Computing Science (CMPUT) 657

Algorithms for Combinatorial Games

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Winter 2022

Proof-Number Search

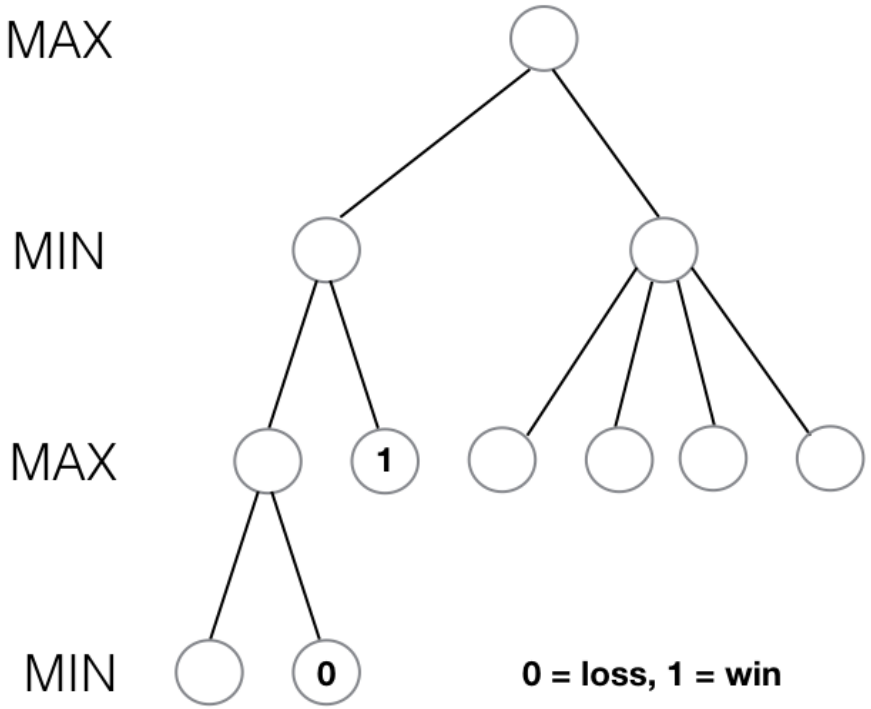
Proof-Number Search - Motivation

- Some branches are (much much) easier to prove than others
- Good move ordering helps
- Uniform-depth search (as in standard alphabeta) can be inefficient
- A deep but mostly forced line may be much easier to prove
- In many games, branching factor is far from uniform
- In many games, strongly forcing move sequences exist

Examples for Non-uniform Branching Factor

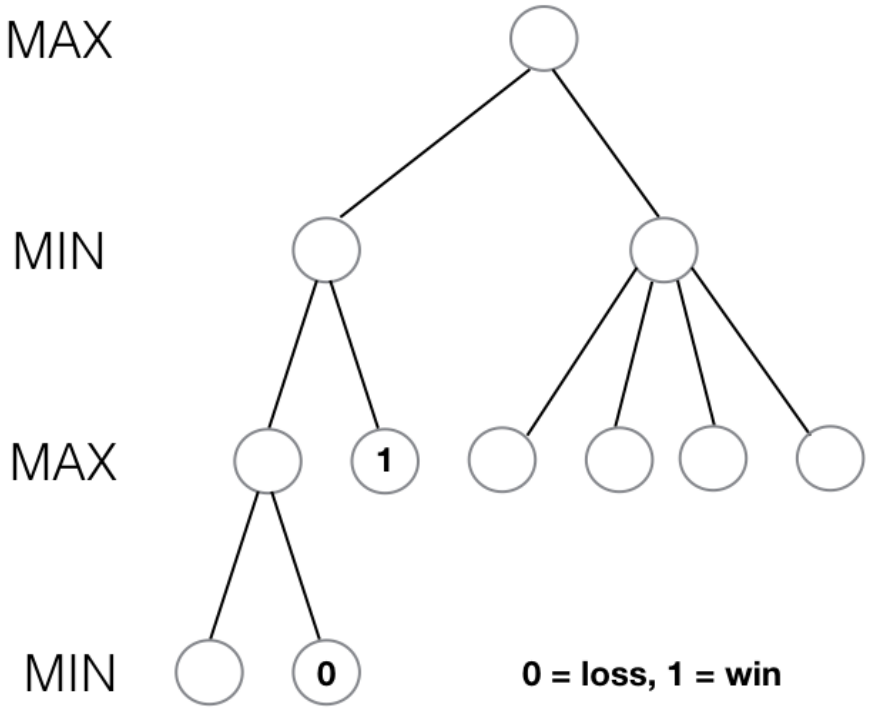
- Chess Example: king in check must escape from check
  - much reduced branching factor
  - much increased chance of finding a checkmate
- Checkers
  - must capture if possible
  - reduced branching factor, close to 1
  - captures help simplify the game - closer to endgame databases
- Go, Life and Death example
  - Often only small set of relevant attacking moves (all others will fail trivially)

Proof-Number Search Motivating Example



- Partially searched tree
- Some nodes proven as wins or losses
- Most nodes still unknown
- Where to expand next?
- Expand node that may lead to quick proof!
- Here: bottom left node

Proof-Number Search Motivating Example (2)



- Search bottom left node
- If win: parent is win, grandparent is win, ... root is win!
- If loss: (do in class)
- In both cases, solving this node is VERY useful!

Proof-Number Search (PNS)

- Invented by Victor Allis
  (AIJ paper 1994, see optional readings)
- Builds on earlier ideas by McAllester, *conspiracy numbers*
- Flexible, balanced: can find either proof or disproof
- Grow both at the same time: a potential proof and a potential disproof tree
- Incremental: grow one node at a time

Concepts - Search Tree vs Game States

- Properties of current search tree in PNS:
- Some leaf nodes may be terminal states of game, wins or losses
- All other leaf nodes have *unknown* result
  non-terminal game state, result not yet computed
- Interior node *n* in PNS search tree:
  - Originally, *n* has unknown result
  - *Expanded* later - generate children of *n*
  - Over time, some interior nodes become proven or disproven, by *propagating results* up from children
- PNS stops as soon as root is proven or disproven

Proof-Number Search Idea

- Given an incomplete (dis-)proof:
  - How far is it from being complete?
  - What is the most promising way to expand it?
- Find *(dis)proof set* of minimal size:
  - A smallest set of leaf nodes that must be (dis-)proven to (dis-)prove the root.

Proof-Number Search Idea (2)

- Principle: optimism in face of uncertainty
  (seen also in Monte Carlo tree search)
- Assume cost of proving each unproven node is 1
  (this is optimistic, lower bound)
- Try to complete a proof:
  reduce size of smallest proof set to 0
- Same mechanism for disproof, disproof set
- Main idea:
  - There is always a node n in both the min. proof and disproof set
  - Expand it!

Most-Promising Node

- Key insight of PNS: there is always a *most-promising node (MPN)*
  Allis called it most-proving node
- MPN is in the intersection of
  - a minimal proof set **and**
  - a minimal disproof set
- Solving MPN will help either a proof or a disproof of the root:
  - proving MPN reduces min. proof set of root
  - disproving MPN reduces min. dispproof set of root

Proof and Disproof Number of a Node

- Proof number *pn* of a node *n*:
  - size of min. proof set for *n*
  - Optimistic estimate of cost of proving *n*
- Disproof number *dn* of *n*:
  - size of min. disproof set for *n*
  - Optimistic estimate of cost of disproving *n*

PNS Algorithm Outline

- Initialise tree with just the root
  - Set *pn* and *dn* of root
- Repeat until root proven or disproven:
  - Find MPN
  - Expand MPN
  - Recompute proof and disproof numbers

Initialize (Dis-)Proof numbers at Leaf Nodes

- Notation:
  - $n.pn$ = proof number of n
  - $n.dn$ = disproof number of n
- Leaf node, not terminal: $n.pn = n.dn = 1$
- Leaf node, win: $n.pn = 0, n.dn = \infty$
- Leaf node, loss: $n.pn = \infty, n.dn = 0$

(Dis-)Proof numbers at Interior Nodes

- Back to basics: to prove win,
  must prove one child at OR-node,
  all children at AND-nodes
- Assume we have proof and disproof numbers of children
- Set *pn* of OR node to *minimum pn* of all children
- Set *dn* to *sum* of *dn* of all children
- AND node is dual: *pn* is sum, *dn* is minimum of children's numbers

Basic Assumptions of Proof Number Search

- Assumption 1: optimism works (see earlier discussion)
- Assumption 2: solving each subtree of a node is independent from solving the others (therefore the sum)
  - True if state space is a tree
  - Can be *very* wrong in a DAG

Basic Proof Number Formulas

- OR node:

$$n.pn = \min_{c \in children(n)} c.pn \qquad n.dn = \sum_{c \in children(n)} c.dn$$

- AND node:

$$n.pn = \sum_{c \in children(n)} c.pn \qquad n.dn = \min_{c \in children(n)} c.dn$$

- Note: Infinities $\pm\infty$ behave as expected, e.g. $\infty + 1 = \infty + \infty = \infty$, $\min(c, \infty) = c$

Wins and Losses

- n is proven win:
  $n.pn = 0, n.dn = \infty$
- n is proven loss:
  $n.pn = \infty, n.dn = 0$
- Wins and losses back up as expected, e.g.
- Child c_i of OR node is win:

$$n.pn = \min(c_1.pn, ..., c_i.pn, ...) = \min(..., 0, ...) = 0$$

$$n.dn = \sum(c_1.dn, ..., c_i.dn, ...) = \sum(..., \infty, ...) = \infty$$

- Similarly for AND nodes, for losses

PNS Algorithm


```
ProofType PNS(Node root)
root.InitializePnDn()
while (root.pn != 0 AND root.dn != 0
        AND ResourcesAvailable())
    Node mpn = SelectMPN(root)
    ExpandNode(mpn)
    UpdateProofNumbers(mpn)
if (root.pn == 0)
    return PROVEN
else if (root.dn == 0)
    return DISPROVEN
else
    return UNKNOWN
```

Select Most Promising Node


```
Node SelectMPN(Node node)
while (NOT node.IsTerminal())
    if (node.Type() == OR_NODE)
        node = FindEqualChildPN(node.children,
                                node.pn)
    else
        node = FindEqualChildDN(node.children,
                                node.dn)
return node

Node FindEqualChildPN(NodeList nodes, int parent_pn)
forall (c in nodes)
    if (c.pn == parent_pn)
        return c

Node FindEqualChildDN(NodeList nodes, int parent_dn)
(same, replace pn by dn)
```

Expand Node


```
ExpandNode(Node node)
forall (legal moves m from node)
    Node c = node.Play(m)
    c.InitializePnDn()
    node.AddChild(c)
```

Update Proof Numbers


```
// updates are bottom-up starting with MPN
UpdateProofNumbers(Node node)
    if (node.Type() == OR_NODE)
        node.pn = min_pn(node.children)
        node.dn = sum_dn(node.children)
    else
        node.pn = sum_pn(node.children)
        node.dn = min_dn(node.children)
    if (node.parent)
        UpdateProofNumbers(node.parent)
    // Eliminate tail-end recursion:
    // use while loop as in the paper
    // Optimization: stop recursion
    // as soon as a node does not change,
    // restart SelectMPN from there
```

Comments on PNS

- “Best-first”, great for unbalanced search trees
- Adapts to find deep but narrow proofs
- Memory hog - needs to store all nodes in memory (df-pn is better)
- No guarantee on finding short win or small proof tree - ignores cost of proof so far
- Behaves more like greedy best-first search in single-agent search than like A*
- There is AO*, an equivalent to A* for finding least-cost solutions in AND/OR trees. We will not discuss it (some info in optional resources)

Successes of PNS

- Victor Allis solved connect-four, qubic, Go-moku (5 in a row)
- Used in endgame solvers e.g. for Awari, Lines of Action
- Depth-first reformulation: df-pn
  - Used in proof of checkers (our reading)
  - Use to solve very hard checkmating problems in shogi
  - Use to solve very hard life and death problems in Go

PNS on a DAG

- Still well-defined (apply formulas bottom-up)
- Problem 1: node has more than one parent
- Problem 2: overcounting proof and disproof numbers

PNS on a DAG - Multiple parents

Problem 1: node has more than one parent

- Backup to all? Cost can explode, go from $\Theta(\log n)$ to $\Theta(n)$ per iteration
- Backup to one parent only? Then values become inconsistent (other parents out of sync) and MPN computation becoms flawed
- Still, we usually accept the single parent backup
  - repair other parents' values if and when they are revisited
- **Research question:** is there a better way?

PNS on a DAG - Overcounting

Problem 2: overcounting proof and disproof numbers

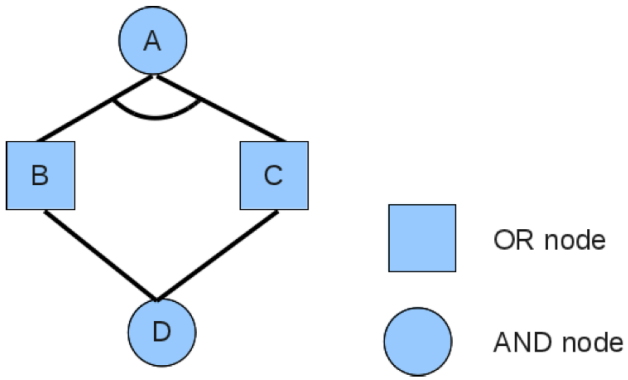
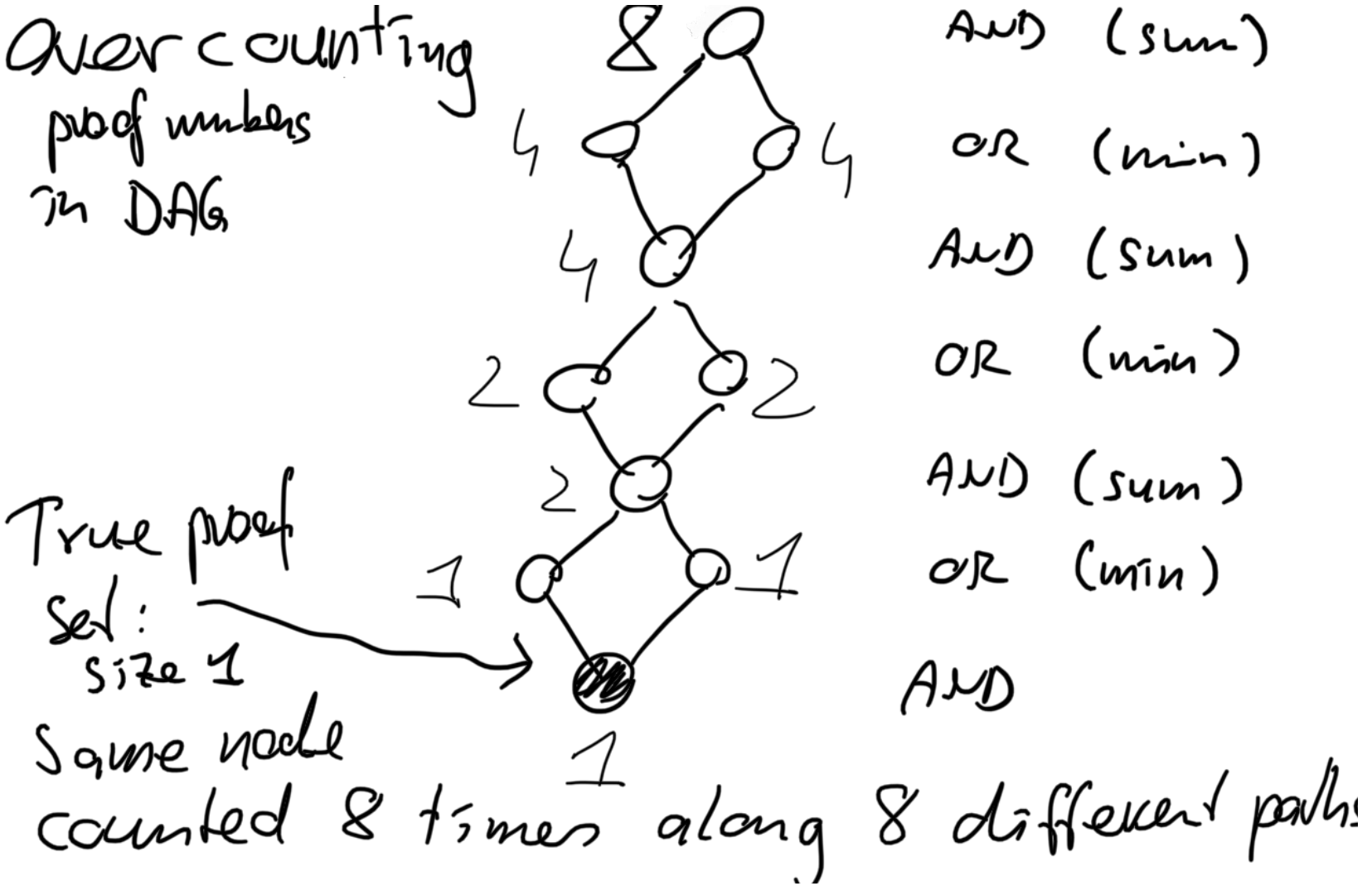


Figure 7: Example of overestimating pn(A)

- Back to basics: pn, dn count number of leaf nodes that must be solved
- In DAG, the same leaf node may be counted along multiple paths
- This overcounting can be exponentially bad (do an example)
- Effect: an easy to prove node can look very hard
- It happens in practice!
- How to fix? see Section 6 of paper

Overcounting Can Get Arbitrarily Bad



Proof Numbers - Heuristic Initialisation

- Heuristic initialisation of pn, dn
- Back to basics:
  pn, dn are lower bounds on cost of solving node
- Initializing them with 1 is naive
- Are pn and dn inversely related? Not really. Discuss.
- Heuristic initialisation is a huge improvement in practice

Heuristic Initialisation - Examples

- How to find better estimates?
- Idea: use features of the position, domain knowledge
- Chess and shogi: estimate king safety
- Chess and shogi: count attacking pieces
- Go: “distance to life” in life and death problems
- Huge improvement in these tasks

Example: Pseudo One Move Lookahead initialisation

- What happens if we expand a node with k children?
- before expand: pn = dn = 1 in both AND and OR nodes
- Assume no wins/losses among children
- After expansion:
  - OR node: pn = 1, dn = k
  - AND node: dn = 1, pn = k
- In some games, it is cheap to compute or approximate the number of children k
  - Example in Go: number of empty points is a good approximation
- Initialize dn of OR, pn of AND with (estimate of) k
- How does this compare/combine with other heuristic initialisations?

PNS - Extension to More than Two Game Results

- PNS is defined for the two results case (win/loss)
- Can extend as in boolean minimax
  - Series of boolean searches, e.g. binary search or sequential search
  - Each search divides results into “good” and “bad” groups as in minimax
- Other approaches: see Section 9 of survey paper

Checkers Case Study

- Checkers solution (Schaeffer 2007)
- Uses *seed* for proof tree:
  strongest lines proposed by human experts
- Pseudo-proof:
  - assume everything with evaluation > 150 is win
  - assume everything < -150 is loss
  - Create “proof tree”
  - For “win”, all leaf nodes in proof have eval > 150
- After 150 is proven, change bounds to ± 200, re-search
- Keep increasing bounds to 250, etc.
- Once bounds reach $\pm\infty$, proof is complete

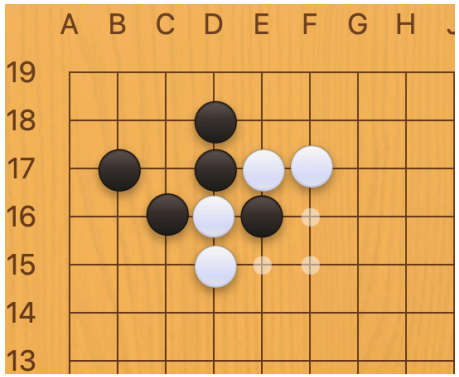
Good Cases for PNS

- Uneven branching factor
- Early wins/losses found in some branches
- Number of moves correlated with winning chance
- Example: checkmating problems
  - If defender has fewer moves it may be trouble
  - King in check: only a few moves
  - Search of these critical positions can become very deep very quickly

Bad Cases for PNS

- “Everything looks the same”
- Uniform branching factor, no early wins/losses
- PNS becomes an expensive way to do blind search

Really Bad Cases for PNS



- Proof numbers can be actively misleading
- Example: short proof needs set of 3 nodes, but there exists a huge subtree with $pn \leq 2$.
- Lots of “forcing moves”, but they don’t work. Only a “quiet” move works
- Example in Go:
  - branching ladder, all branches fail, can be hundreds of nodes with $pn = 2$, search depth 100 or more
  - Capture locally in *net*: $pn > 2$, but depth < 10

Negamax Formulation of PNS

- Used in df-pn algorithm
- Proof numbers for current player in each node
- Only one set of formulas
- Define:
  - $n.\phi = n.pn$ in OR node, $n.dn$ in AND node
  - $n.\delta = n.dn$ in OR node, $n.pn$ in AND node
- Formula: $n.\phi = \min(c_1.\delta, ..., c_n.\delta)$
- Formula: $n.\delta = \sum(c_1.\phi, ..., c_n.\phi)$
- Do you see how it is equivalent to normal PNS?

Depth-First Proof-Number Search (df-pn)

Depth-First Proof-Number Search (df-pn)

- Depth-first version of proof-number search (Section 4 in survey)
- Developed in (Nagai 1999; 2002)
- Reduces re-expansions of interior nodes
- Uses transposition table for memory
- Works well even with severely limited memory
- df-pn(r) (Kishimoto and Müller 2003, Kishimoto 2005) improves df-pn in case of repetitions (e.g. checkers, ko in Go)

Main Contributions of Df-pn

- Efficiency
- Good use of memory
- Multiple iterative deepening at interior nodes
- Extensions to address double-counting in DAG's
- Another extension: df-pn+ algorithm
  - adds cost function for edges
  - can find optimal solutions in AND/OR graphs with costs

Main Idea of Df-pn

- In PNS, search often stays in one subtree for a long time
- PNS repeatedly expands MPN until tree solved as win or loss
- Main idea:
  - As long as we can determine the MPN . . .
  - . . . we do not need to compute proof and disproof numbers exactly
- We can compute thresholds for how long we stay in a subtree
- We can use them to delay updates and gain efficiency

Example of Delayed Updates

- Example: compute pn from children
  n.pn = min(100, 90, **20**, 60, *50*, 75) = 20
- Initially, we will stay in the subtree with pn = 20
- How long? Until its proof number *exceeds* $pn_2 = 50$
- Why 50? Smallest proof number among other children
- We must also check if a child selection would change higher up in the tree.
- How? Can pass down a threshold *pt* from parent
- Formula for new threshold: $\min(parent.pt, pn_2 + 1)$

Main Idea of Df-pn for Sum Computation

- n.pn = sum(c1.pn,...cn.pn)
- Assume we have threshold n.pt for node n
- Say we are working on child c_i. How long?
- Answer:
  until either $n.pn \geq n.pt$,
  or the increase in c_i exceeds the difference $n.pt - n.pn$.
- Set threshold of child
  $c_i.pt = c_i.pn + (n.pt - n.pn)$

Df-pn Negamax Formulation

- $n.\phi$ proof number for *current* player
- $n.\delta$ disproof number for *current* player
- Formula: $n.\phi = \min(c_1.\delta, ..., c_n.\delta)$
- Formula: $n.\delta = \sum(c_1.\phi, ..., c_n.\phi)$
  - $n.\phi$ defined in terms of children's δ
  - $n.\delta$ defined in terms of children's ϕ
- Write thresholds in these terms: see code

Equivalence Between PNS and df-pn

- Theorem 2.3 in Nagai's thesis:
  df-pn always expands an MPN
- Comment: there may be multiple MPN -
  PNS and df-pn may do tiebreaking differently
- See Nagai's thesis for detailed proof

Multiple Iterative Deepening (MID)

- Iterative “deepening” in each node, not just root
- “Deepening” here means increase threshold, not depth
- Reason:
  - (dis-)proof numbers can decrease as well as increase
  - Example: some children of an AND node are proven $\longrightarrow$ sum decreases

Df-pn Pseudocode

- Nagai's thesis, Appendix A, or PNS survey paper Figure 5
- Calls MID for root with thresholds $\pm\infty$
- Uses transposition table

Df-pn+

- Two Ideas:
- Heuristic initialization of leaf nodes
  - Measure difficulty of (dis-)proof of leaf
- Edge costs
  - Measure “desirability” of exploring this move
  - High edge costs: favor shallow trees, breadth-first like
  - Low edge cost: favor depth-first

Performance with Low Memory

- Can run with (incredibly) little memory
- Efficient pruning techniques (Nagai): SmallTreeGC and SmallTreeReplacement
- SmallTreeGC: garbage collect nodes with small subtrees
- SmallTreeReplacement: Hashing with open addressing, try multiple entries (e.g. 10), replace one with smallest subtree
- Alternative: hashing with chaining - store more than one entry at one location

Summary of PNS and Df-pn

- Algorithms specialized for finding quick proofs
- Work well for unbalanced trees with deep forcing lines
- PNS is memory hungry
- Df-pn is an efficient alternative
- Many enhancements
- Serious problem: overcounting nodes in DAG
- For proving games, these are often the best choices