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

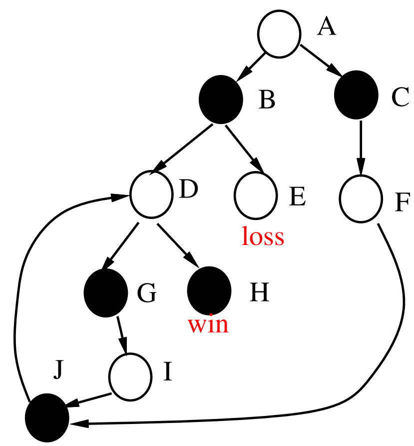
Abstract

- **Some advanced research issues.**
  - **The graph history interaction (GHI) problem.**
  - **Opponent models.**
  - **Multi-player game tree search.**
  - **Bit board speedup.**
  - **Proof-number search.**
- **More research topics.**
  - **The influence of rules on games.**
    - ▷ *Allowing long cycles in Go.*
    - ▷ *The scoring of a suicide ply in chess.*
  - **Why a position is difficult to human?**
  - **Unique features in games.**

Graph history interaction problem

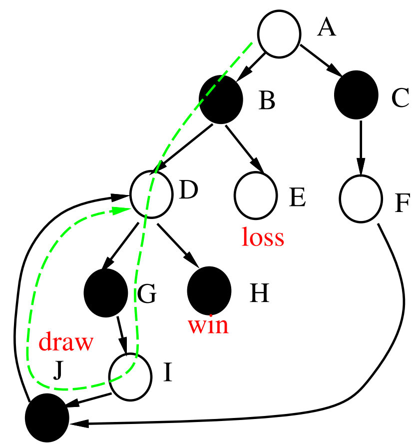
- **The graph history interaction (GHI) problem [Campbell 1985]:**
  - **In a game graph, a position can be visited by more than one paths from a starting position.**
  - **The value of the position depends on the path visiting it.**
    - *It can be win, loss or draw for Chinese chess.*
    - *It can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



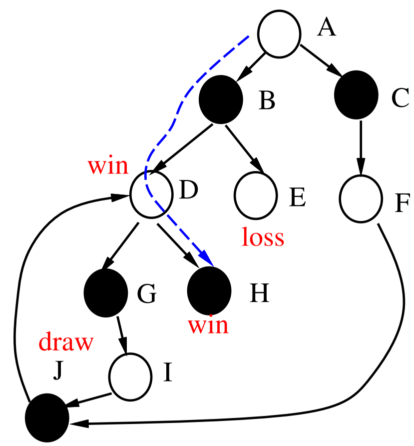
- **Assume if the game falls into a loop, then it is a draw.**

GHI: when loop draws



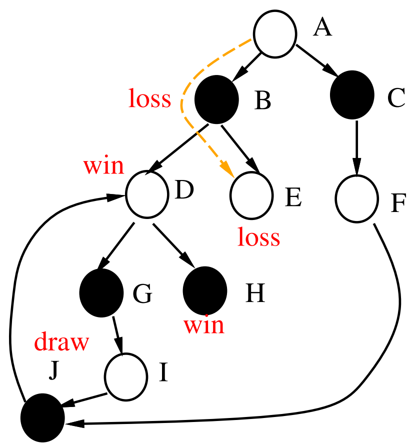
- **Assume if the game falls into a loop, then it is a draw.**
- $A \to B \to D \to G \to I \to J \to D$ **is draw by rules of repetition.**
  - ▷ ***Memorized J as a draw position.***

GHI: when loop draws



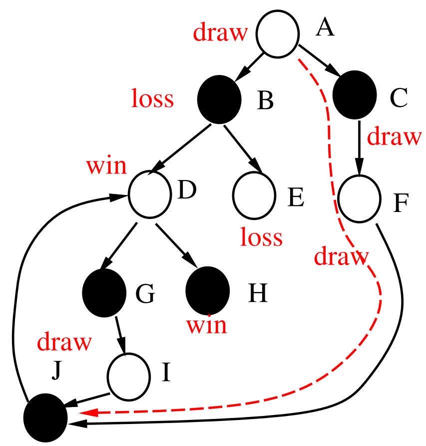
- **Assume if the game falls into a loop, then it is a draw.**
- $A \to B \to D \to G \to I \to J \to D$ **is draw by rules of repetition.**
  - ▷ ***Memorized J as a draw position.***
- $A \to B \to D \to H$ **is a win. Hence D is win.**

GHI: when loop draws



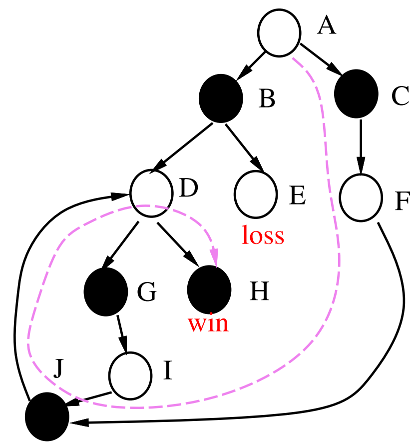
- **Assume if the game falls into a loop, then it is a draw.**
- $A \to B \to D \to G \to I \to J \to D$ **is draw by rules of repetition.**
  - ▷ *Memorized J as a draw position.*
- $A \to B \to D \to H$ **is a win. Hence** D **is win.**
- $A \to B \to E$ **is a loss. Hence** B **is loss.**

GHI: when loop draws



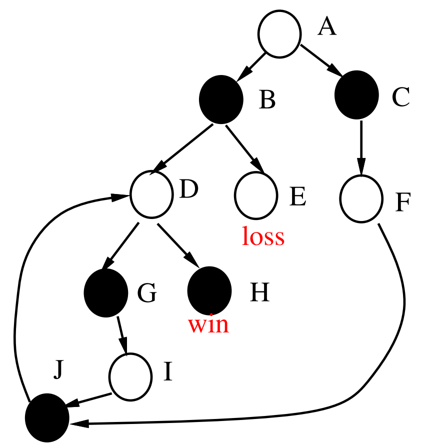
- **Assume if the game falls into a loop, then it is a draw.**
- $A \to B \to D \to G \to I \to J \to D$ **is draw by rules of repetition.**
  - ▷ *Memorized J as a draw position.*
- $A \to B \to D \to H$ **is a win. Hence** D **is win.**
- $A \to B \to E$ **is a loss. Hence** B **is loss.**
- $A \to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



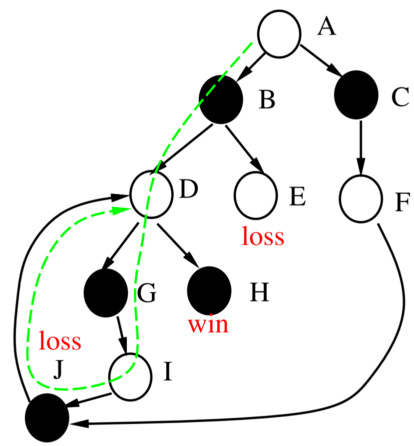
- **Assume if the game falls into a loop, then it is a draw.**
- $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ **is draw by rules of repetition.**
  - ▷ ***Memorized J as a draw position.***
- $A \rightarrow B \rightarrow D \rightarrow H$ **is a win. Hence**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

GHI: when loop wins



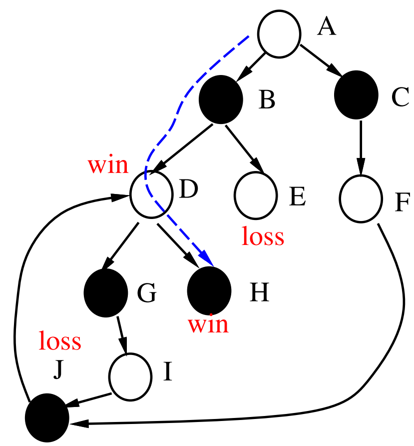
- **Assume the one causes loops wins the game.**

GHI: when loop wins



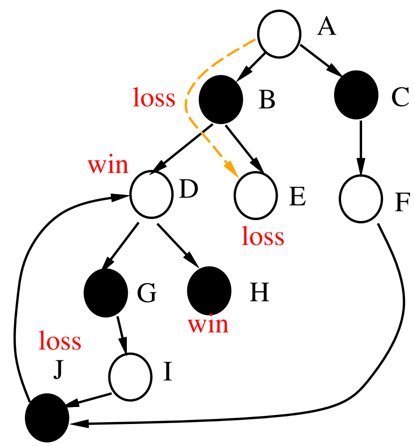
- **Assume the one causes loops wins the game.**
- $A \to B \to D \to G \to I \to J \to D$ **is loss because of rules of repetition.**
  - ▷ ***Memorized*** J ***as a loss position (for the root).***

GHI: when loop wins



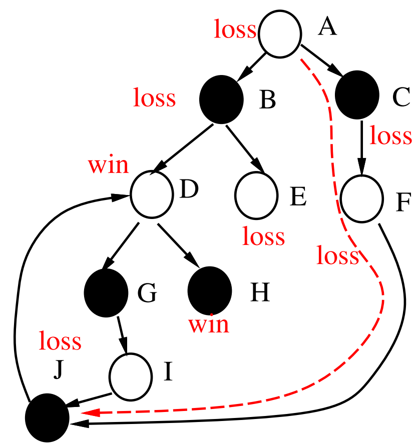
- **Assume the one causes loops wins the game.**
- $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ **is loss because of rules of repetition.**
  - ▷ ***Memorized J as a loss position (for the root).***
- $A \rightarrow B \rightarrow D \rightarrow H$ **is a win.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

GHI: when loop wins



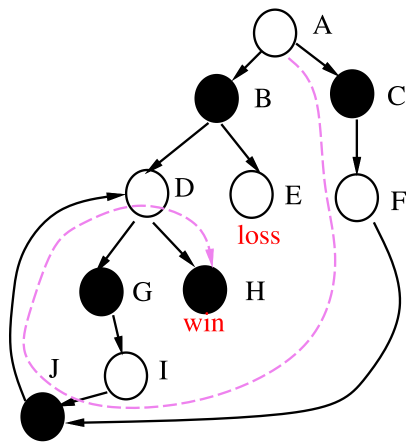
- **Assume the one causes loops wins the game.**
- $A \to B \to D \to G \to I \to J \to D$ **is loss because of rules of repetition.**
  - ▷ *Memorized J as a loss position (for the root).*
- $A \to B \to D \to H$ **is a win. Hence D is win.**
- $A \to B \to E$ **is a loss. Hence B is loss.**

GHI: when loop wins



- **Assume the one causes loops wins the game.**
- $A \rightarrow B \rightarrow D \rightarrow G \rightarrow I \rightarrow J \rightarrow D$ **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.**

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.**
- **However, $A \to C \to F \to J \to D \to H$ 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.
- Finding a better data structure for solving this problem remains to be a challenging research issue.

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 evaluation 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.**

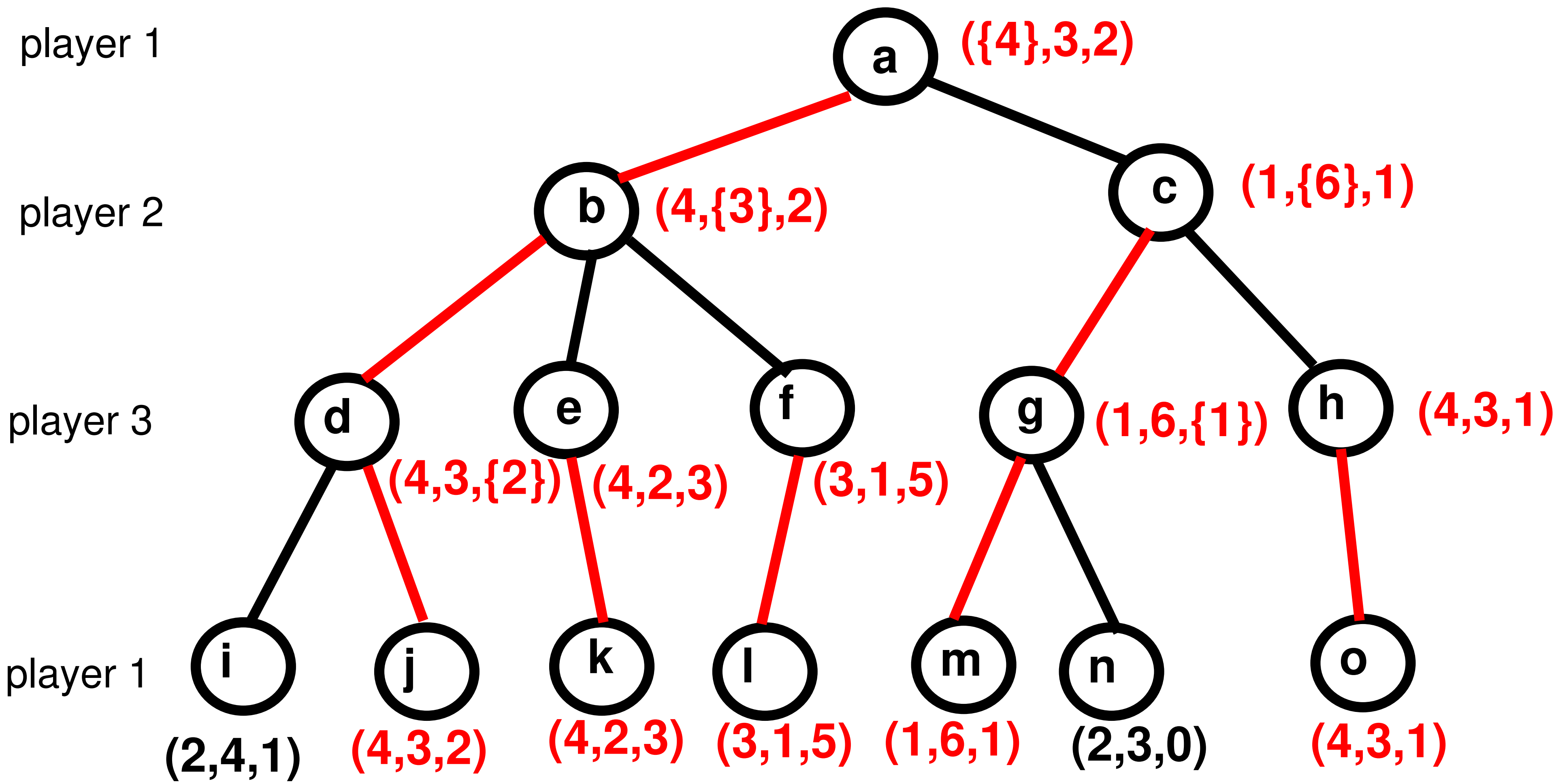
Multi-player game tree search

- **Games with more than 2 players.**
  - **Mahjong: 4 players**
  - **Contract bridge or bridge: 4 players**
  - **Monopoly: 2 to many players**
  - **Scrabble: 2 to 4 players**
  - **Risk: 2 to 6 players**
- **Assume we have n players, $y_1, \ldots, y_n$ in a game.**
  - **We have n evaluating functions, $score_i$, one for each player.**
  - **Given a position p with the children $p_1, \ldots, p_m$, let $score_i(p)$ be the score of y_i for p.**
    - ▷ ***If p is a terminal position for y_i, then $m = 0$ and $score_i(p)$ is the "true" score of y_i in p.***
    - ▷ ***Otherwise, $score_i(p) = \max_{j=1}^{m} score_i(p_j)$.***
  - **The above algorithm is called MAX^n where stands for during each turn, each player maximizes his own score without considering scores of others.**

MAX^n: algorithm

- $next\_player(idx)$: **the player who is next to player** idx.
- **Brute force algorithm for multi-player games.**
- **Algorithm MAXN(position** p**, player** idx**)**
  - **output:** $best$ **which is an array with** $best[i]$ **being the best value for player** i **so far.**
  - **If** p **is terminal, then return** $best[i] = score_i(p), \forall i$**;**
  - **initialize** $best$ **to be** $best[i] = -\infty, \forall i$**;**
  - **Let** p_i **be the** i**th child of** p**;**
  - **for** $i = 1$ **to last child of** p **do**
    - ▷ $current$ = ***MAXN(***p_i***, next_player(***idx***));***
    - ▷ ***if*** $current[idx] > best[idx]$, $best = current$; ***// maximized player idx***
  - **return** $best$**;**

MAX^n: example ($n = 3$)



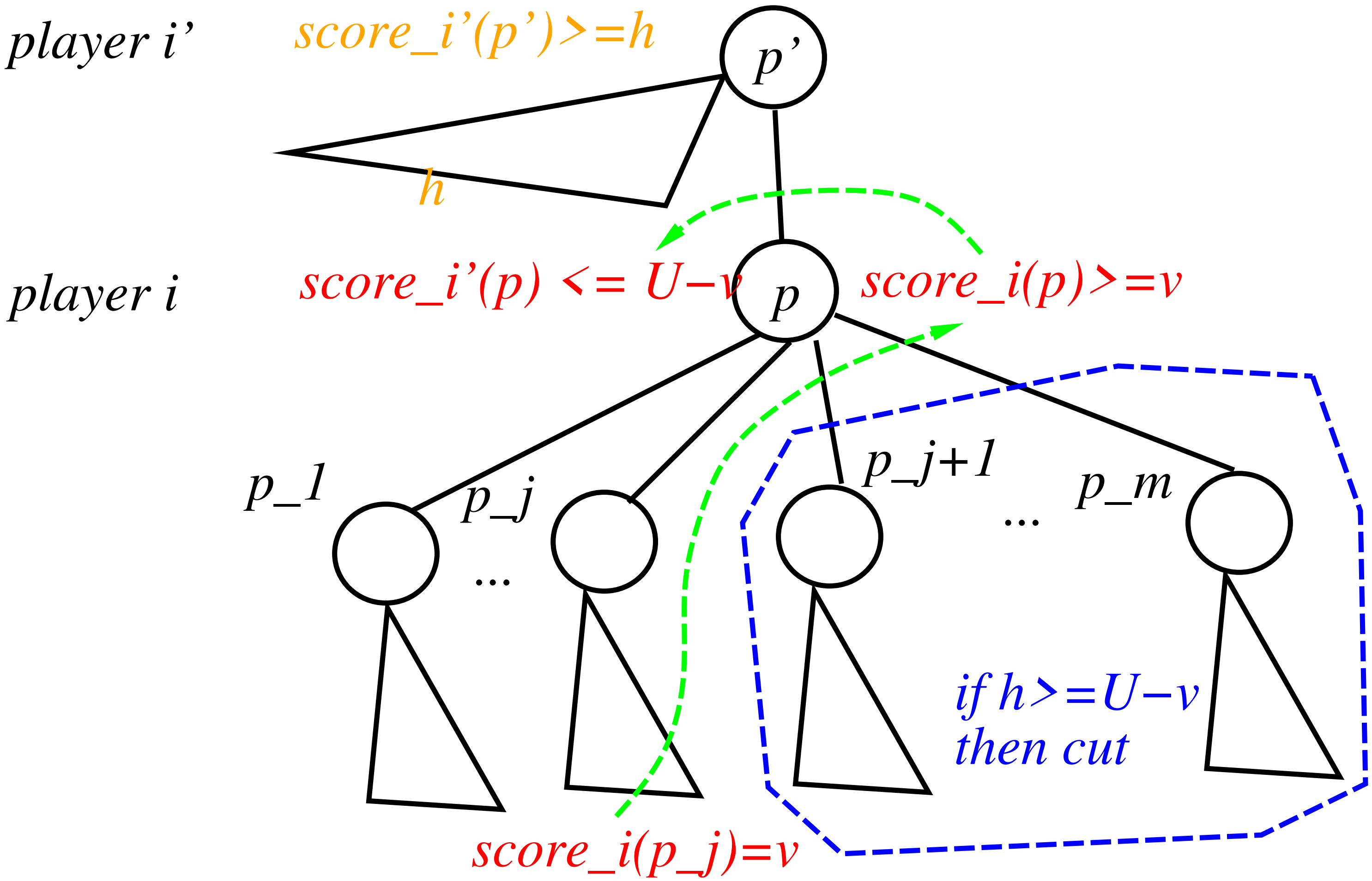
Opportunities for pruning

- **Let p be a position in a multi-player game.**
- **Alpha-beta pruning is a special case for $n = 2$ and cannot be generalized for $n > 2$.**
  - **Property used in alpha-beta pruning:**
    - ▷ *Zero sum: for a position p, $score_1(p) + score_2(p) = 0$.*
    - ▷ *What is good for y_1 is definitely bad for y_2.*
  - **The above may not be true for $n > 2$.**
- **For a position p, if there is no constraints on the n scores of p, then it is impossible to have any cut offs for MAX^n.**
  - **In applications we often have the following properties.**
    - ▷ *The sum of all n scores for p has an upper bound U.*
    - ▷ *The score of p for any player has a lower bound L.*
  - **Examples:**
    - ▷ *Go for n players: each player owns pieces of a distinct color.*
      $\rightarrow$ the sum of all points $\leq$ the board size, and the score cannot be negative.
    - ▷ *Othello for n players: each player owns pieces of a distinct color and flips all pieces of different colors.*
      $\rightarrow$ the sum of all points $\leq$ the plys played so far and the score cannot be negative.

Pruning

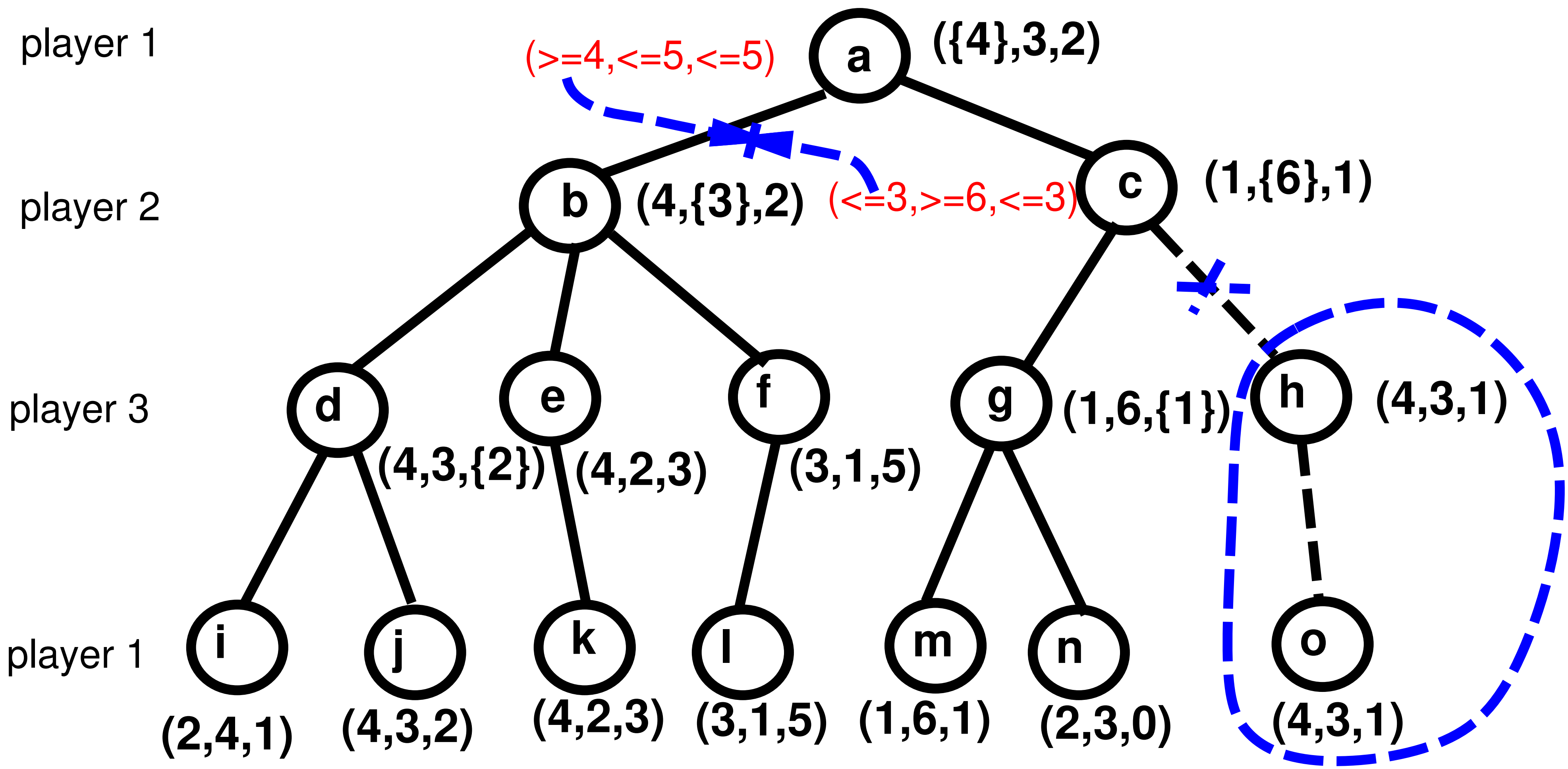
- Recall: a position p with the children $p_1, \ldots, p_m$ and the parent p', and $score_i(p)$ is the score of player i for p.
- Direct pruning:
  - During the turn of the ith player, if $score_i(p_j) = U$, then no more search is needed.
- Shallow pruning:
  - Without loss of generality, assume $L = 0$.
  - During the turn of the ith player, if $score_i(p_j) = v$ so far, then $score_i(p) \geq v$ since each player is a max player.
  - This implies $score_j(p) \leq U - v$ if $j \neq i$.
  - Let i' be the index of the immediate previous player.
  - We know $score_{i'}(p') \geq h$ if he has done some searching.
  - If $h \geq U - v$, then we have a cut off.

MAX^n: ideas



player i'
score_i'(p')>=h
p'
h
player i
score_i'(p) <= U−v
p
score_i(p)>=v
p_1
p_j
p_j+1
p_m
...
...
if h>=U−v
then cut
score_i(p_j)=v

MAX^n: cutoff example ($n = 3, U = 9$)



Remarks about pruning in MAX^n

- **Direct pruning is a degenerated case of the shallow pruning by the following settings.**
  - **If $v = U$, then the scores of all other players are all zero.**
  - **Using the lower bound L, you can get a cut off.**
- **Compared to two-player alpha-beta pruning, both direct and shallow prunningscan be used in $n \geq 2$.**
- **Deep pruning does not work when $n > 2$.**
  - **Assume you are searching the node w, v is your parent and u is an ancestor that is not v.**
  - **Assume node x is the turn of player $player(x)$.**
  - **Any value of $score_{player(u)}(u)$ cannot produce any cutoff on searching the tree T_w because $player(v)$ makes the decision first in propagating the values up.**
  - **Any value of $score_{player(u)}(w)$ can be propagated up and be used by u.**

Algorithm for shallow cut off

- **Functions and data structures**
  - $next\_player(idx)$: the player who is next to player idx.
  - $score_i(p)$: the score of player i for the position p.
  - U: the upper bound of sum of all scores among all players on a position.
  - Assume L is 0.
  - $best$ and $current$ are both arrays of size n.
- **Algorithm shallow(position p, player idx, value $bound$)**
  - return value: $best$ which is an array with $best[i]$ being the best value for player i so far.
  - If p is terminal, then return $best[i] = score_i(p), \forall i$;
  - Let p_i be the ith child of p;
  - $best = shallow(p_1, next(idx), U)$; // recursive call on the first child
  - for $i = 2$ to last child of p do

    4.1: *if* $best[idx] = U$, *then return* $best$ *// immediate cut off*

    4.2: *if* $best[idx] \geq bound$, *then return* $best$ *// shallow cut off*

    4.3: $current = shallow(p_i, next\_player(idx), U - best[idx])$*;*

    4.4: *if* $current[idx] > best[idx]$, $best = current$*; // maximize player idx*
  - return $best$;

Comments

- An generalization for alpha-beta cutoff on adjacent depths.
- Does not work on deep alpha-beta cutoff [Korf 1991].
- In the **best case**, the effective branching factor is $\frac{1+\sqrt{4b-3}}{2}$ where b is the average branching factor.
  - Comparing to alpha-beta cut off, the best effective branching factor is $\sqrt{b}$.
- In the average case, the effective branching factor is approaching $O(b)$.
  - Comparing to alpha-beta cut off, the the average effective branching factor is $b^{0.75}$ [Fuller et al 1975].
  - This implies most of the cut off come from deep pruning in the average case..
- More research are needed to get more cutoff by observing additional constraints on the values from the application domain.
- MCTS can be easily extended to work on any number of players, but need to work on better properties of convergence.

Hardware Speedup

- **Using hardware to speed up searching is not new.**
  - **Parallel computing.**
    - ▷ *The Northwestern University CHESS program series on the 1970's makes full usage of hardware advantages from supercomputers [Atkin & Slate 1977].*
  - **Special hardware acceleration:**
    - ▷ *Belle: a chess machine with special micro instructions for move generation, alpha-beta pruning and transposition table operations [Condon & Thompson 1982].*
    - ▷ *Deep Blue: custom VLSI FPGA chips for operating chess playing expert systems [Hsu et al 1995].*
- **The above were very costed.**

Bit board techniques

- **Everyone can make use of the benefits of hardware acceleration now by smart usage of fast parallel bitwise operations provided by modern day CPU's.**
  - **Intel CPU's: MMX and SSE [Intel 2021]**
  - **AMD: 3D Now! [AMD 2000]**
- **Main technique**
  - **Using bits to represent the board and pieces on the board.**
    - ▷ ***Transfer a board into an $n \times m$ picture***
    - ▷ ***Transfer pieces into patterns of pixel rectangles***
  - **These instructions are usually in the form of SIMD (single instruction multiple data).**
  - **Many are for image related operations.**
  - **May also make use of GPU.**

Special instruction sets (1/2)

- Make use of fast parallel bitwise operations provided by modern day CPU's.
- Many different types
  - Find aggregated information
  - Parallel bit deposit and extract
  - ...
- Most of the instructions can be done using AND, OR, NOT operations, but can be done much faster using special CPU instructions.

Special instruction sets (2/2)

- **Find aggregated information:**
  - **population count (POPCNT): the number of 1-bits in a "word".**
  - **leading/trailing zero count: LZCNT, TZCNT**
- **Parallel bit deposit and extract**
  - **Pack in sequence selected bits (PEXT): extract something out**
    - ▷ $PEXT(W, Mask)$ ***returns a word by packing to the right those bits in the word*** W ***whose corresponding bits in the word*** $Mask$ ***are equal to*** 1.
    - ▷ ***Example:*** $PEXT(010110010, 010101010)$ ***extracts the four even numbered bit and then pack it to the right. Thus it returns*** 01100.
  - **Distribute bits in sequence to selected locations (PDEP): deposit something into.**
    - ▷ $PDEP(W, Mask)$ ***returns a word by sending the*** i***th bit in the word*** W ***to the location addressed by the*** i***th*** 1.
    - ▷ ***Example:*** $PEXT(01100, 010101010)$ ***deposits the four bits to the even numbered location. Thus it returns*** 010100000.

Example I

- **In Go, how to find the number of empty intersections on the board?**
  - **Assume you have a long hardware word W of 19*2=38 bits.**
    - ▷ ***Use 19 words $W_1, \ldots, W_{19}$ to represent the rows.***
  - **Encoding: bits i and $i+1$ in W_j represents the status of the intersection at the ith column and jth row.**
    - ▷ 00 ***means empty.***
    - ▷ 10 ***means a black stone.***
    - ▷ 01 ***means a white stone.***
  - **POPCOUNT(W_j) gives the number of stones in the jth row.**
  - **19-POPCOUNT(W_j) gives the number of empty intersections in the jth row.**

Example II

- **In Chinese Dark Chess (CDC), how to find all pieces of a color on the board?**
  - **Assume you have a long hardware word W of 32*3=96 bits.**
  - **Encoding: bits $3i$, $3i+1$, and $3i+2$ in W_b represents the status of the ith cell on the board with regard to black. Similarly, we have W_R for the red side.**
    - ▷ 000 ***means empty, or pieces of other color or dark.***
    - ▷ xyz ***means the xyzth kind of piece where there are up to only 7 different kinds of pieces of a color. Thus the encodings used are from 1 to 7.***
- **Algorithm Find_PCES(color c)**
  - **// find all pieces of color c**
  - $i = 0$
  - **while** $W_c \; != \; 0$ **do**
    - ▷ $a = TZCNT(W_c)$
    - ▷ $a = a \mod 3$
    - ▷ $W_c = W_c$ ***right shift*** a ***bits // find next piece***
    - ▷ $m[i++] = W_c \;\&\; 07$ ***// gives a piece of color*** c
    - ▷ $W_c \;\&= \;\; (07)$ ***// mask off the lowest 3 bits***
  - **return** m

Example III

- **In Othello, how to pack information in a column in a continuous sequence of cells?**
  - **Problem:**
    - ▷ *The board of Othello is a 8 by 8 rectangle. Assume we use a word to represent the board and use the row-major ordering, then cells in a column are non-adjacent.*
    - ▷ *Example: The first (leftmost) column are numbered 0, 8, 16, 24, 32, 40, 48, and 56 in a row-major ordering.*
  - **Encoding:**
    - ▷ *Assume you have a hardware word W of 64 bits.*
    - ▷ W_b *and* W_w *are words for black and white stones respectively.*
    - ▷ 0 *means empty or other color.*
    - ▷ $(W_b|W_w)$ *gives the word for empty spaces.*
- **Algorithm Find_Column(color c, int idx)**
  - // pack information in column idx into adjacent bits
  - // Loc is an array which gives the masks of bits in column idx
  - $Mask = Loc[idx]$
  - $W = PEXT(W_c, Mask)$
  - return W

Comments

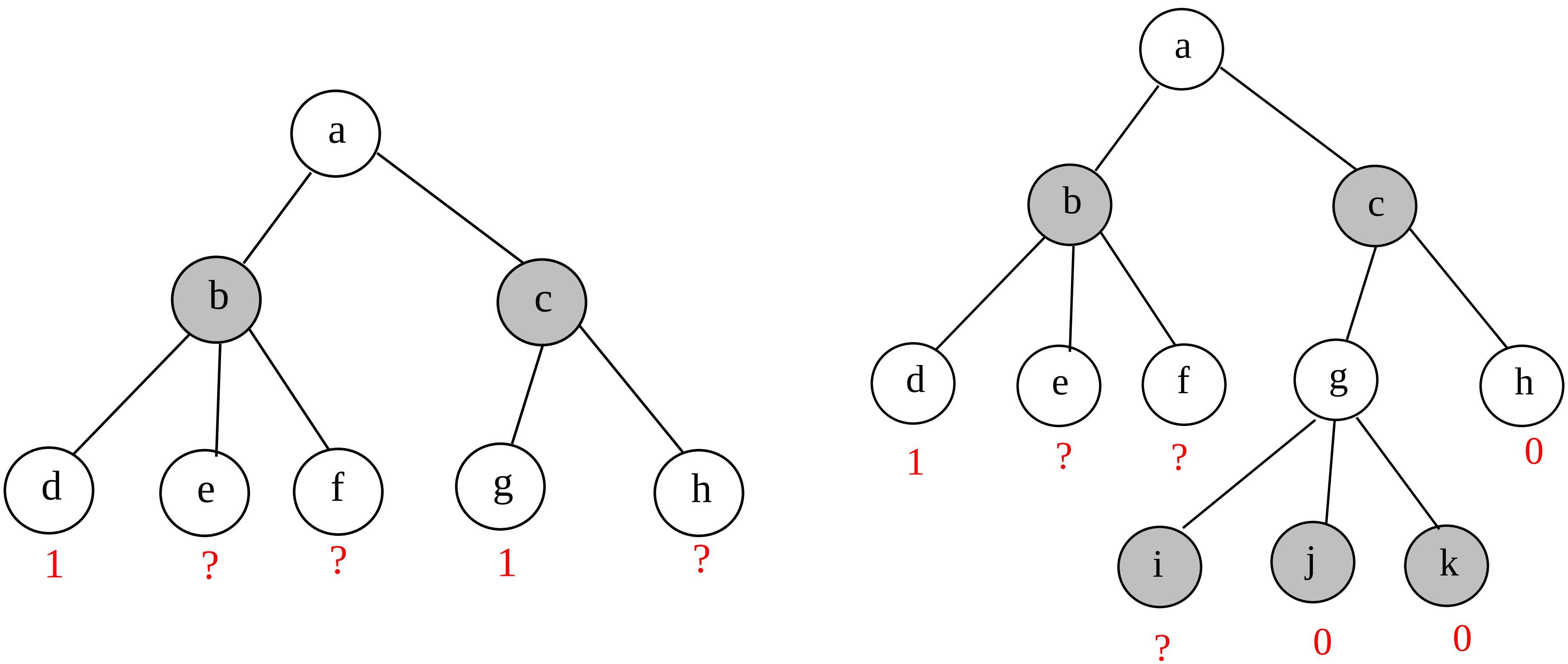
- Read carefully the instruction set of the CPU used to find out any special SIMD operations that are or aren't provided.
- The speedup is a lot, sometimes more than 50 times, if the encoding used is good [Browne 2014].

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.

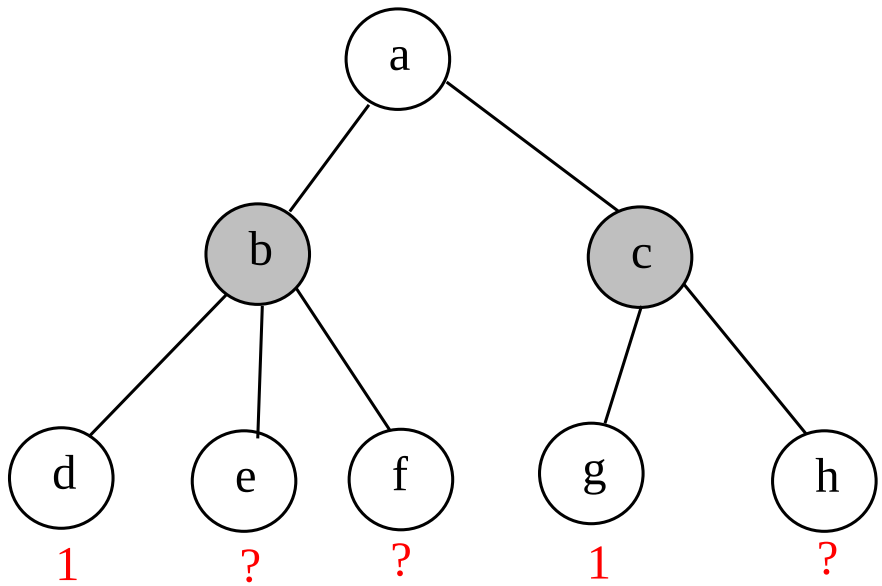
Which node to search next?

- **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.**



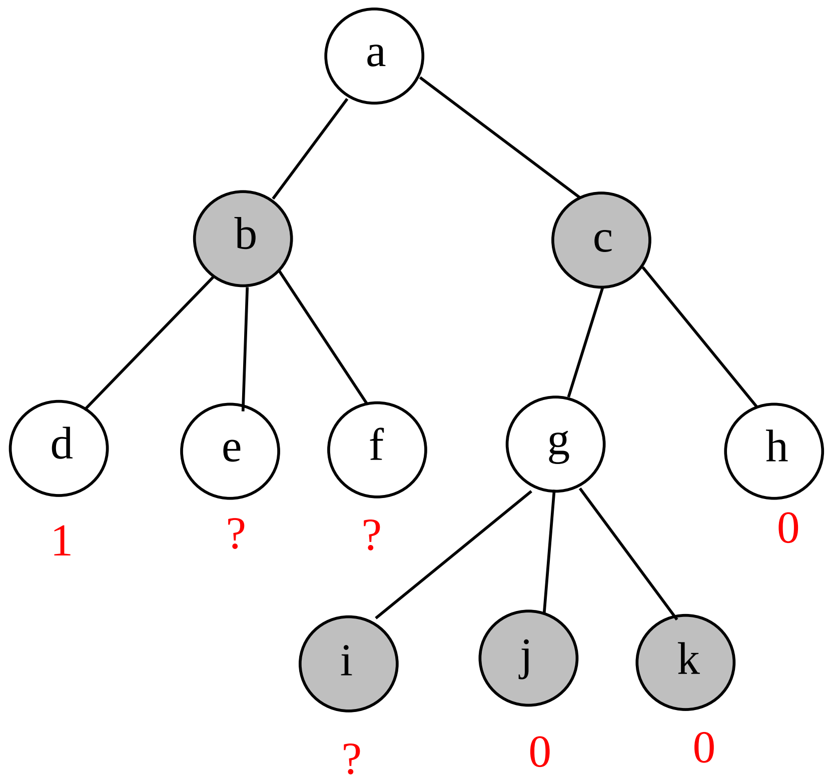
Most proving node

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



Most disproving node

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

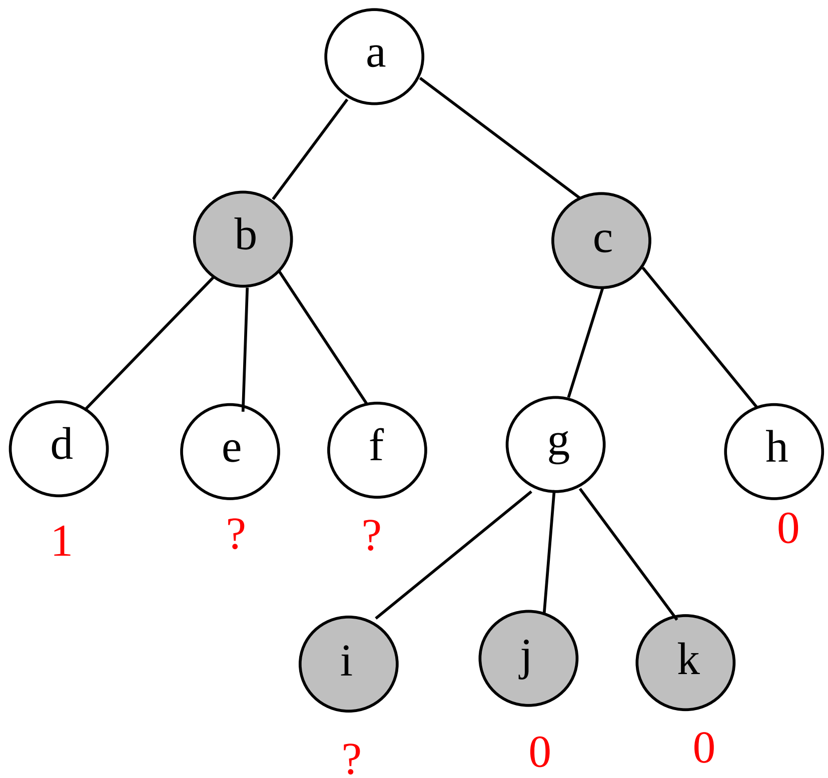


Proof or Disproof Number

- **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.**

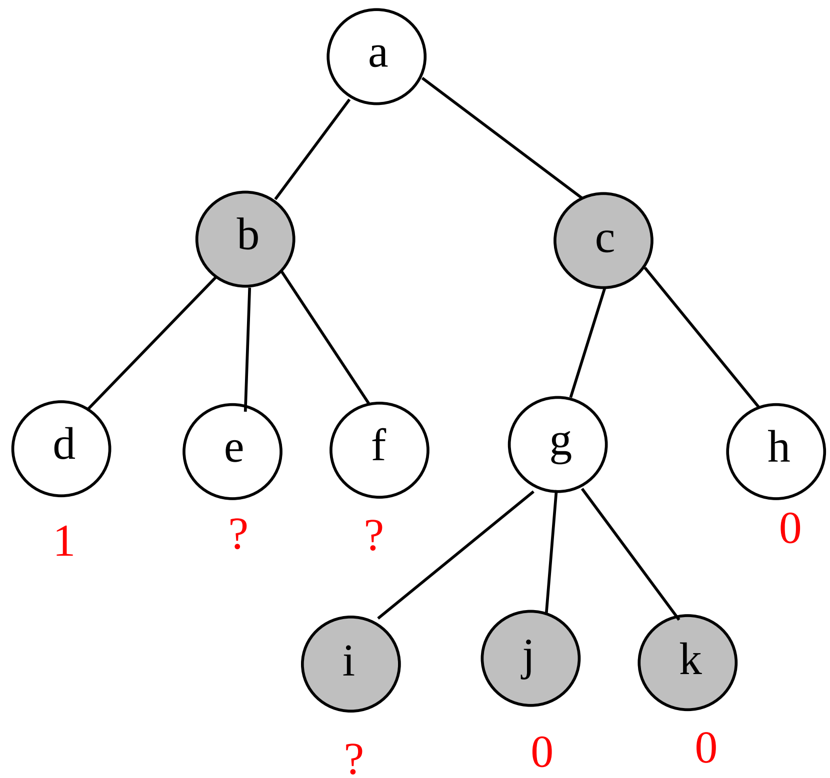
Proof number

- **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

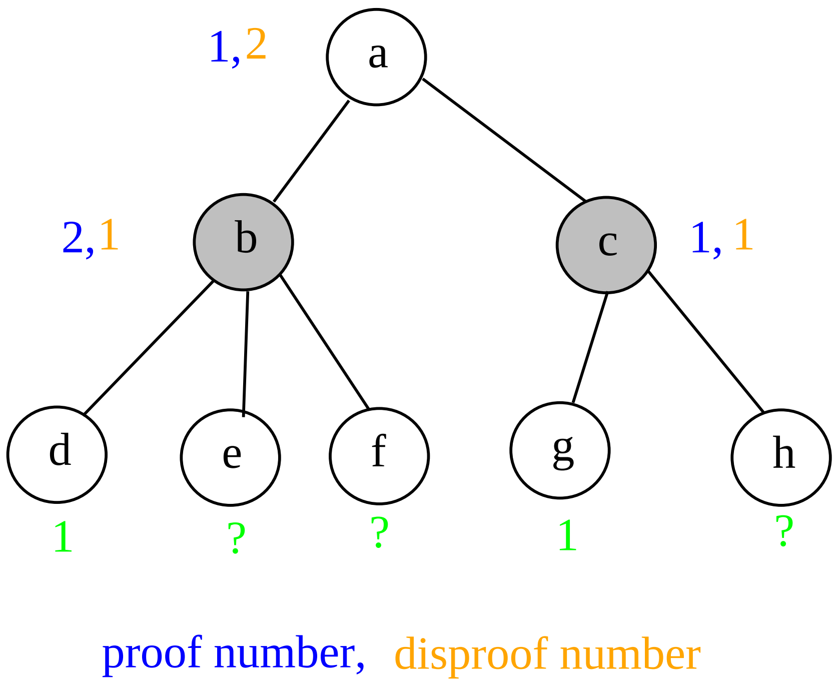
- u **is a leaf:**
  - **If** $value(u)$ **is unknown, then** $disproof(u)$ **is cost of evaluating** u**.**
  - **If** $value(u)$ **is 1, then** $disproof(u) = \infty$**.**
  - **If** $value(u)$ **is 0, then** $disproof(u) = 0$**.**
- u **is an internal node with all of the children** $u_1, \ldots,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

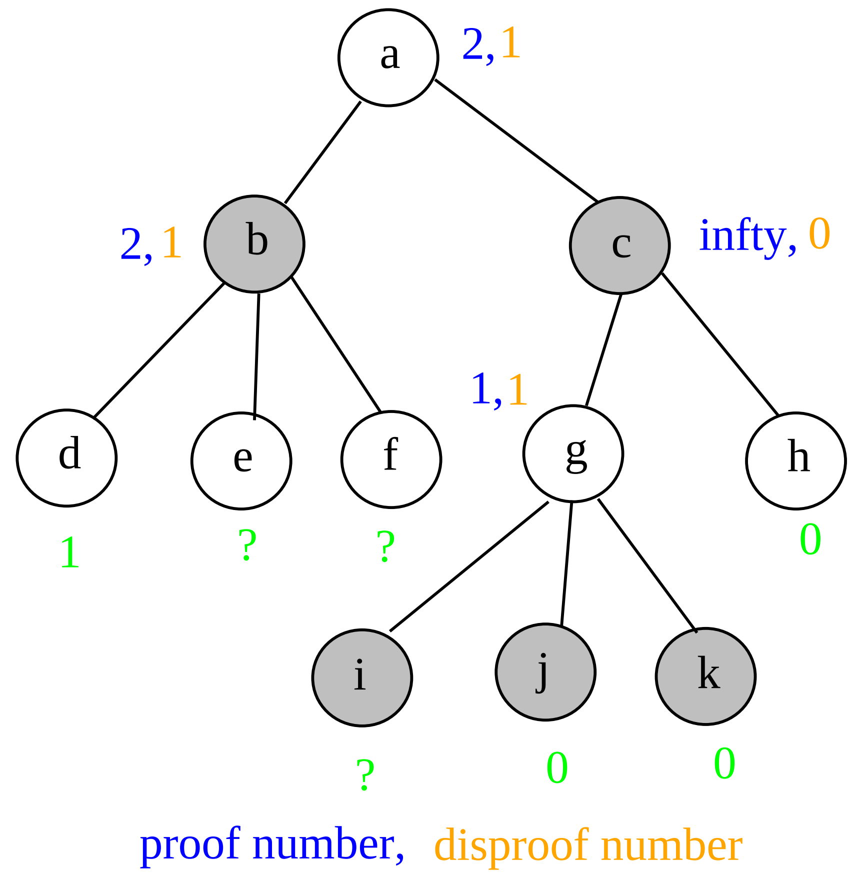
$$disproof(u) = \sum_{i=1}^{i=b} disproof(u_i);$$

  - **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)

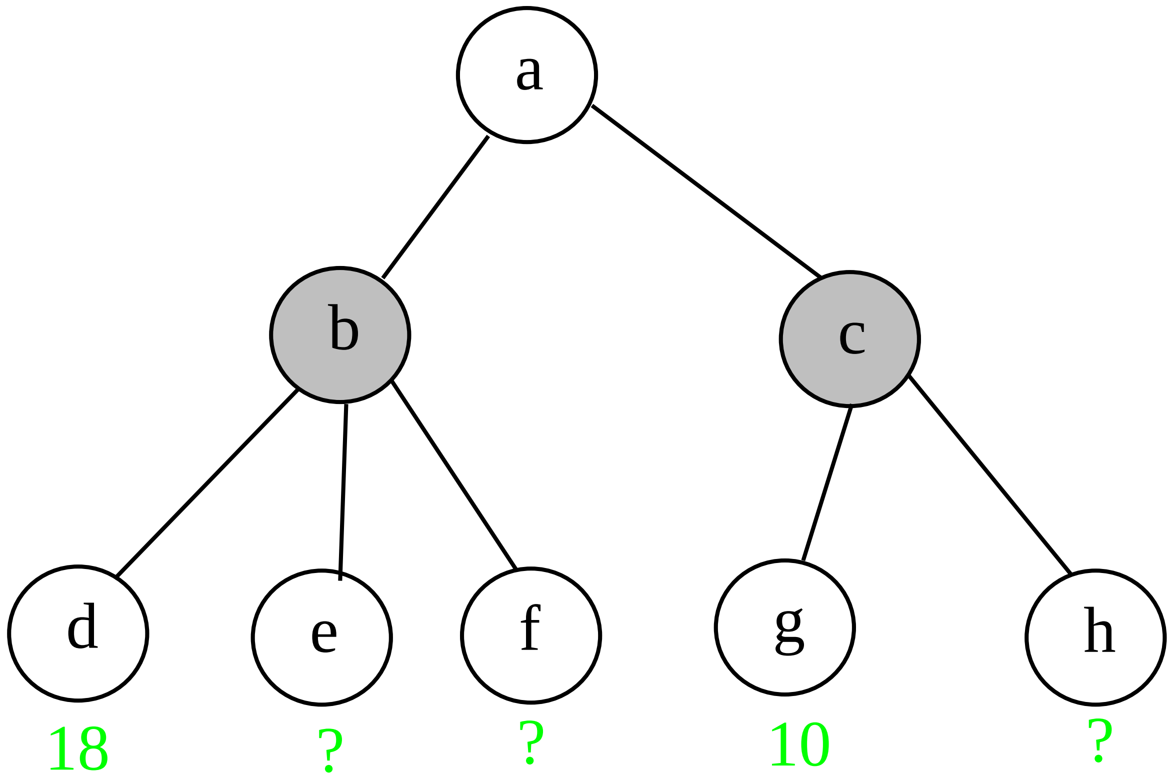
- **{∗ Continued from the last page ∗}**
  - **solve u;**
  - **repeat {∗ bottom up updating the values ∗}**
    - ▷ ***update*** $proof(u)$ ***and*** $disproof(u)$
    - ▷ $u \leftarrow u'$***s parent***

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

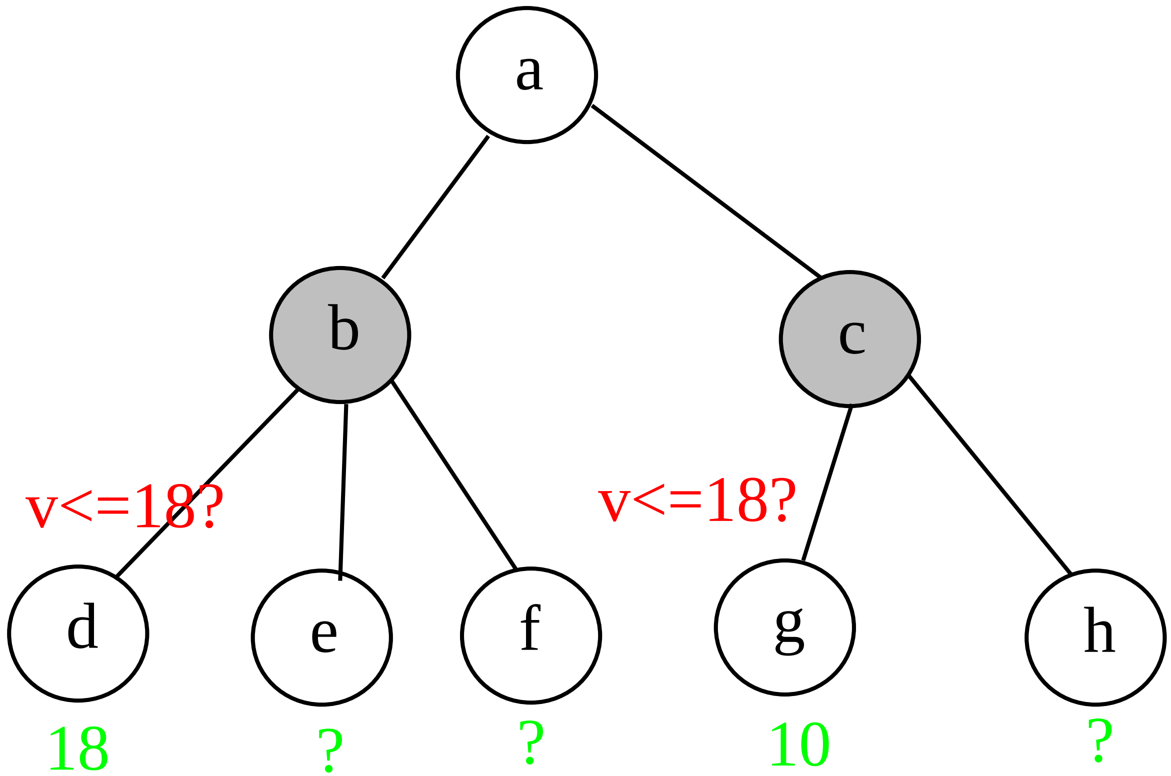
Multi-Valued game Tree

- **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

- **Does a variation 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 (1/4)

- **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 (2/4)

- C.A. Luckhardt and K.B. Irani in "An algorithmic solution of N-person games", Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86), p.158-162, AAAI Press.
- Nathan R. Sturtevan A Comparison of Algorithms for Multi-player Games Computers and Games, Third International Conference, CG 2002, Edmonton, Canada, July 25-27, 2002.
- Richard Korf "Multi-player alpha-beta pruning" in Artificial Intelligence 48 (1991), p.99-111.
- Condon, J.H. and K. Thompson, "Belle Chess Hardware", In Advances in Computer Chess 3 (ed. M.R.B.Clarke), Pergamon Press, 1982.
- Hsu, Feng-hsiung; Campbell, Murray; Hoane, A. Joseph, Jr. (1995). "Deep Blue System Overview" (PDF). Proceedings of the 9th International Conference on Supercomputing. 1995 International Conference on Supercomputing. Association for Computer Machinery. pp. 240-–244

References and further readings (3/4)

- "Chess Skill in Man and Machine", Chess 4.5 - The Northwestern University Chess Program, L. Atkin & D. Slate, pp. 82—118, Springer-Verlag, 1977.
- Fuller, S.H, Gaschnig, J.G. and Gillogly, J.J. Analysis of the Alpha-beta Pruning Algorithm Carnegie Mellon University. Computer Science Department https://books.google.com.tw/books?id=cOTmlwEACAAJ, 1973.
- C. Browne. Bitboard methods for games ICGA Journal, vol. 37, no. 2, pp. 67–84, 2014
- Intel, Intel Architecture Instruction Set Extension and Future Features Programming Reference, 2021. https://community.intel.com/legacyfs/online/drupal_files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf
- AMD, 3D Now! Technology manual, 2000. https://www.amd.com/system/files/TechDocs/21928.pdf

References and further readings (4/4)

- 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).