rules of repetition.**
  - ▷ *Memorized J as a draw position.*
- $A \to B \to D \to H$ **is a win. Hence** D **is win.**
- $A \to B \to E$ **is a loss. Hence** B **is loss.**
- $A \to C \to F \to J$ **is draw because** J **is recorded as draw.**
- A **is draw because one child is loss and the other chile is draw.**

GHI: when loop draws



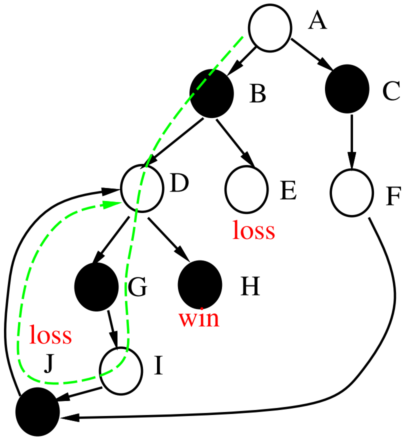
- Assume if the game falls into a loop, then it is a draw.
- $A \to B \to D \to G \to I \to J \to D$ is draw by rules of repetition.
  - ▷ *Memorized J as a draw position.*
- $A \to B \to D \to H$ is a win. Hence D is win.
- $A \to B \to E$ is a loss. Hence B is loss.
- $A \to C \to F \to J$ is draw because J is recorded as draw.
- A is draw because one child is loss and the other chile is draw.
- However, $A \to C \to F \to J \to D \to H$ is a win (for the root).

GHI: when loop wins



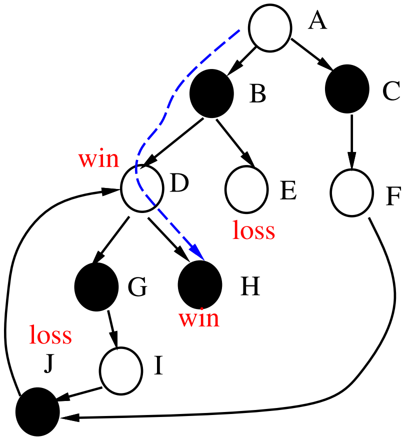
- **Assume the one causes loops wins the game.**

GHI: when loop wins



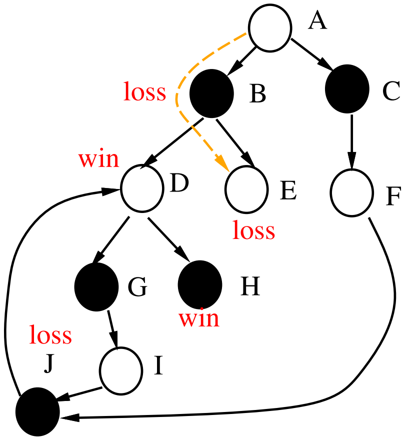
- **Assume the one causes loops wins the game.**
- $A \to B \to D \to G \to I \to J \to D$ **is loss because of rules of repetition.**
    - ▷ ***Memorized J as a loss position (for the root).***

GHI: when loop wins



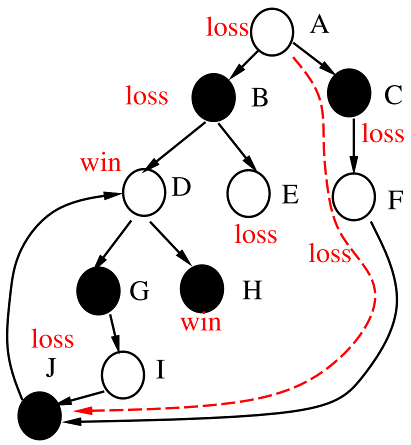
- **Assume the one causes loops wins the game.**
- $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ **is loss because of rules of repetition.**
    - ▷ *Memorized J as a loss position (for the root).*
- $A \rightarrow B \rightarrow D \rightarrow H$ **is a win. Hence D is win.**

GHI: when loop wins



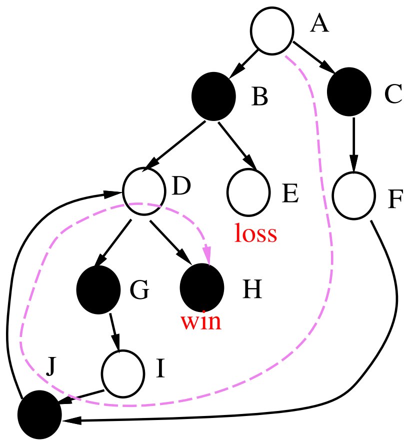
- **Assume the one causes loops wins the game.**
- $A \to B \to D \to G \to I \to J \to D$ **is loss because of rules of repetition.**
    - ▷ *Memorized J as a loss position (for the root).*
- $A \to B \to D \to H$ **is a win. Hence** D **is win.**
- $A \to B \to E$ **is a loss. Hence** B **is loss.**

GHI: when loop wins



- **Assume the one causes loops wins 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 (for the root).*
- $A \to B \to D \to H$ **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.**

GHI: when loop wins



- **Assume the one causes loops wins 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 (for the root).***
- $A \rightarrow B \rightarrow D \rightarrow H$ **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.**
- **However,** $A \rightarrow C \rightarrow F \rightarrow J \rightarrow D \rightarrow H$ **is a win (for the root).**

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 being draw 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.
    - Maybe we can store some forms of hash code to verify it.
- 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.***

Other usage of the opponent model

- **Depend on strength of your opponent, decide whether to force an easy draw or not.**
    - **This is called the contempt factor.**
- **Example in CDC:**
    - **It is easy to chase the king of your opponent using your pawn.**
    - **Drawing a weaker opponent is a waste.**
    - **Drawing a stronger opponent is a gain.**
- **It is feasible to use a learning model to "guess" the level of your opponent as the game goes and then adapt to its model in CDC [Chang et al 2021].**

Opponent models – comments

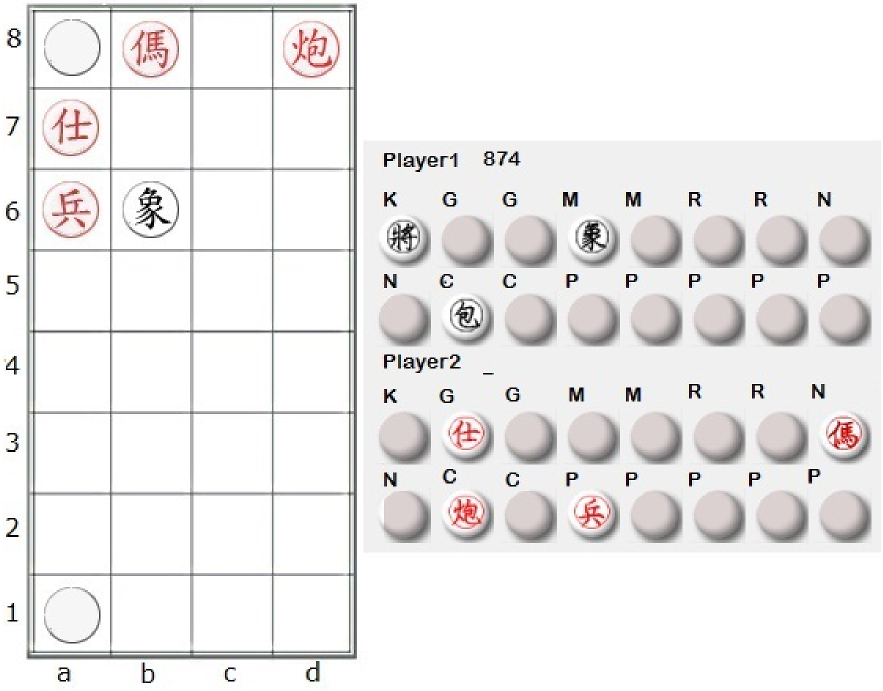
- **Comments:**
  - **Need to know your opponent's model precisely or to have some knowledge about your opponent.**
  - **How to learn the opponent model on-line or off-line?**
  - **When there are more than 2 possible opponent strategies, use a probability model (PrOM search) to form a strategy.**
- **Remark: A common misconception is if your opponent uses a worse strategy f_3 than the one, namely f_2, used in your model, then he may get advantage.**
  - **This is impossible if f_2 is truly better than f_3.**
  - **If f_1 can beat f_2, then f_1 can sure beat f_3.**

Search with chance nodes

- **Many stochastic games have nodes whose outcome cannot be decided ahead of time in the game tree.**
  - **A priori chance node: you make a decision first and then followed by a random toss.**
    - ▷ *EinStein Wrfelt Nicht (EWN): you make a random toss to decide what pieces that you can move, and then you make a move.*
  - **A posteriori chance node: a random toss is made first and then you make a decision.**
    - ▷ *Chinese dark chess: you pick a dark piece to flip, and then the piece is revealed decided by a random toss*
- **Example: Chinese dark chess (CDC)**
  - **Two-player, zero sum**
  - **Complete information**
  - **Perfect information**
  - **Stochastic**
  - **There is a chance node during searching [Ballard 1983].**
- **Previous work**
  - **Alpha-beta based [Ballard 1983]**
  - **Monte-Carlo based [Lancoto et al 2013]**

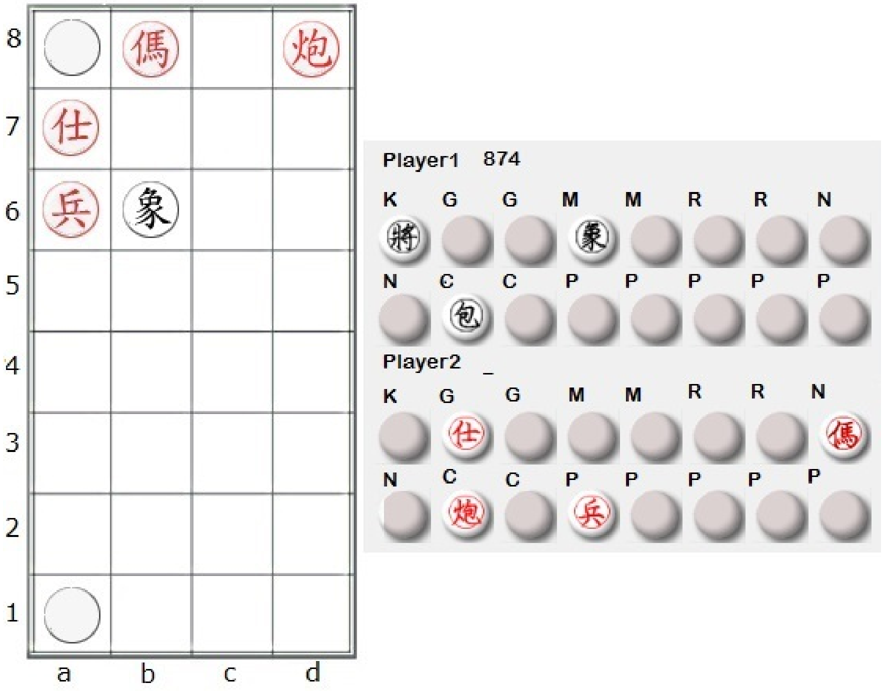
Example (1/4)

- **It's BLACK turn and BLACK has 6 different possible legal moves which includes the four different moving made by its elephant and the two flipping moves at a1 or a8.**
  - **It is difficult for BLACK to secure a win by moving its elephant along any of the 3 possible directions, namely up, right or left, or by capturing the RED pawn at the left hand side.**



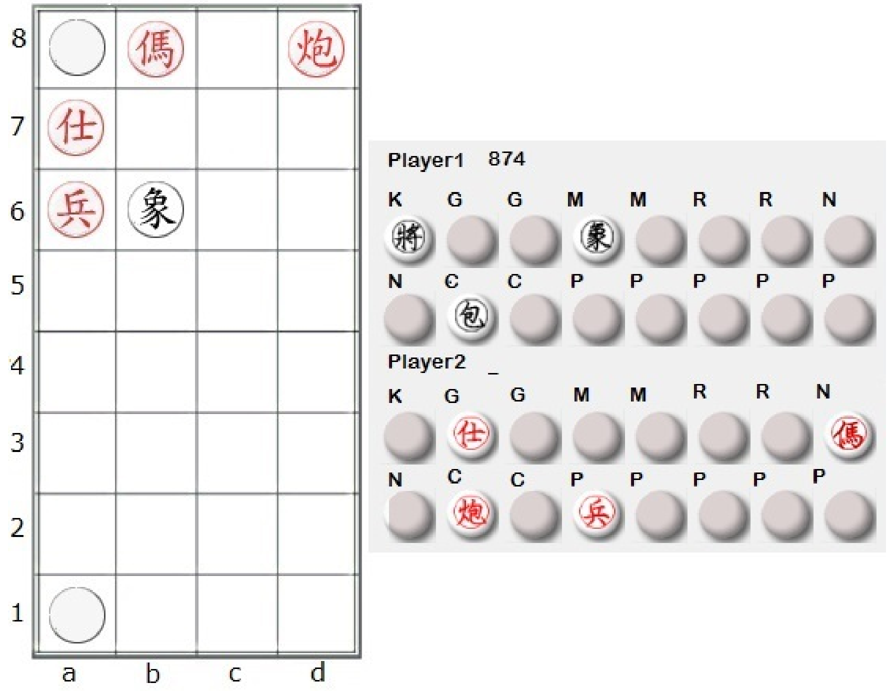
Example (2/4)

- **If BLACK flips a1, then there are 2 possible cases.**
  - **If a1 is BLACK cannon, then it is difficult for RED to win.**
    - ▷ *RED guard is in danger.*
  - **If a1 is BLACK king, then it is difficult for BLACK to lose.**
    - ▷ *BLACK king can go up through the right.*



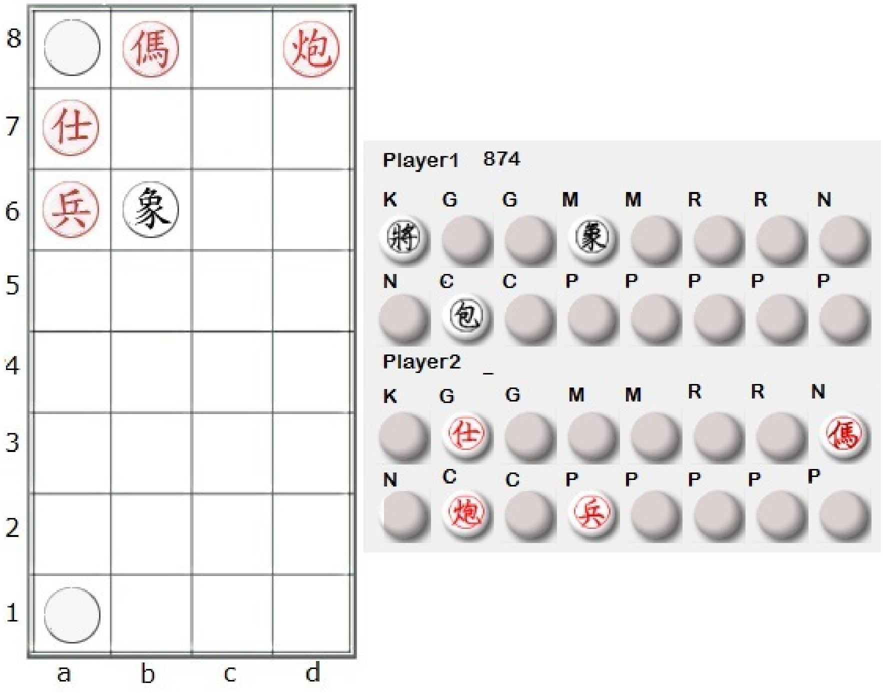
Example (3/4)

- **If BLACK flips a8, then there are 2 possible cases.**
  - **If a8 is BLACK cannon, then it is easy for RED to win.**
    - ▷ *RED cannon captures it immediately.*
  - **If a8 is BLACK king, then it is also easy for RED to win.**
    - ▷ *RED cannon captures it immediately.*



Example (4/4)

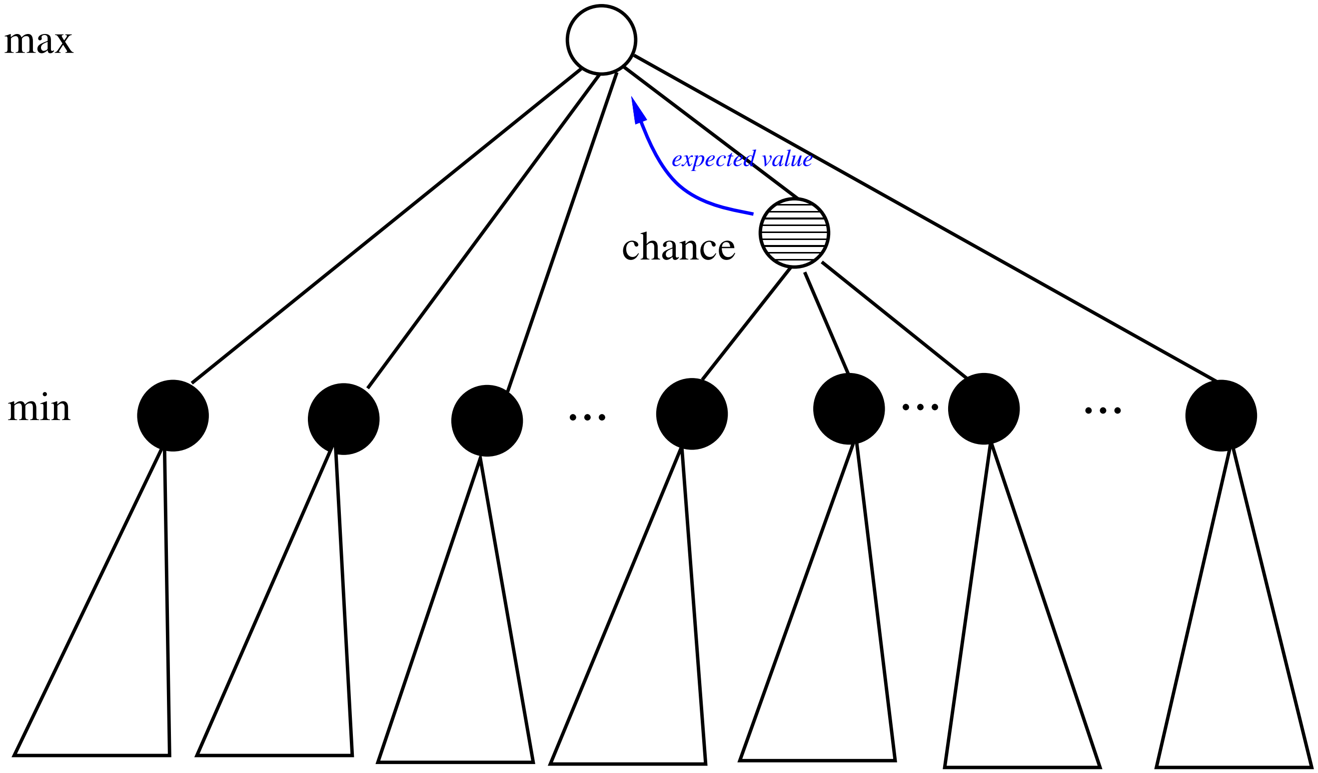
- **Conclusion:**
  - It is vary bad for BLACK to flip a8.
  - It is bad for BLACK to move its elephant.
  - It is better for BLACK to flip a1.



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 $score(i)$.**
  - **The expected value $E = \sum_{i=1}^{N} score(i) * Pr(x = i)$.**
  - **The minimum value is $m = \min_{i=1}^{N}\{score(i) \mid Pr(x = i) > 0\}$.**
  - **The maximum value is $M = \max_{i=1}^{N}\{score(i) \mid Pr(x = i) > 0\}$.**
- **Example: open game in Chinese dark chess.**
  - **For the first ply, $N = 14 * 32$.**
    - ▷ ***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 $score(1) = score(8)$ since the first player owns the revealed king no matter its color is.***
    - ▷ $Pr(x = 1) = Pr(x = 8) = 1/14$.

Illustration



Algorithm: Chance_Search (MAX node)

- **Algorithm** $F3.0'$(**position** p, **value** $alpha$, **value** $beta$, **integer** $depth$)

```
// max node
determine the successor positions $p_1, \ldots, p_b$
if $b = 0$ // a terminal node
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
then return $f(p)$ else begin
    ▷ $m := -\infty$
    ▷ for $i := 1$ to $b$ do
    ▷ begin
    ▷     if $p_i$ is to play a chance node $x$
              then $t := Star0\_F3.0'(p_i, x, max\{alpha, m\}, beta, depth - 1)$
    ▷     else $t := G3.0'(p_i, max\{alpha, m\}, beta, depth - 1)$
    ▷     if $t > m$ then $m := t$
    ▷     if $m \geq beta$ then return($m$) // beta cut off
    ▷ end
end;
return $m$
```

Algorithm: Chance_Search (MIN node)

- **Algorithm** $G3.0'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
    - // min node
    - determine the successor positions $p_1, \ldots, p_b$
    - if $b = 0$ // a terminal node
      or $depth = 0$ // remaining depth to search
      or time is running up // from timing control
      or some other constraints are met // add knowledge here
    - then return $f(p)$ else begin

```
▷ $m := \infty$
▷ for $i := 1$ to $b$ do
▷ begin
▷     if $p_i$ is to play a chance node $x$
          then $t := Star0\_G3.0'(p_i, x, alpha, min\{beta, m\}, depth - 1)$
▷     else $t := F3.0'(p_i, alpha, min\{beta, m\}, depth - 1)$
▷     if $t < m$ then $m := t$
▷     if $m \leq alpha$ then return$(m)$ // alpha cut off
▷ end
```

    - end;
    - return m

Algorithm: $Star0$, uniform case (MAX)

- version when all choices have equal probabilities
- max node
- Algorithm $Star0\_EQU\_F3.0'$(position p, node x, value $alpha$, value $beta$, integer $depth$)

```
    // a chance node $x$ with $c$ equal probability choices $k_1$, ..., $k_c$
    // exhaustive search all possibilities and return the expected value
    determine the possible values of the chance node $x$ to be $k_1, \ldots, k_c$
    $vsum = 0$; // current sum of expected value
    for $i = 1$ to $c$ do
    begin
        let $p_i$ be the position of assigning $k_i$ to $x$ in $p$;
        $vsum$ += $G3.0'(p_i, -\infty, +\infty, depth)$;
    end
return $vsum/c$; // return the expected score
```

Algorithm: $Star0$, uniform case (MIN)

- version when all choices have equal probabilities
- min node
- Algorithm $Star0\_EQU\_G3.0'$(position p, node x, value $alpha$, value $beta$, integer $depth$)

```
    • // a chance node $x$ with $c$ equal probability choices $k_1$, ..., $k_c$
    • // exhaustive search all possibilities and return the expected value
    • determine the possible values of the chance node $x$ to be $k_1, \ldots, k_c$
    • $vsum = 0$; // current sum of expected value
    • for $i = 1$ to $c$ do
    • begin
        ▷ let $p_i$ be the position of assigning $k_i$ to $x$ in $p$;
        ▷ $vsum$ += $F3.0'$($p_i$,$-\infty$, $+\infty$,$depth$);
    • end
```

- return $vsum/c$; // return the expected score

Star0: note

- $depth$ **stays the same since we are unwrapping a chance node.**
- **The search window from normal alpha-beta pruning cannot be applied in a chance node search since we are looking at the average of the outcome.**
    - **It is okay for one choice to have a very large or small value because it may be evened out by values from other choices.**

With a probability distribution: MAX node

- **MAX node**
- **Algorithm** $Star0\_F3.0'$**(position** p**, node** x**, value** $alpha$**, value** $beta$**,integer** $depth$**)**

```
// a chance node $x$ with $c$ choices $k_1$, ..., $k_c$
// the $i$th choice happens with the probability Pr$_i$
// exhaustive search all possibilities and return the expected value
determine the possible values of the chance node $x$ to be $k_1, \ldots, k_c$
$vexp = 0$; // current sum of expected value
for $i = 1$ to $c$ do
begin
    let $p_i$ be the position of assigning $k_i$ to $x$ in $p$;
    $vexp$ += $Pr_i * G3.0'(p_i, -\infty, +\infty, depth)$;
end
```

- **return** $vexp$; // return the expected score

With a probability distribution: MIN node

- **MIN node**
- **Algorithm** $Star0\_G3.0'$**(position** p**, node** x**, value** $alpha$**, value** $beta$**,integer** $depth$**)**

```
// a chance node $x$ with $c$ choices $k_1$, ..., $k_c$
// the $i$th choice happens with the probability Pr$_i$
// exhaustive search all possibilities and return the expected value
determine the possible values of the chance node $x$ to be $k_1, \ldots, k_c$
$vexp = 0$; // current sum of expected value
for $i = 1$ to $c$ do
begin
    let $p_i$ be the position of assigning $k_i$ to $x$ in $p$;
    $vexp$ += Pr$_i$ * $F3.0'(p_i, -\infty, +\infty, depth)$;
end
return $vexp$; // return the expected score
```

Ideas for improvements

- During a chance search, an exhaustive search method is used without any pruning.
- Ideas for further improvements
  - When some of the best possible cases turn out very bad results, we know lower/upper bounds of the final value.
  - When you are in advantage, search for a bad choice first.
    - ▷ *If the worst choice cannot is not too bad, then you can take this chance.*
  - When you are in disadvantage, search for a good choice first.
    - ▷ *If the best choice cannot is not good enough, then there is not need to take this chance.*
- Examples: the average of 2 drawings of a dice is similar to a position with 2 possible moves with scores in [1..6].
  - The first drawing is 5. Then bounds of the average:
    - ▷ *lower bound is 3*
    - ▷ *upper bound is 5.5.*
  - The first drawing is 1. Then bounds of the average:
    - ▷ *lower bound is 1*
    - ▷ *upper bound is 3.5.*

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 ith choice is evaluated.**
  - **Assume $v_{min} \leq score(i) \leq v_{max}$.**
- **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?**
  - $i = 0$.
    - ▷ $m_0 = v_{min}$
    - ▷ $M_0 = v_{max}$
  - $i = 1$, **we first compute $score(1)$, and then know**
    - ▷ $m_1 \geq score(1) * Pr(x = 1) + v_{min} * (1 - Pr(x = 1))$, ***and***
    - ▷ $M_1 \leq score(1) * Pr(x = 1) + v_{max} * (1 - Pr(x = 1))$.
  - $\cdots$
  - $i = i^*$, **we have computed $score(1), \ldots, score(i^*)$, and then know**
    - ▷ $m_{i^*} \geq \sum_{i=1}^{i^*} score(i) * Pr(x = i) + v_{min} * (1 - \sum_{i=1}^{i^*} Pr(x = i))$, ***and***
    - ▷ $M_{i^*} \leq \sum_{i=1}^{i^*} score(i) * Pr(x = i) + v_{max} * (1 - \sum_{i=1}^{i^*} Pr(x = i))$.

Changes of bounds: uniform case (1/2)

- For simplicity, let's assume $Pr(x = i) = \frac{1}{c}$.
- For all i, and the evaluated value of the ith choice is v_i.
- Assume the search window entering a chance node with $N = c$ choices is $(alpha, beta)$.
- The value of a chance node after the first i choices are explored can be expressed as
  - an expected value $E_i = vsum_i/i$;
    - ▷ $vsum_i = \sum_{j=1}^{i} v_j$
    - ▷ *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 $[m_i, M_i]$.
    - ▷ $m_i = (\sum_{j=1}^{i} v_j + v_{min} \cdot (c - i))/c$
    - ▷ $M_i = (\sum_{j=1}^{i} v_j + v_{max} \cdot (c - i))/c$
  - Invariants:
    - ▷ $E_i \in [m_i, M_i]$
    - ▷ $E_c = m_c = M_c$

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 ith node.**
  - $m_i = (\sum_{j=1}^{i-1} v_j + v_i + v_{min} \cdot (c - i))/c$
  - $M_i = (\sum_{j=1}^{i-1} v_j + v_i + v_{max} \cdot (c - i))/c$
- **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**
    - ▷ $m_i \geq beta$, ***chance node cut off I;***
      ⇒ The lower bound found so far is good enough.
      ⇒ Similar to a beta cut off.
      ⇒ The returned value is m_i.
    - ▷ $M_i \leq alpha$, ***chance node cut off II.***
      ⇒ The upper bound found so far is bad enough.
      ⇒ Similar to an alpha cut off.
      ⇒ The returned value is M_i.

Chance node cut off: uniform case (1/3)

- **The above two cut offs comes from each time a choice is completely searched.**
  - **When $m_i \geq beta$, chance node cut off I,**
    - ▷ ***which means*** $(\sum_{j=1}^{i-1} v_j + v_i + v_{min} \cdot (c-i))/c \geq beta$.
  - **When $M_i \leq alpha$, chance node cut off II,**
    - ▷ ***which means*** $(\sum_{j=1}^{i-1} v_j + v_i + v_{max} \cdot (c-i))/c \leq alpha$.
- **Further cut off can be obtained before when that choice is in searching.**
  - **Assume after searching the first $i-1$ choices, no chance node cut off happens.**
  - **Before searching the ith choice, we know that if v_i is large enough, then it will raise the lower bound of the chance node and it will have a chance of getting a chance node cut off I.**
  - **How large should v_i be for this to happen?**
    - ▷ ***chance node cut off I:*** $(\sum_{j=1}^{i-1} v_j + v_i + v_{min} \cdot (c-i))/c \geq beta$
    - ▷ $\Rightarrow v_i \geq B_{i-1} = c \cdot beta - (\sum_{j=1}^{i-1} v_j - v_{min} * (c-i))$
    - ▷ B_{i-1} ***is the threshold for cut off I to happen.***

Chance node cut off: uniform case (2/3)

- **Similarly,**
  - **Assume after searching the first $i-1$ choices, no chance node cut off happens.**
  - **Before searching the ith choice, we know that if v_i is small enough, then it will lower the upper bound of the chance node and it will have a chance of getting a chance node cut off II.**
  - **How small should v_i be for this to happen?**
    - ***chance node cut off II:*** $(\sum_{j=1}^{i-1} v_j + v_i + v_{max} \cdot (c-i))/c \leq alpha$
    - $\Rightarrow v_i \leq A_{i-1} = c \cdot alpha - (\sum_{j=1}^{i-1} v_j - v_{max} * (c-i))$
    - A_{i-1} ***is the threshold for cut off II to happen.***

Chance node cut off: uniform case (3/3)

- **Hence set the window for searching the ith choice to be (A_{i-1}, B_{i-1}) which means no further search is needed if the result is not within this window.**
  - (A_{i-1}, B_{i-1}) **is the window for searching the ith choice instead of using $(alpha, beta)$.**
- **How to incrementally update A_i and B_i?**
  - $A_0 = c \cdot (alpha - v_{max}) + v_{max}$
  - $B_0 = c \cdot (beta - v_{min}) + v_{min}$
  - $A_i = A_{i-1} + v_{max} - v_i$
  - $B_i = B_{i-1} + v_{min} - v_i$
- **Comment:**
  - **May want to use zero-window search to test first.**

Changes of bounds: non-uniform case (1/3)

- **Assume the search window entering a chance node with $N = c$ choices is $(alpha, beta)$.**
- **The ith choice happens with the probability $Pr(x = i) = Pr_i$.**
- **For all i, the evaluated value of the ith choice is v_i.**
- **The value of a chance node after the first i choices are explored can be expressed as**
  - **an expected value $E_i = vexp_i$;**
    - ▷ $vexp_i = \sum_{j=1}^{i} Pr_j * v_j$
    - ▷ *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 $[m_i, M_i]$.**
    - ▷ $m_i = vexp_i + \sum_{j=i+1}^{c} Pr_j * v_{min}$
    - ▷ $M_i = vexp_i + \sum_{j=i+1}^{c} Pr_j * v_{max}$
  - **Invariants:**
    - ▷ $E_i \in [m_i, M_i]$
    - ▷ $E_c = m_c = M_c$

Changes of bounds: non-uniform case (2/3)

- **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 ith node.**
  - $m_i = vexp_{i-1} + Pr_i * v_i + \sum_{j=i+1}^{c} Pr_j * v_{min}$
  - $M_i = vexp_{i-1} + Pr_i * v_i + \sum_{j=i+1}^{c} Pr_j * v_{max}$
- **How to incrementally update m_i and M_i:**
  - $m_0 = v_{min}$
  - $M_0 = v_{max}$
  - $$m_i = m_{i-1} + Pr_i * (v_i - v_{min}) \tag{1}$$
  - $$M_i = M_{i-1} + Pr_i * (v_i - v_{max}) \tag{2}$$

Changes of bounds: non-uniform case (3/3)

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

Chance node cut off: non-uniform case (1/2)

- **When** $m_i \geq beta$, **chance node cut off I,**
  - **which means** $vexp_{i-1} + Pr_i * v_i + \sum_{j=i+1}^{c} Pr_j * v_{min} \geq beta$
  - $\Rightarrow v_i \geq B_{i-1} = \frac{1}{Pr_i} \cdot (beta - (vexp_{i-1} + \sum_{j=i+1}^{c} Pr_j * v_{min}))$
- **When** $M_i \leq alpha$, **chance node cut off II,**
  - **which means** $vexp_{i-1} + Pr_i * v_i + \sum_{j=i+1}^{c} Pr_j * v_{max} \leq alpha$
  - $\Rightarrow v_i \leq A_{i-1} = \frac{1}{Pr_i} \cdot (alpha - (vexp_{i-1} + \sum_{j=i+1}^{c} Pr_j * v_{max}))$
- **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.**

Chance node cut off: non-uniform case (2/2)

- **How to incrementally update A_i and B_i?**
  - $$A_0 = \frac{1}{Pr_1} \cdot (alpha - v_{max} * \sum_{i=1}^{c} Pr_i) + v_{max} \tag{3}$$
  - $$B_0 = \frac{1}{Pr_1} \cdot (beta - v_{min} * \sum_{i=1}^{c} Pr_i) + v_{min} \tag{4}$$
  - $$A_i = \frac{1}{Pr_{i+1}} * (Pr_i * A_{i-1} + Pr_{i+1} * v_{max} - Pr_i * v_i) \tag{5}$$
  - $$B_i = \frac{1}{Pr_{i+1}} * (Pr_i * B_{i-1} + Pr_{i+1} * v_{min} - Pr_i * v_i) \tag{6}$$

Algorithm: Chance_Search

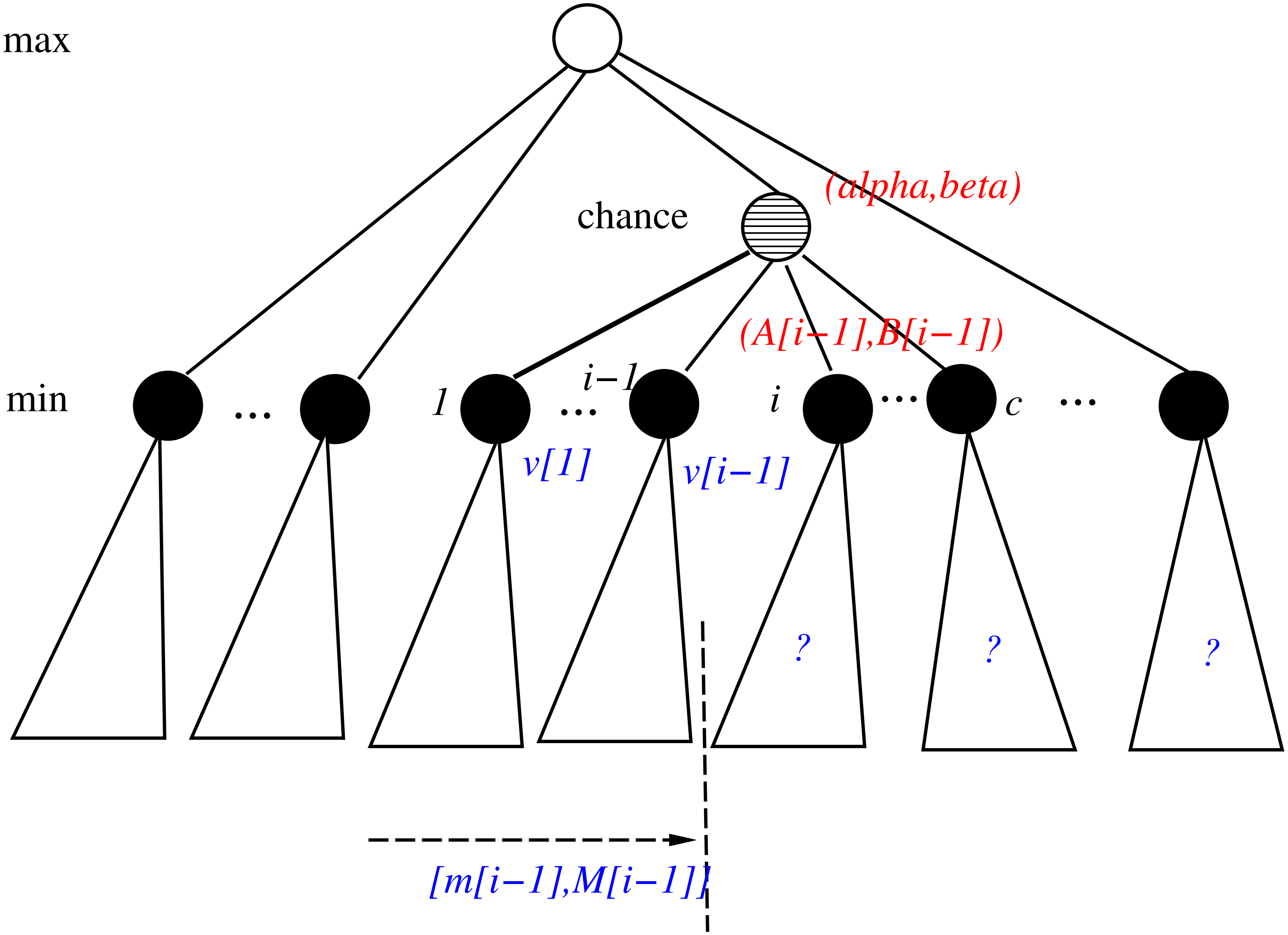
- **Algorithm** $F3.1'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**

```
• // max node
• determine the successor positions $p_1, \ldots, p_b$;
• if $b = 0$ // a terminal node
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
• then return $f(p)$; else begin
      ▷ $m := -\infty$;
      ▷ for $i := 1$ to $b$ do
      ▷ begin
      ▷     if $p_i$ is to play a chance node $x$
            then $t := Star1\_F3.1'(p_i, x, max\{alpha, m\}, beta, depth - 1)$;
      ▷     else $t := G3.1'(p_i, max\{alpha, m\}, beta, depth - 1)$;
      ▷     if $t > m$ then $m := t$;
      ▷     if $m \geq beta$ then return($m$); // beta cut off
      ▷ end;
• end;
• return $m$;
```

Star1: uniform case

```
Algorithm $Star1\_EQU\_F3.1'$(position $p$, node $x$, value $alpha$, value $beta$, integer $depth$)
    // a chance node $x$ with $c$ equal probability choices $k_1$, ..., $k_c$
    determine the possible values of the chance node $x$ to be $k_1, \ldots, k_c$
    $A_0 = c \cdot (alpha - v_{max}) + v_{max}$, $B_0 = c \cdot (beta - v_{min}) + v_{min}$;
    $m_0 = v_{min}$, $M_0 = v_{max}$ // 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 $x$ in $p$;
        $t := G3.1'(p_i, \max\{A_{i-1}, v_{min}\}, \min\{B_{i-1}, v_{max}\}, depth)$
        $m_i = m_{i-1} + (t - v_{min})/c$, $M_i = M_{i-1} + (t - v_{max})/c$;
        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
        $vsum$ += $t$;
        $A_i = A_{i-1} + v_{max} - t$, $B_i = B_{i-1} + v_{min} - t$;
    end
return $vsum/c$;
```

Illustration: Star1



Star1: non-uniform case

```
Algorithm $Star1\_F3.1'$(position $p$, node $x$, value $alpha$, value $beta$, integer $depth$)
    // a chance node $x$ with $c$ choices $k_1$, ..., $k_c$
    // the $i$th choice happens with the probability Pr$_i$
    determine the possible values of the chance node $x$ to be $k_1, \ldots, k_c$
    initialize $A_0$ and $B_0$ using formulas (3) and (4)
    $m_0 = v_{min}$, $M_0 = v_{max}$ // current lower and upper bounds
    $vexp = 0$; // current weighted sum of expected values
    for $i = 1$ to $c$ do
    begin
        let $Pr_i$ be the position of assigning $k_i$ to $x$ in $p$;
        $t$ := $G3.1'$($p_i$,max{$A_{i-1}$,$v_{min}$},min{$B_{i-1}$,$v_{max}$},$depth$)
        incrementally update $m_i$ and $M_i$ using formulas (1) and (2)
        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
        $vexp$ += $Pr_i * t$;
        incrementally update $A_i$ and $B_i$ using formulas (5) and (6)
    end
return $vexp$;
```

Example: Chinese dark chess

- **Assumption:**
  - **The range of the scores of Chinese dark chess is $[-10, 10]$ inclusive, $alpha = -10$ and $beta = 10$.**
  - $N = 7$.
  - $Pr(x = i) = 1/N = 1/7$.
- **Calculation:**
  - $i = 0$,
    - ▷ $m_0 = -10$.
    - ▷ $M_0 = 10$.
  - $i = 1$ **and if** $score(1) = -2$, **then**
    - ▷ $m_1 = -2 * 1/7 + -10 * 6/7 = -62/7 \simeq -8.86$.
    - ▷ $M_1 = -2 * 1/7 + 10 * 6/7 = 58/7 \simeq 8.26$.
  - $i = 1$ **and if** $score(1) = 3$, **then**
    - ▷ $m_1 = 3 * 1/7 + -10 * 6/7 = -57/7 \simeq -8.14$.
    - ▷ $M_1 = 3 * 1/7 + 10 * 6/7 = 63/7 = 9$.

General case

- **Assume the ith choice happens with a chance w_i/c where $c = \sum_{i=1}^{N} w_i$ and N is the total number of choices.**
  - $m_0 = v_{min}$
  - $M_0 = v_{max}$
  - $m_i = (\sum_{j=1}^{i-1} w_j \cdot v_j + w_i \cdot v_i + v_{min} \cdot (c - \sum_{j=1}^{i} w_j))/c$
    - ▷ $m_i = m_{i-1} + (w_i/c) \cdot (v_i - v_{min})$
  - $M_i = (\sum_{j=1}^{i-1} w_j \cdot v_j + w_i \cdot v_i + v_{max} \cdot (c - \sum_{j=1}^{i} w_j))/c$
    - ▷ $M_i = M_{i-1} + (w_i/c) \cdot (v_i - v_{max})$
  - $A_0 = (c/w_1) \cdot (alpha - v_{max}) + v_{max}$
  - $B_0 = (c/w_1) \cdot (beta - v_{min}) + v_{min}$
  - $A_{i-1} = (c \cdot alpha - (\sum_{j=1}^{i-1} w_j \cdot v_j - v_{max} \cdot (c - \sum_{j=1}^{i} w_j)))/w_i$
    - ▷ $A_i = (w_i/w_{i+1}) \cdot (A_{i-1} - v_i) + v_{max}$
  - $B_{i-1} = (c \cdot beta - (\sum_{j=1}^{i-1} w_j \cdot v_j - v_{min} \cdot (c - \sum_{j=1}^{i} w_j)))/w_i$
    - ▷ $B_i = (w_i/w_{i+1}) \cdot (B_{i-1} - v_i) + v_{min}$

The probability distribution

- **Assume a chance node x has c choices $k_1, \ldots, k_c$.**
- **The ith choice happens with the probability Pr_i and $\sum_{i=1}^{c} Pr_i = 1$.**
- **Special case 1, called uniform (EQU): $Pr_i = 1/c$.**
  - **All choices happen with a equal chance.**
  - **Example: EinStein Wrfelt Nicht (EWN) when all pieces are not captured.**
- **Special case 2, called GCD: $Pr_i = w_i/D$ where each w_i is an integer and $\sum_{i=1}^{c} w_i = D$.**
  - **example: Chinese dark chess.**
- **The above two special cases usually happen in game playing and can use the characteristics to do some optimization in number calculations.**

Algorithm: $Star0$, GCD case, MAX node

- **An GCD version for a MAX node.**
- **Algorithm $Star0\_GCD\_F3.0'$(position p, node x, value $alpha$, value $beta$, integer $depth$)**

```
    // a chance node $x$ with $c$ choices $k_1$, ..., $k_c$
    // whose occurrence probability are $w_1/D$, ..., $w_c/D$
    // and each $w_i$ is an integer
    // exhaustive search all possibilities and return the expected value
    determine the possible values of the chance node $x$ to be $k_1, \ldots, k_c$
    $vsum = 0$; // current sum of weight values
    for $i = 1$ to $c$ do
    begin
        let $p_i$ be the position of assigning $k_i$ to $x$ in $p$;
        $vsum$ += $w_i$ * $G3.0'(p_i, -\infty, +\infty, depth)$;
    end
```

- **return $vsum/D$; // return the expected score**

Algorithm: $Star0$, GCD case, MIN node

- **An GCD version for a MIN node.**
- **Algorithm** $Star0\_GCD\_G3.0'$**(position** p**, node** x**, value** $alpha$**, value** $beta$**, integer** $depth$**)**

```
// a chance node $x$ with $c$ choices $k_1$, …, $k_c$
// whose occurrence probability are $w_1/D$, …, $w_c/D$
// and each $w_i$ is an integer
// exhaustive search all possibilities and return the expected value
determine the possible values of the chance node $x$ to be $k_1, \ldots, k_c$
$vsum = 0$; // current sum of weight values
for $i = 1$ to $c$ do
begin
    ▷ let $p_i$ be the position of assigning $k_i$ to $x$ in $p$;
    ▷ $vsum$ += $w_i$ * $F3.0'(p_i, -\infty, +\infty, depth)$;
end
```

- **return** $vsum/D$; **// return the expected score**

Comments (1/2)

- 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 return 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.
  - Do not always pick one with a slightly larger expected value. Give the second one some chance to be selected.

Comments (2/2)

- **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?
- **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.
  - Not only we can terminate the searching of choices earlier, but also we can terminate the searching of a particular choice earlier.
- **Exist other improvements by searching choices of a chance node "in parallel".**

Implementation hints (1/2)

- **Fully unwrap a chance node takes more time than that of a non-chance node.**
  - **If you set your depth limit to d for a game without chance nodes, then the depth limit should be lower for that game when chance node is introduced.**
  - **Technically speaking, a chance node adds at least one level.**
    - ▷ *Depending on the number of choices you have compared to the number of non-chance children, you may need to reduce the search depth limit by at least 3 or 5, and maybe 7.*
    - ▷ *Estimate the complexity of a chance node by comparing the number of choices of a chance node and the number of non-chance-node moves.*
- **Without searching a chance node, it is easy to obtain not enough progress by just searching a long sequence of non-chance nodes.**
  - **In CDC, when there are only a limited number of revealed pieces, there is not much you can do by just moving around.**

Implementation hints (2/2)

- Practical considerations, for example in Chinese Dark Chess (CDC), are as follows.
    - You normally do not need to consider the consequence of flipping more than 2 dark pieces.
        - ▷ *Set a maximum number of chance node searching in any DFS search path.*
    - It makes little sense to consider ending a search with exploring a chance node.
        - ▷ *When depth limit left is less than 3 or 4, stop exploring chance nodes.*
    - It also makes little sense to consider the consequence of exploring 2 chance nodes back to back.
        - ▷ *Make sure two chance nodes in a DFS search path is separated by at least 3 or 4 non-chance nodes.*
    - It is rarely the case that a chance node exploration is the first ply to consider in move ordering unless it is recommended by a prior knowledge or no other non-chance-node moves exists.

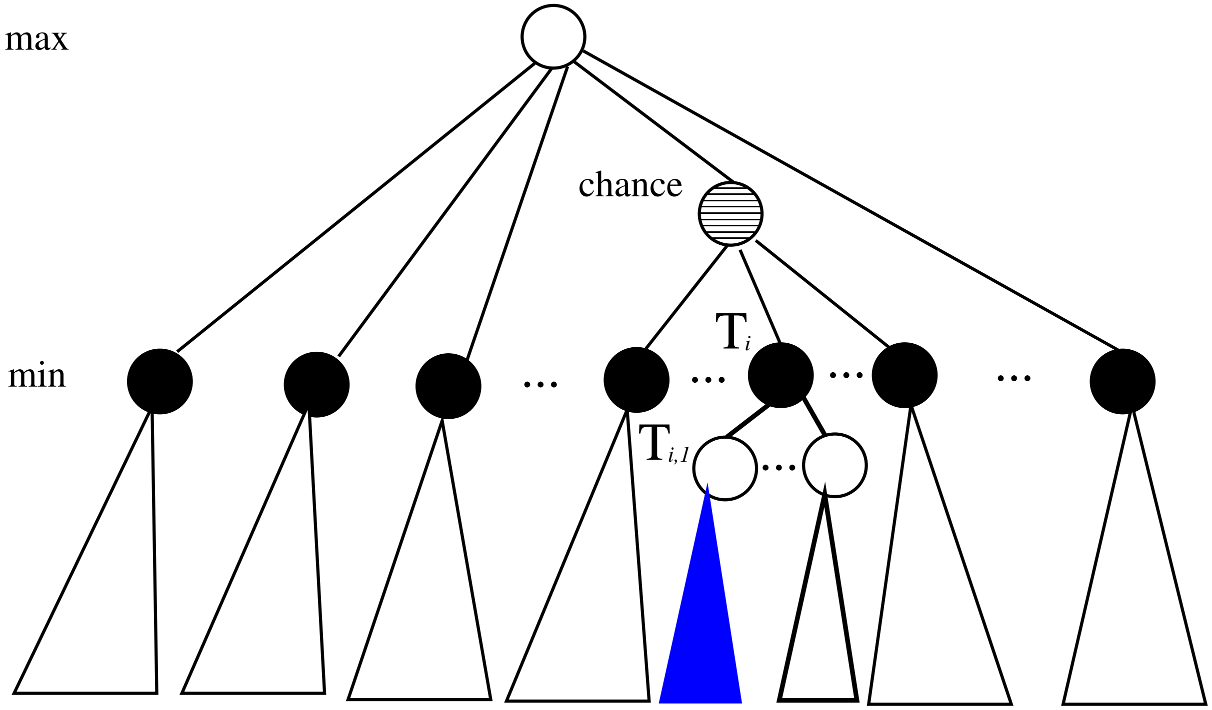
Ideas for furthermore improvements (1/2)

- 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 $x = 2$, ... etc.
  - partially search a subtree gives you some information about the possible range of this chance node.
- Assume p is a MAX chance node, e.g., root makes a flip.
  - T_i is the tree of p when for the ith choice, namely, with the root p_i which is a MIN node.
  - $T_{i,j}$ is the jth branch of T_i, namely, with the root $p_{i,j}$.
  - v_i is the evaluated value of T_i.
  - $v_{i,j}$ is the evaluated value of $T_{i,j}$.
- We have completely searched $T_{1,s}$ and obtained a value $v_{1,s}$.
  - Since p_i is a MIN node, $v_{1,s}$ is an upper bound of v_1 which is usually lower than the maximum possible value.
  - The upper bound of v_1 is thus lowered.
  - It is possible because of this probe, an alpha cut can be performed.
- The above process is called an exact probe.
  - We can first probe each T_i.
  - It is better to probe the worse possible branch of T_i first.

Ideas for furthermore improvements (2/2)

- **Assume p is a MIN chance node, e.g., the opponent makes a flip.**
  - T_i is the tree of p when for the ith choice, namely, with the root p_i which is a MAX node.
  - $T_{i,j}$ is the jth branch of T_i, namely, with the root $p_{i,j}$.
  - v_i is the evaluated value of T_i.
  - $v_{i,j}$ is the evaluated value of $T_{i,j}$.
- **We have completely searched $T_{1,s}$ and obtained a value $v_{1,s}$.**
  - Since p_i is a MAX node, $v_{1,s}$ is a lower bound of v_1 which is usually larger than the minimum possible value.
  - The lower bound of v_1 is thus raised.
  - It is possible because of this probe, a beta cut can be performed.
- **The above process is called an exact probe.**
  - We can first probe each T_i.
  - It is better to probe the best possible branch of T_i first.

Illustration: Probe



The first child of Ti is probed.

Star2

- **Algorithm $Star2\_F3.2'$(position p, node x, value $alpha$, value $beta$) // MAX node**
  - // a chance node x with c choices k_1, …, k_c
  - // the ith choice happens with the probability $\Pr_i$
  - determine the possible values of the chance node x to be $k_1, \ldots, k_c$
  - // Do some probings to decide whether some cut off can be performed.
  - for each choice i from 1 to c do
    - ▷ *Let p_i be the position obtained from p by making x the choice k_i.*
    - ▷ *do an exact probe on the first child of p_i*
    - ▷ *If p is a MAX chance node, then p_i is a MIN node and you may get an alpha cut off for p_i since the probe returns an upper bound for p_i.*
    - ▷ *If p is a MIN chance node, then p_i is a MAX node and you may get an beta cut off for p_i since the probe returns a lower bound for p_i.*
  - // normal exhaustive search phase
  - If no cut off is found in the above, do the normal Star1 search.
    - ▷ *Additional alpha/beta cut off from searching a particular choice.*
    - ▷ *Chance node cut off I that is similar to beta cut off.*
    - ▷ *Chance node cut off II that is similar to alpha cut off.*
  - return $vexp$;

More ideas for probes

- Move ordering in exploring the choices is critical in performance.
- Picking which child to do the probe is also critical.
- Can do exact probes on h children, called **probing factor** $h > 1$, of a choice instead of fixing the number of probings to be exactly one.
  - When $h = 0$, star2 == star1.
  - Sequential probing
    - ▷ *Probe h children of a choice at one time.*
    - ▷
```
for $i = 1$ to $c$ do
    probe $h$ children of the $i$th choice
```
  - Cyclic probing
    - ▷ *Probe 1 child of a choice at one time for all choices, and do this for h rounds.*
    - ▷
```
for $j = 1$ to $h$ do
    for $i = 1$ to $c$ do
        probe the $j$th child of the $i$th choice
```
  - When $h = 1$, cyclic probing == sequential probing.
- May decide to probe different number of children for each choice.

Star2.5: cyclic probing

- **Using a cyclic probing order in Star2 with a probing factor h.**
- **Algorithm $Star2.5\_F3.2'$(position p, node x, value $alpha$, value $beta$, integer h) // MAX node, h is the probing factor**
  - // a chance node x with c choices k_1, …, k_c
  - // the ith choice happens with the probability $\Pr_i$
  - determine the possible values of the chance node x to be $k_1, \ldots, k_c$
  - // Do a cyclic probing to decide whether some cut off can be performed.
  - for j from 1 to h do
    for each choice i from 1 to c do
    - ▷ *Let p_i be the position obtained from p by making x the choice k_i.*
    - ▷ *do an exact probe on the jth child of p_i*
    - ▷ *If p is a MAX chance node, then p_i is a MIN node and you may get an alpha cut off for p_i since the probe returns an upper bound for p_i.*
    - ▷ *If p is a MIN chance node, then p_i is a MAX node and you may get an beta cut off for p_i since the probe returns a lower bound for p_i.*
  - If no cut off is found in the above, do the normal Star1 search.
    - ▷ *Additional alpha/beta cut off from searching a particular choice.*
    - ▷ *Chance node cut off I that is similar to beta cut off.*
    - ▷ *Chance node cut off II that is similar to alpha cut off.*

Comments

- Experimental results provided in [Ballard '83] on artificial game trees.
- Star1 may not be able to cut more than 20% of the leaves.
- Star2.5 with $h = 1$ cuts more than 59% of the nodes and is about twice better than Star1.
- Sequential probing is best when $h = 3$ which cuts more than 65% of the nodes and roughly cut about the same nodes as Star2.5 using the same probing factor.
- Sequential probing gets worse when $h > 4$. For example, it only cut 20% of the leaves when $h = 20$.
- Star2.5 continues to cut more nodes when h gets larger, though the gain is not that great. At $h = 3$, about 70% of the nodes are cut. At $h = 20$, about 72% of the nodes are cut.

Approximated Probes

- **We can also have heuristics for issuing approximated probes which returns approximated values.**
- **Strategy I: random probing of some promising choices**
  - Do a move ordering heuristic to pick one or some promising choices to expand first.
  - These promising choices can improve the lower or upper bounds and can cause beta or alpha cut off.
- **Strategy II: fast probing of all choices**
  - Possible implementations
    - ▷ *do a static evaluation on all choices*
    - ▷ *do a shallow alpha-beta searching on each choice*
    - ▷ *do a MCTS-like simulation on the choices*
  - Use these information to decide whether you have enough confidence to do a cut off.

Using MCTS with chance nodes (1/2)

- **Assume a chance node x has c choices $k_1, \ldots, k_c$ and the ith choice happens with the probability Pr_i**
- **Selection**
  - **If x is picked in the PV during selection, then a random coin tossing according to the probability distribution of the choices is needed to pick which choice to descent.**
    - ▷ *It is better to even the number of simulations done on each choice.*
    - ▷ *Use random sampling without replacement. When every one is picked once, then start another round of picking.*
- **Expansion**
  - **If the last node in the PV is x, then expand all choices and simulate each choice some number of times.**
    - ▷ *Watch out the discuss on maxing chance nodes in a searching path such as whether it is desirable to have 2 chance nodes in sequence ... etc.*

Using MCTS with chance nodes (2/2)

- Simulation
  - When a chance node is to be simulated, then be sure to randomly, according to the probability distribution, pick a choice.
    - *Use some techniques to make sure you are doing an effective sampling when the number of choices is huge*
    - *Watch out what are "reasonable" in a simulated plyout on the mixing of chance nodes.*
- Back propagation
  - The UCB score of x is $w_i + c\sqrt{(lnN/N_i)}$ where w_i is the weighted winning rate, or score, of the children, N_i is the total number of simulations done on all choices. and N is the total number of simulations done on the parent of x.

Sparse sampling (1/2)

- Assume in searching the number of possible outcomes in a, maybe chance, node is too large. A technique called sparse sampling can be used [Kearn et al 2002].
  - Can also be used in the expansion phase of MCTS.
- Ideas:
  - The number of choices, $a = |\mathcal{A}|$, considered is enlarged as the number of visits to the node increases.
  - Use the current choice set as an estimation of its goodness.
  - Only consider k_t randomly selected choices, called $\mathcal{S}_t$, in the first t visits where $k_t = \lceil c * t^{\alpha} \rceil$, and c and α are constants.
- Algorithm SS for sparse sampling
  - $t := 1$
  - Initial k_t to be a small constant, say 1.
  - Initial the candidate set $\mathcal{S}$ to be an empty set.
  - Randomly pick k_t children from $\mathcal{A}$ into $\mathcal{S}$
  - loop: Performs some t' samplings from $\mathcal{S}$.
    - ▷ *Add randomly $k_{t+t'} - k_t$ new children from $\mathcal{A}$ into $\mathcal{S}$*
    - ▷ $t \mathrel{+}= t'$
  - goto loop

Sparse sampling (2/2)

- **The estimated value is accurate with a high probability [Kearns et al 2002] [Lanctot et al 2013]**
- **Theorem:**

$$Pr(|\tilde{V} - V| \leq \lambda \cdot d) \geq 1 - (2 \cdot k_t \cdot c)^d exp\{\frac{-\lambda^2 \cdot k_t}{2 \cdot v_{max}^2}\},$$

  where

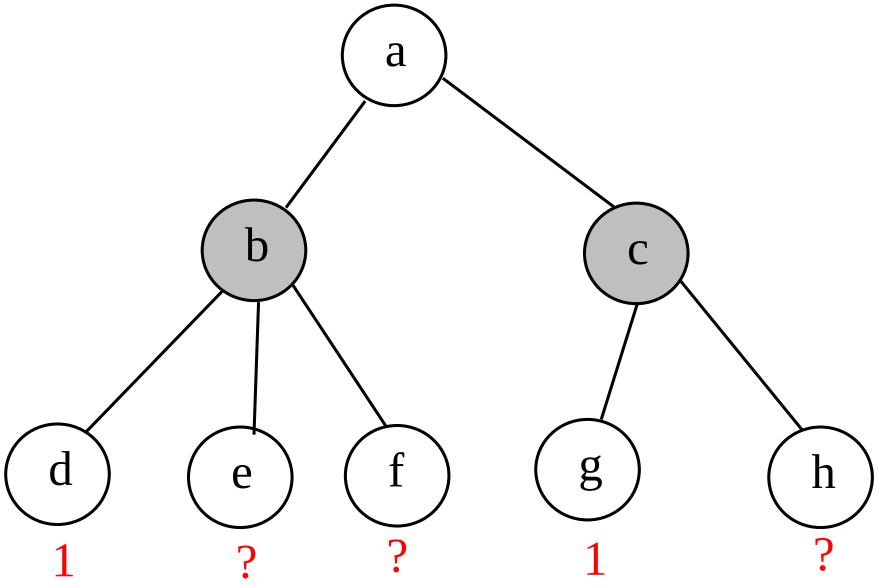
  - ▷ k_t ***is the number of choices considered with*** t ***samplings,***
  - ▷ $\tilde{V}$ ***is the estimation considering only*** k_t ***choices,***
  - ▷ V ***is the value considering all choices,***
  - ▷ c ***is the actual number of choices,***
  - ▷ d ***is the depth simulated,***
  - ▷ $\lambda \in (0, 2 \cdot v_{max}]$ ***is a parameter chosen, and***
  - ▷ v_{max} ***is the maximum possible value.***
- **Note: the proof is done by making sampling with replacement, while the algorithm asks for sampling without replacement.**

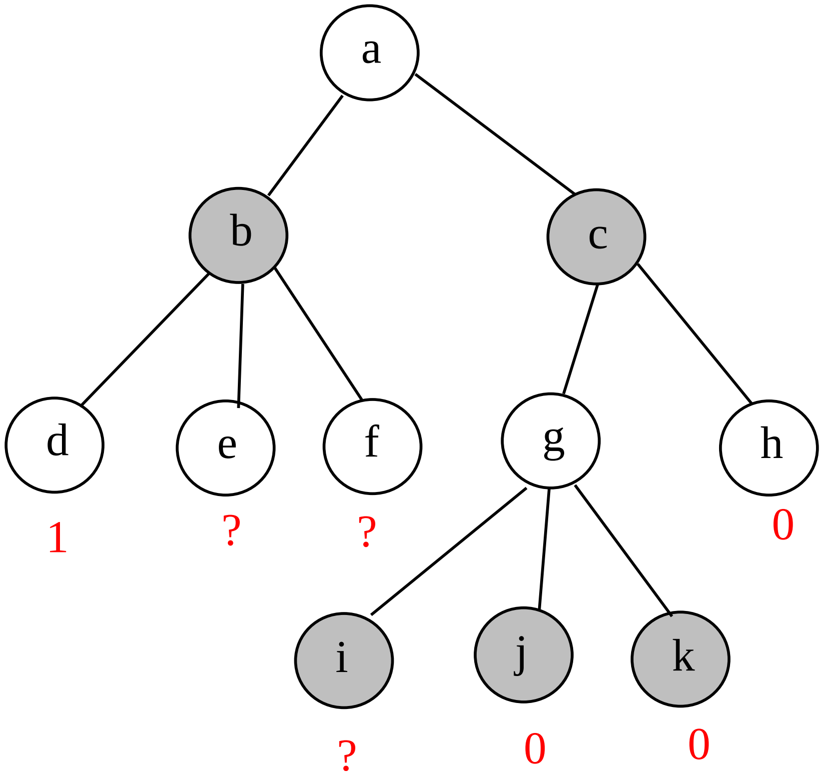
Proof number search

- 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?

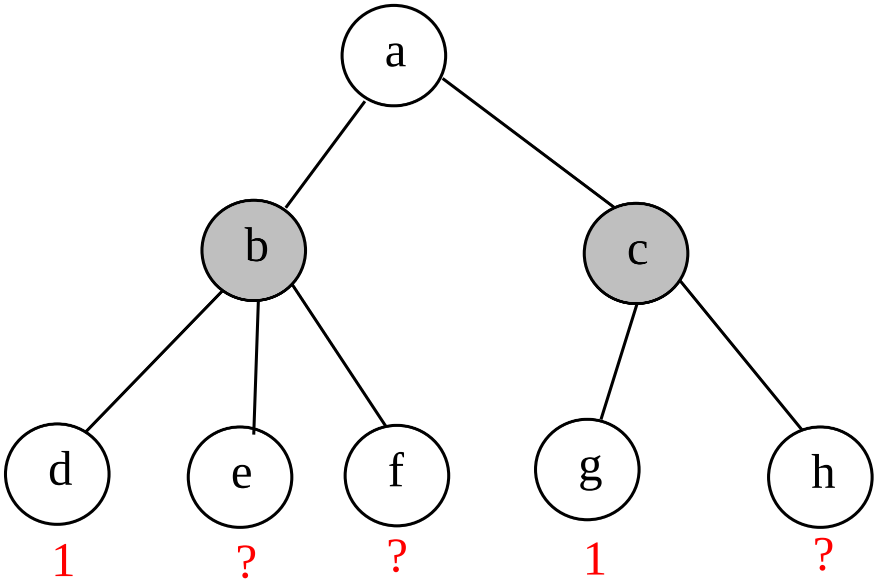
- **A most proving node for a node u: a descendent node if its value is 1, then the value of u is 1.**
- **A most disproving node for a node u: a descendent node if its value is 0, then the value of u is 0.**





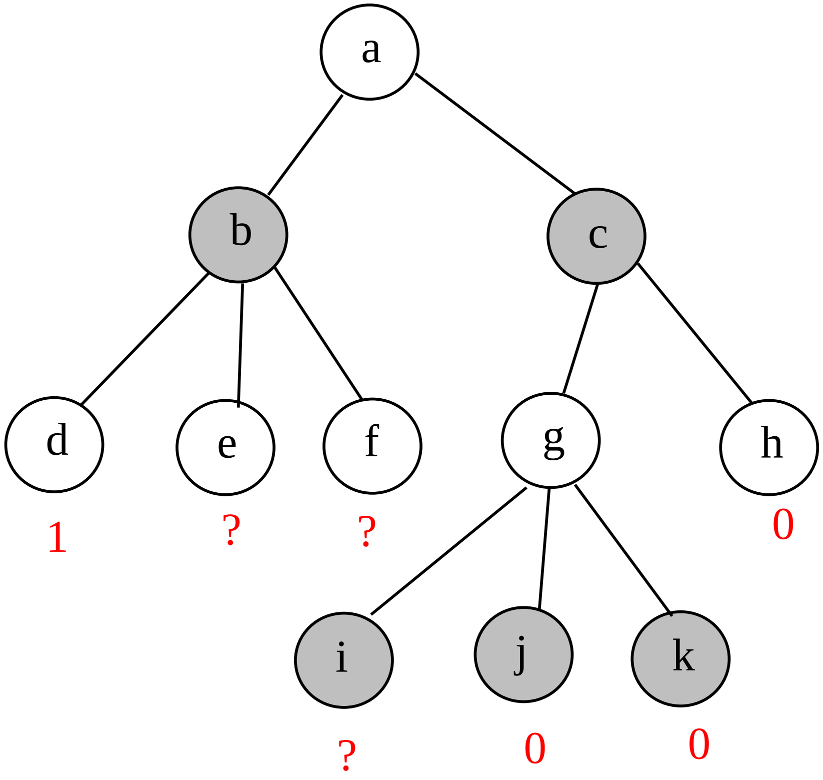
Most proving node

- **Node h is a most proving node for a.**



Most disproving node

- **Node e or f is a most disproving node for a.**

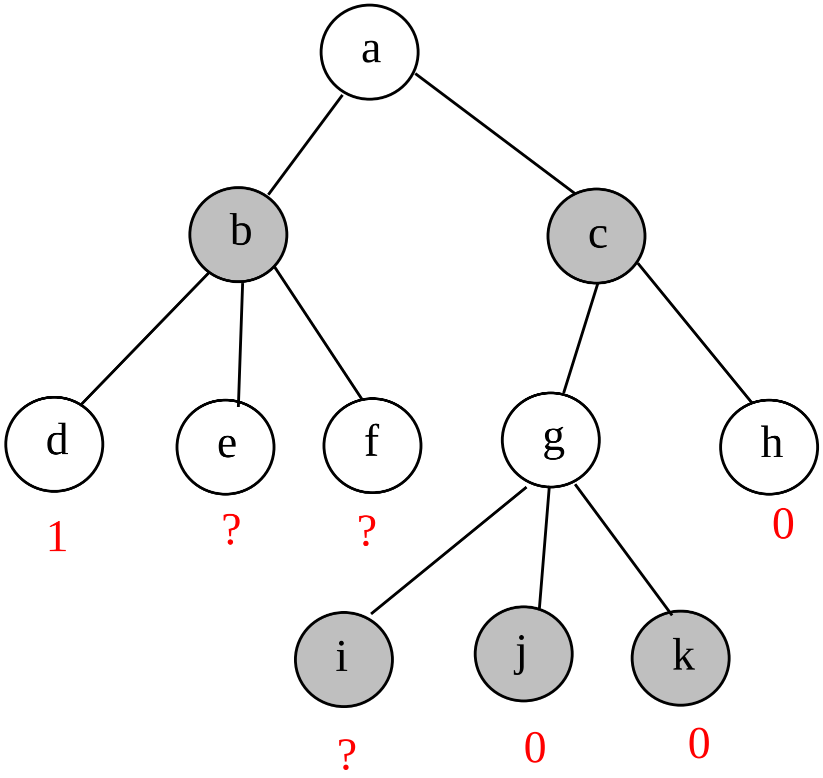


Proof or Disproof Number

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

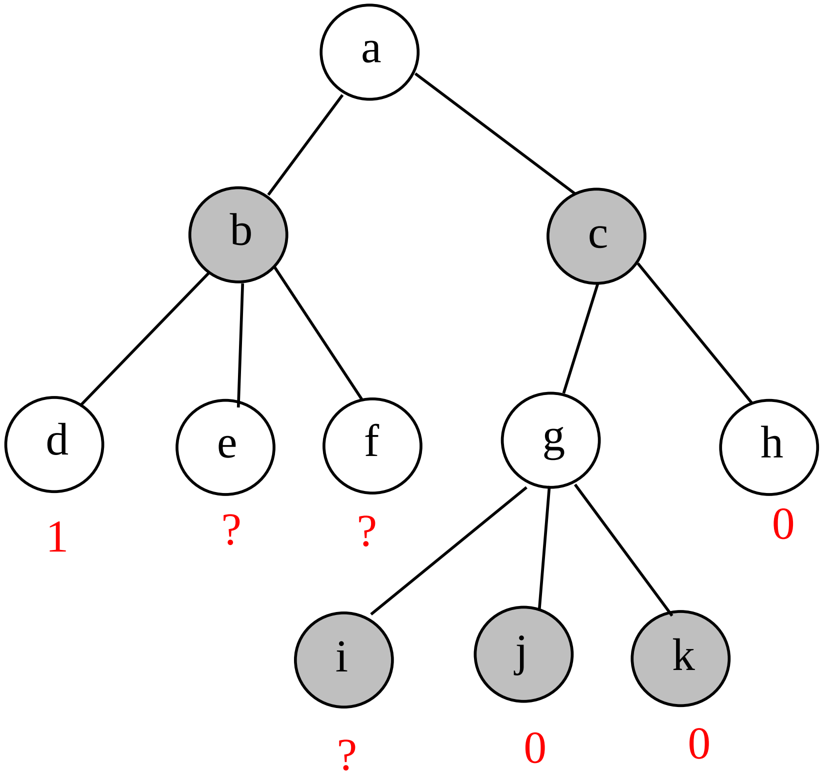
Proof number

- **Proof number for the root a is 2.**
  - ▷ ***Need to at least prove e and f.***



Disproof number

- **Disproof number for the root a is 2.**
  - ▷ ***Need to at least disprove i, and either e or f.***



Proof Number: Definition

- u **is a leaf:**
  - **If** $value(u)$ **is unknown, then** $proof(u)$ **is the cost of evaluating** u**.**
  - **If** $value(u)$ **is 1, then** $proof(u) = 0$**.**
  - **If** $value(u)$ **is 0, then** $proof(u) = \infty$**.**
- u **is an internal node with all of the children** $u_1, \ldots, u_b$**:**
  - **if** u **is a MAX node,**

$$proof(u) = \min_{i=1}^{i=b} proof(u_i);$$

  - **if** u **is a MIN node,**

$$proof(u) = \sum_{i=1}^{i=b} proof(u_i).$$

Disproof Number: Definition

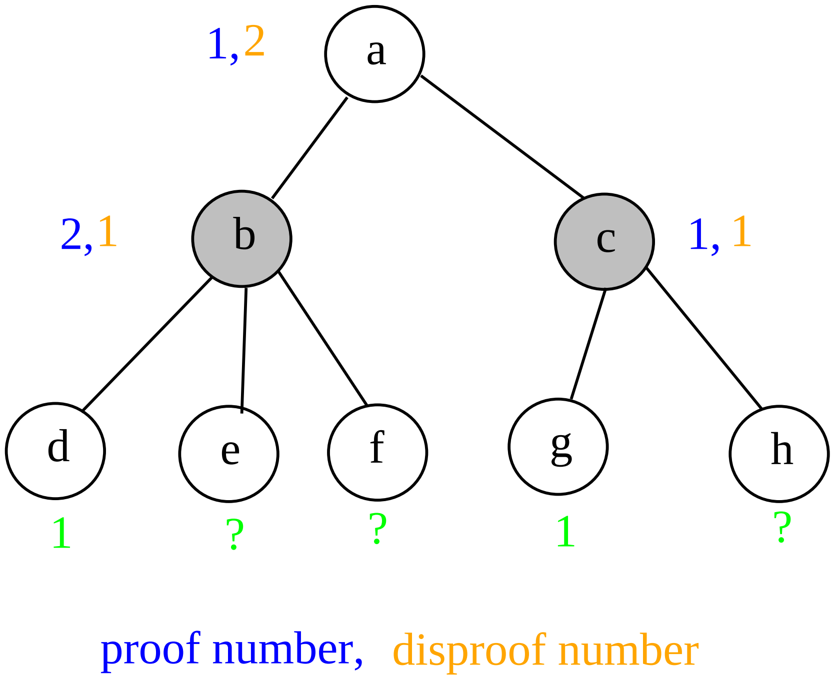
- u **is a leaf:**
  - **If** $value(u)$ **is unknown, then** $disproof(u)$ **is cost of evaluating** u**.**
  - **If** $value(u)$ **is 1, then** $disproof(u) = \infty$**.**
  - **If** $value(u)$ **is 0, then** $disproof(u) = 0$**.**
- u **is an internal node with all of the children** $u_1, \ldots, u_b$**:**
  - **if** u **is a MAX node,**

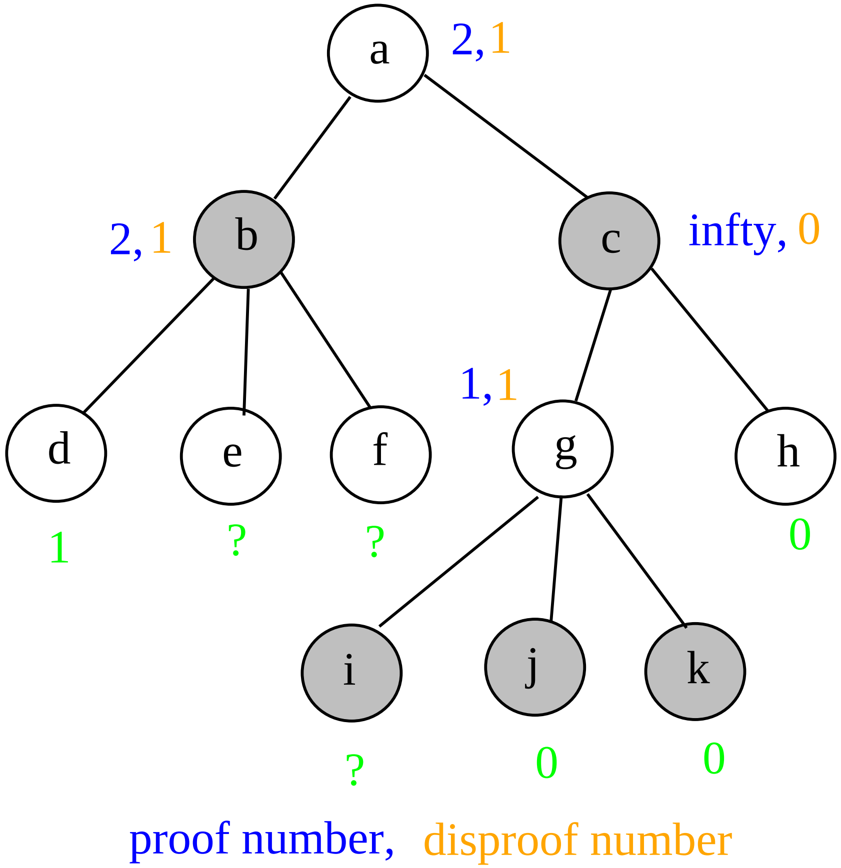
$$disproof(u) = \sum_{i=1}^{i=b} disproof(u_i);$$

  - **if** u **is a MIN node,**

$$disproof(u) = \min_{i=1}^{i=b} disproof(u_i).$$

Illustrations





How these numbers are used (1/2)

- **Scenario:**
  - **For example, the tree T represents an open game tree or an endgame tree.**
    - ▷ *If T is an open game tree, then maybe it is asked to prove or disprove a certain open game is win.*
    - ▷ *If T is an endgame tree, then maybe it is asked to prove or disprove a certain endgame is win o loss.*
    - ▷ *Each leaf takes a lot of time to evaluate.*
    - ▷ *We need to prove or disprove the tree using as few time as possible.*
  - **Depend on the results we have so far, pick a leaf to prove or disprove.**
- **Goal: solve as few leaves as possible so that in the resulting tree, either $proof(root)$ or $disproof(root)$ becomes 0.**
  - **If $proof(root) = 0$, then the tree is proved.**
  - **If $disproof(root) = 0$, then the tree is disproved.**
- **Need to be able to update these numbers on the fly.**

How these numbers are used (2/2)

- **Let $GV = \min\{proof(root), disproof(root)\}$.**
  - **GT is "prove" if $GV = proof(root)$, which means we try to prove it.**
  - **GT is "disprove" if $GV = disproof(root)$, which means we try to disprove it.**
  - **In the case of $proof(root) = disproof(root)$, we set GT to "prove" for convenience.**
- **From the root, we search for a leaf whose value is unknown.**
  - **The leaf found is a most proving node if GT is "prove", or a most disproving node if GT is "disprove".**
  - **To find such a leaf, we start from the root downwards recursively as follows.**
    - ▷ ***If we have reached a leaf, then stop.***
    - ▷ ***If GT is "prove", then pick***
      a child with the least proof number for a MAX node, and
      any node that has a chance to be proved for a MIN node.
    - ▷ ***If GT is "disprove", then pick***
      a child with the least disproof number for a MIN node, and
      any node that has a chance to be disproved for a MAX node.

PN-search: algorithm (1/2)

- **$\{*$ Compute and update proof and disproof numbers of the root in a bottom up fashion until it is proved or disproved. $*\}$**
- *loop:*
  - **If $proof(root) = 0$ or $disproof(root) = 0$, then we are done, otherwise**
    - *$proof(root) \leq disproof(root)$: **we try to prove it.***
    - *$proof(root) > disproof(root)$: **we try to disprove it.***
  - **$u \leftarrow root$; $\{*$ find a 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***
      $u \leftarrow$ leftmost child of u with the smallest non-zero proof number;
    - ***else if** u **is a MIN node, then***
      $u \leftarrow$ leftmost child of u with a non-zero proof number;
  - **else 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;
    - ***else if** u **is a MIN node, then***
      $u \leftarrow$ leftmost child of u with the smallest non-zero disproof number;

PN-search: algorithm (2/2)

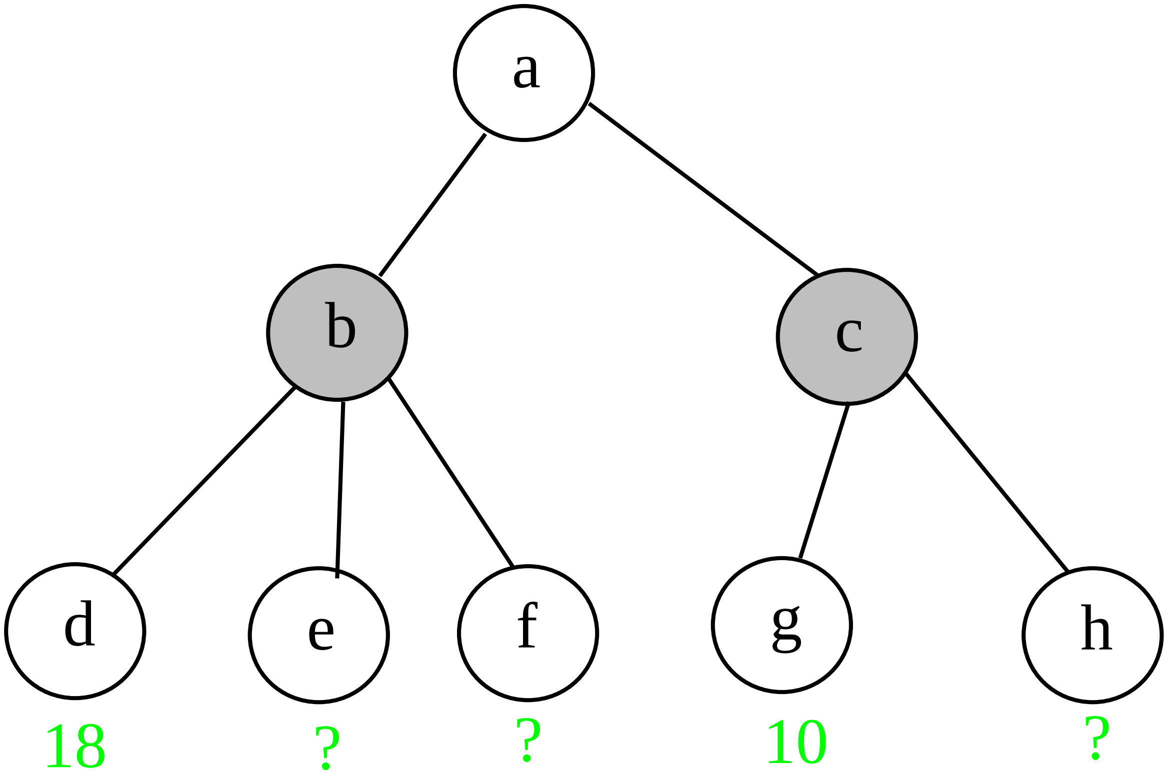
- **{$*$ Continued from the last page $*$}**
  - **solve u;**
  - **repeat {$*$ bottom up updating the values $*$}**
    - ***update*** $proof(u)$ ***and*** $disproof(u)$
    - $u \leftarrow u'$***s parent***

    until u is the root
  - **go to $loop$;**

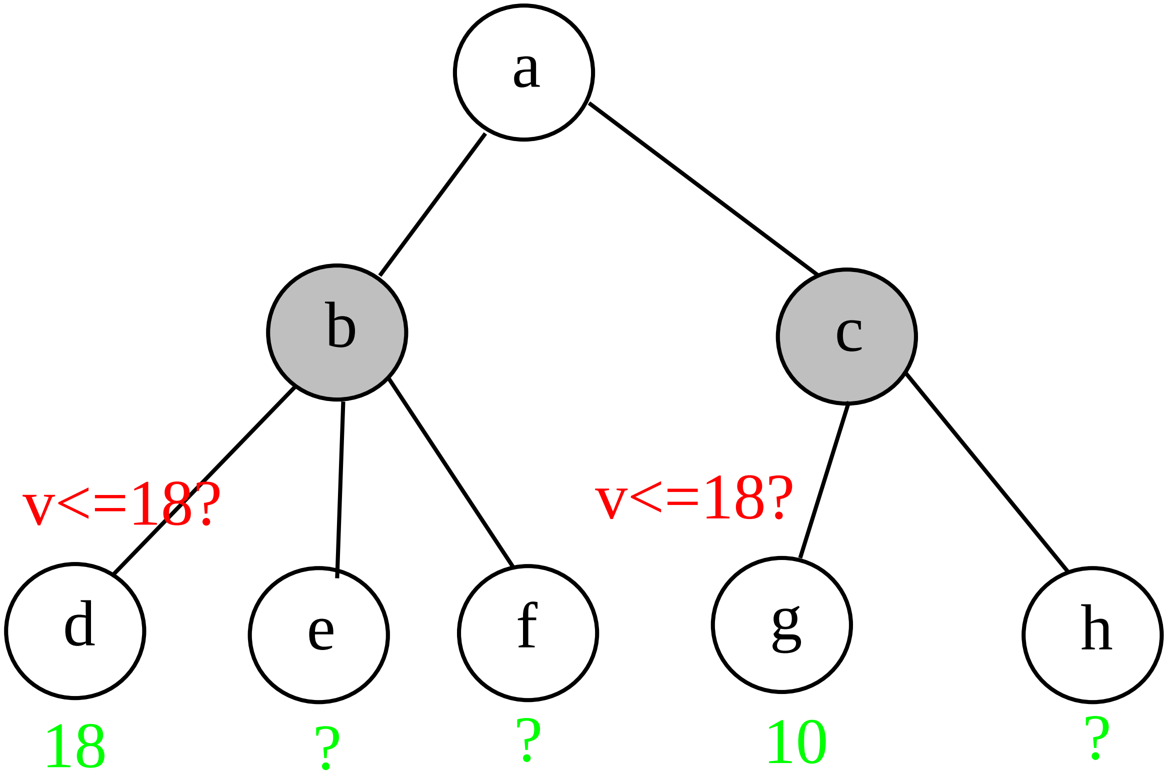
Multi-Valued game Tree

- **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.**
  - $proof_v(u)$: **the minimum number of leaves needed to visited in order for the value of u to $\geq v$.**
    - ▷ $proof(u) \equiv proof_1(u)$.
  - $disproof_v(u)$: **the minimum number of leaves needed to visited in order for the value of u to $< v$.**
    - ▷ $disproof(u) \equiv disproof_1(u)$.

Illustration



Illustration



Multi-Valued proof number

- u **is a leaf:**
  - **If** $value(u)$ **is unknown, then** $proof_v(u)$ **is cost of evaluating** u**.**
  - **If** $value(u) \geq v$**, then** $proof_v(u) = 0$**.**
  - **If** $value(u) < v$**, then** $proof_v(u) = \infty$**.**
- u **is an internal node with all of the children** $u_1, \ldots, u_b$**:**
  - **if** u **is a MAX node,**

$$proof_v(u) = \min_{i=1}^{i=b} proof_v(u_i);$$

  - **if** u **is a MIN node,**

$$proof_v(u) = \sum_{i=1}^{i=b} proof_v(u_i).$$

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,**

$$disproof_v(u) = \sum_{i=1}^{i=b} disproof_v(u_i);$$

  - **if** u **is a MIN node,**

$$disproof_v(u) = \min_{i=1}^{i=b} disproof_v(u_i).$$

Revised PN-search(v): algorithm (1/2)

- **$\{*$ Compute and update proof$_v$ and disproof$_v$ numbers of the root in a bottom up fashion until it is proved or disproved. $*\}$**
- *loop:*
  - **If $proof_v(root) = 0$ or $disproof_v(root) = 0$, then we are done, otherwise**
    - ▷ ***$proof_v(root) \leq disproof_v(root)$: we try to prove it.***
    - ▷ ***$proof_v(root) > disproof_v(root)$: we try to disprove it.***
  - **$u \leftarrow root$; $\{*$ find a 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
      $u \leftarrow$ leftmost child of $u$ with the smallest non-zero proof$_v$ number;
▷   else if $u$ is a MIN node, then
      $u \leftarrow$ leftmost child of $u$ with a non-zero proof$_v$ number;
```

  - **else 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$_v$ number;
▷   else if $u$ is a MIN node, then
      $u \leftarrow$ leftmost child of $u$ with the smallest non-zero disproof$_v$ number;
```

PN-search: algorithm (2/2)

```
{* Continued from the last page *}
    solve $u$;
    repeat {* bottom up updating the values *}
        update $proof_v(u)$ and $disproof_v(u)$
        $u \leftarrow u'$s parent
    until $u$ is the root
    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

- 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.

More research topics

- **Do variations of a game make it different?**
  - **Whether Stalemate is draw or win in chess.**
  - **Japanese and Chinese rules in Go.**
  - **Chinese and Asia rules in Chinese chess.**
  - **...**
- **Why a position is easy or difficult to human players?**
  - **Can be used in tutoring or better understanding of the game.**

Unique features in games

- **Games are used to model real-life problems.**
- **Do unique properties shown in games help modeling real applications?**
  - Chinese chess
    - ▷ *Very complicated rules for loops: can be draw, win or loss.*
    - ▷ *The usage of cannons for attacking pieces that are blocked.*
  - Go: the rule of Ko to avoid short cycles, and the right to pass.
  - Chinese dark chess: a chance node that makes a deterministic ply first, and then followed by a random toss.
  - EWN: a chance node that makes a random toss first, and then followed with a deterministic ply later.
  - Shogi: the ability to capture an opponent's piece and turn it into your own.
  - Chess: stalemate is draw.
  - Promotion: a piece may turn into a more/less powerful one once it satisfies some pre-conditions.
    - ▷ *Chess*
    - ▷ *Shogi*
    - ▷ *Chinese chess: the mobility of a pawn is increased once it advances twice, but is decreased once it reaches the end of a column.*

References and further readings (1/3)

- L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.
- David Carmel and Shaul Markovitch. Learning and using opponent models in adversary search. Technical Report CIS9609, Technion, 1996.
- M. Campbell. The graph-history interaction: on ignoring position history. In *Proceedings of the 1985 ACM annual conference on the range of computing : mid-80's perspective*, pages 278–280. ACM Press, 1985.
- Akihiro Kishimoto and Martin Müller (2004). A General Solution to the Graph History Interaction Problem. AAAI, 644–648, 2004.
- Kuang-che Wu, Shun-Chin Hsu and Tsan-sheng Hsu "The Graph History Interaction Problem in Chinese Chess," Proceedings of the 11th Advances in Computer Games Conference, (ACG), Springer-Verlag LNCS# 4250, pages 165–179, 2005.

References and further readings (2/3)

* **Bruce W. Ballard The *-minimax search procedure for trees containing chance nodes Artificial Intelligence, Volume 21, Issue 3, September 1983, Pages 327-350**
- **Marc Lanctot, Abdallah Saffidine, Joel Veness, Chris Archibald, Mark H. M. Winands Monte-Carlo *-MiniMax Search Proceedings IJCAI, pages 580–586, 2013.**
- **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.**
- **Chaslot, Guillaume, Winands, Mark, Herik, H., Uiterwijk, Jos, Bouzy, Bruno. (2008). Progressive Strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation. 04. 343-357. 10.1142/S1793005708001094.**

References and further readings (3/3)

- Coutoux A., Hoock JB., Sokolovska N., Teytaud O., Bonnard N. (2011) Continuous Upper Confidence Trees. In: Coello C.A.C. (eds) Learning and Intelligent Optimization. LION 2011. Lecture Notes in Computer Science, vol 6683. Springer, Berlin, Heidelberg.
- Hung-Jui Chang and Cheng Yueh and Gang-Yu Fan and Ting-Yu Lin and Tsan-sheng Hsu (2021). Opponent Model Selection Using Deep Learning. Proceedings of the 2021 Advances in Computer Games (ACG).