# An Update on Game Tree Research

Akihiro Kishimoto and Martin Mueller

# Tutorial 4: Proof-Number Search Algorithms

Presenter:
Akihiro Kishimoto, IBM Research - Ireland

# Overview of this Talk

- Techniques to solve games/game positions with AND/OR tree search algorithms using proof and disproof numbers
    - Proof-number search
    - Depth-first proof-number search
    - Issues and enhancements
    - Parallelism
    - Multi-valued scenario
    - Applications

# Proof-Number Search - Motivation

- Some branches are much easier to prove than others
- Good move ordering helps
- Uniform-depth search (as in alpha-beta) is a problem: deep but mostly forced line may be much easier to prove
- Branching factor is far from uniform in many games
    - Chess and shogi: 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 & helps simplify games
    - Life and Death in Go: stones close to life – can compute small set of relevant attacking moves

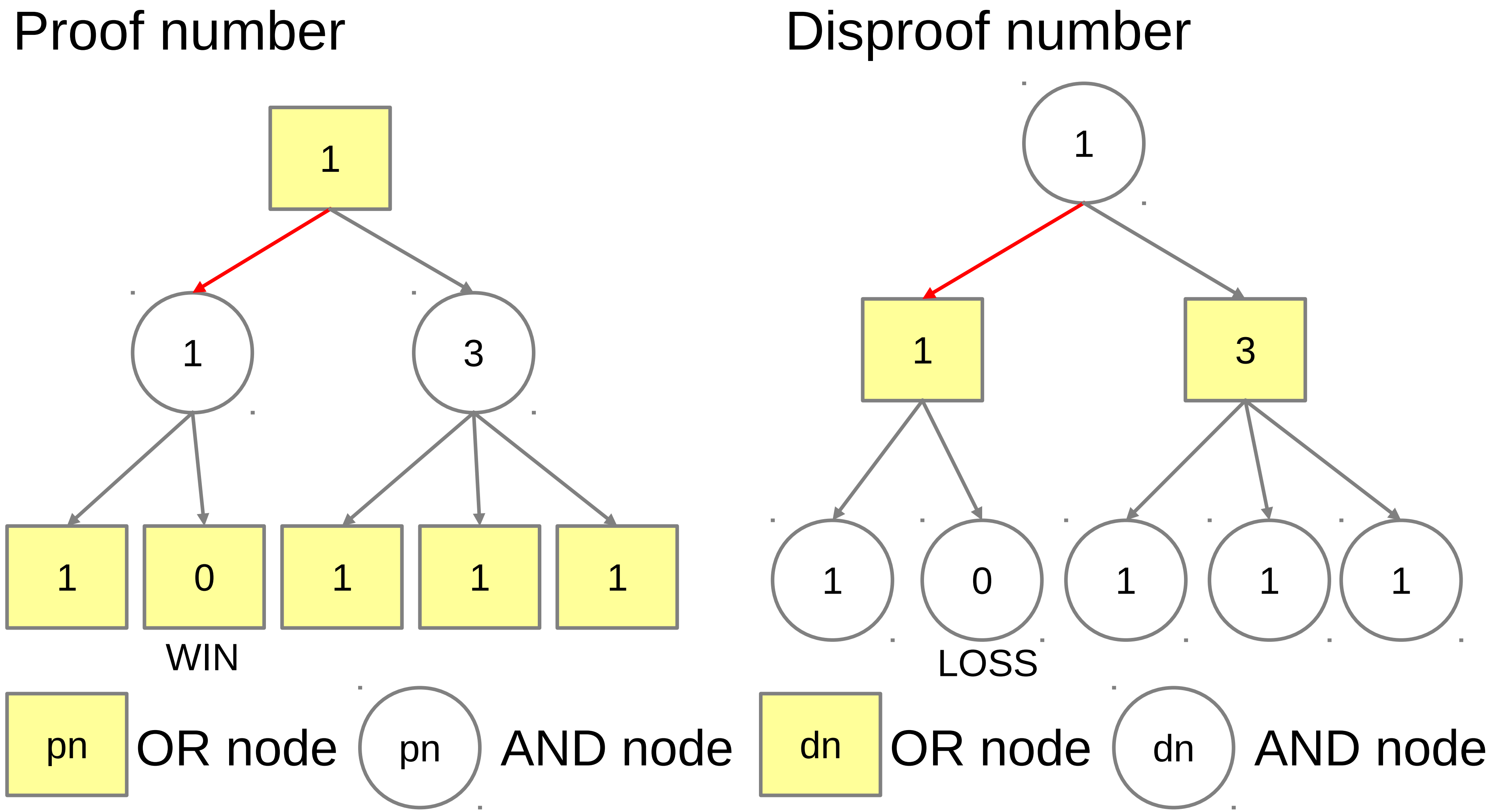
# Proof-Number Search (1 / 2) [Allis et al, 1994]

- Builds on earlier ideas of *conspiracy numbers* [McAllester, 1988]
- Flexible, balanced: can find either proof or disproof
- Grow both a proof and a disproof tree at the same time, one node at a time
- Some leaf nodes will be (dis-)proven, many others will be unknown – interior state, game result not known
- Stop as soon as root is proven or disproven
- Given an incomplete (dis-)proof: how far is it from being complete? What is the most promising way to expand it?

# Proof-Number Search (2 / 2)

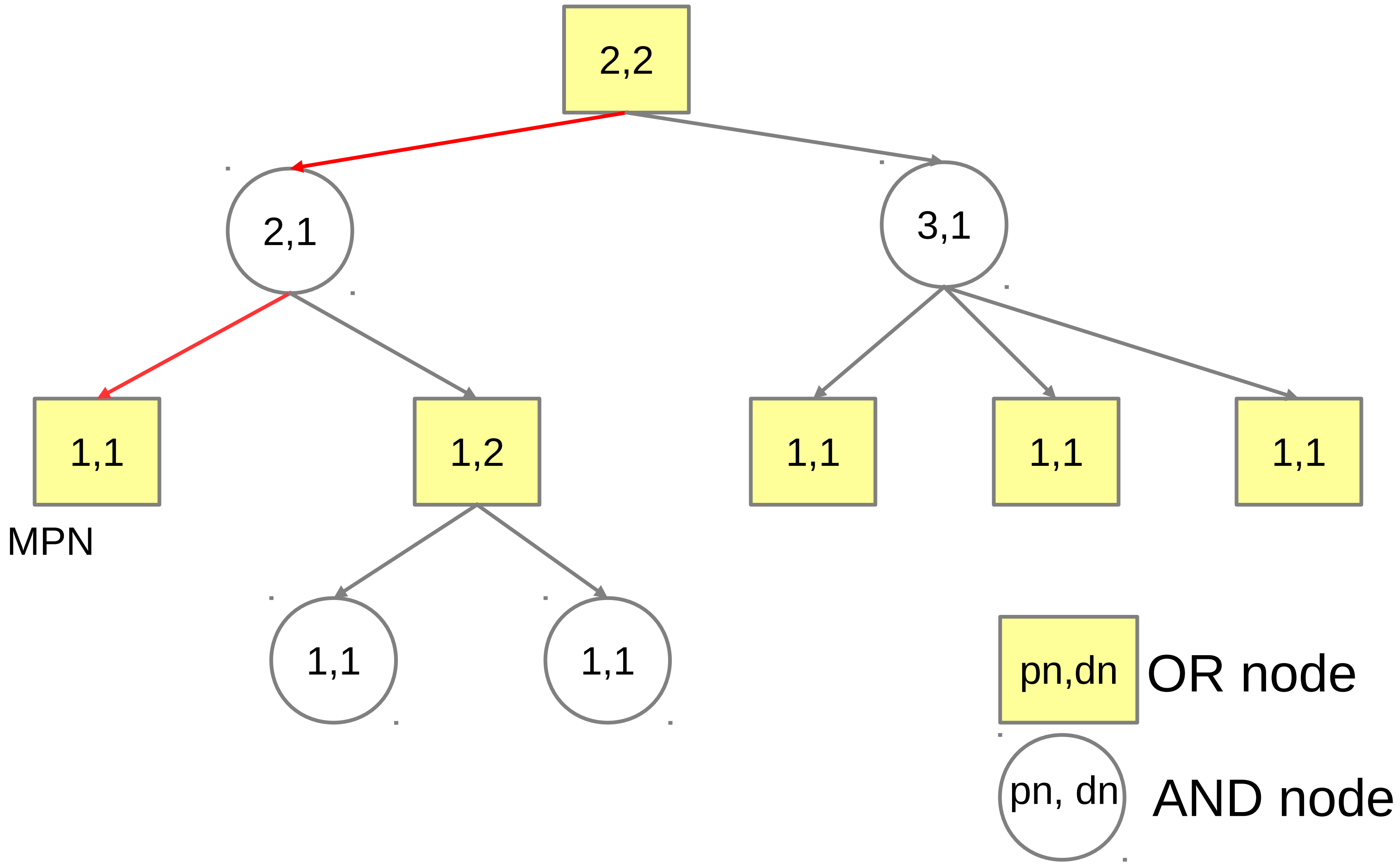
- Find (dis-)proof set of minimal size: set of leaf nodes that must be (dis-)proven to (dis-)prove root
- Principle: optimism in face of uncertainty
- Assume cost of proving each unproven node is 1 (this will be enhanced later)
- Complete proof: reduce size of smallest proof set to 0 (same for disproof)
- Main idea: always expand nodes from min. proof and disproof set

# Example of Proof and Disproof Numbers



# Most-Promising Node (aka Most Proving Nodes)

Example (C.f. [Kishimoto et al, 2012])



# Key Insight of PNS

- There is always a most-promising node (MPN)
  - If search space is tree
  - Discuss issues for directed acyclic/cyclic graphs later
- Solving MPN will help either a proof or disproof: proving it reduces min. proof set, while disproving it reduces min. disproof set of the root

# PNS Algorithm Outline (1 / 2)

- Notation: pn(n) = proof number of node n

  dn(n) = disproof number of node n
- Non-terminal leaf: pn(n) = dn(n) = 1
- Terminal node, win: pn(n)=0, dn(n) = INF
- Terminal node, loss: pn(n) = INF, dn(n) = 0
- Interior OR node: pn(n)=min(pn(c1),...,pn(ck))

  dn(n)=dn(c1) + … + dn(ck)
- Interior AND node: pn(n) = pn(c1)+ … + pn(ck)

  dn(n)=min(dn(c1), …,dn(ck))

  c1,...,ck: n's children

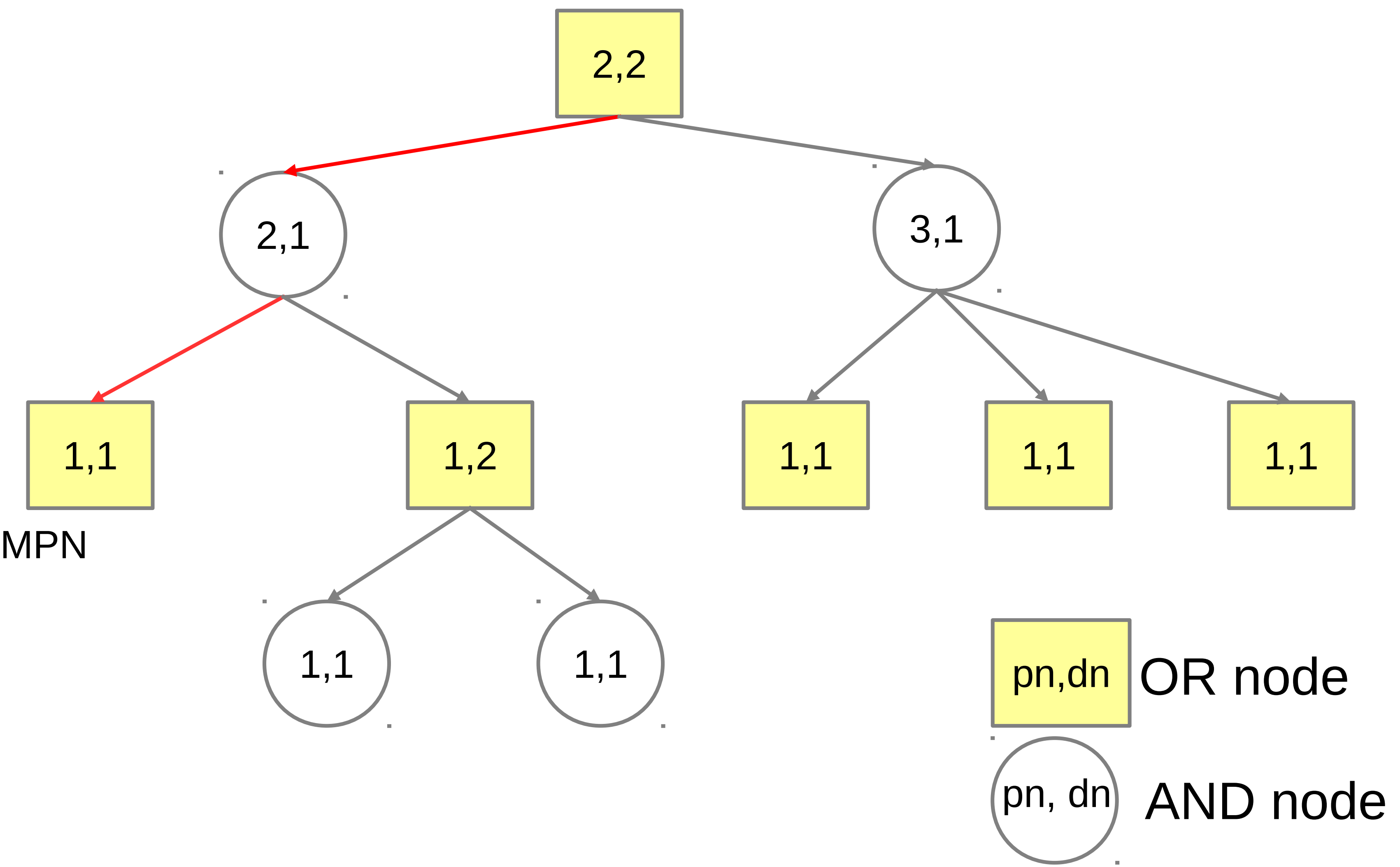
(Big) Assumption: solving subtrees are independent tasks

# PNS Algorithm Outline (2 / 2)

a) Start from root and find MPN
b) Expand MPN
c) Recompute proof and disproof numbers of the nodes on the path from root to MPN
d) Repeat until root proven or disproven

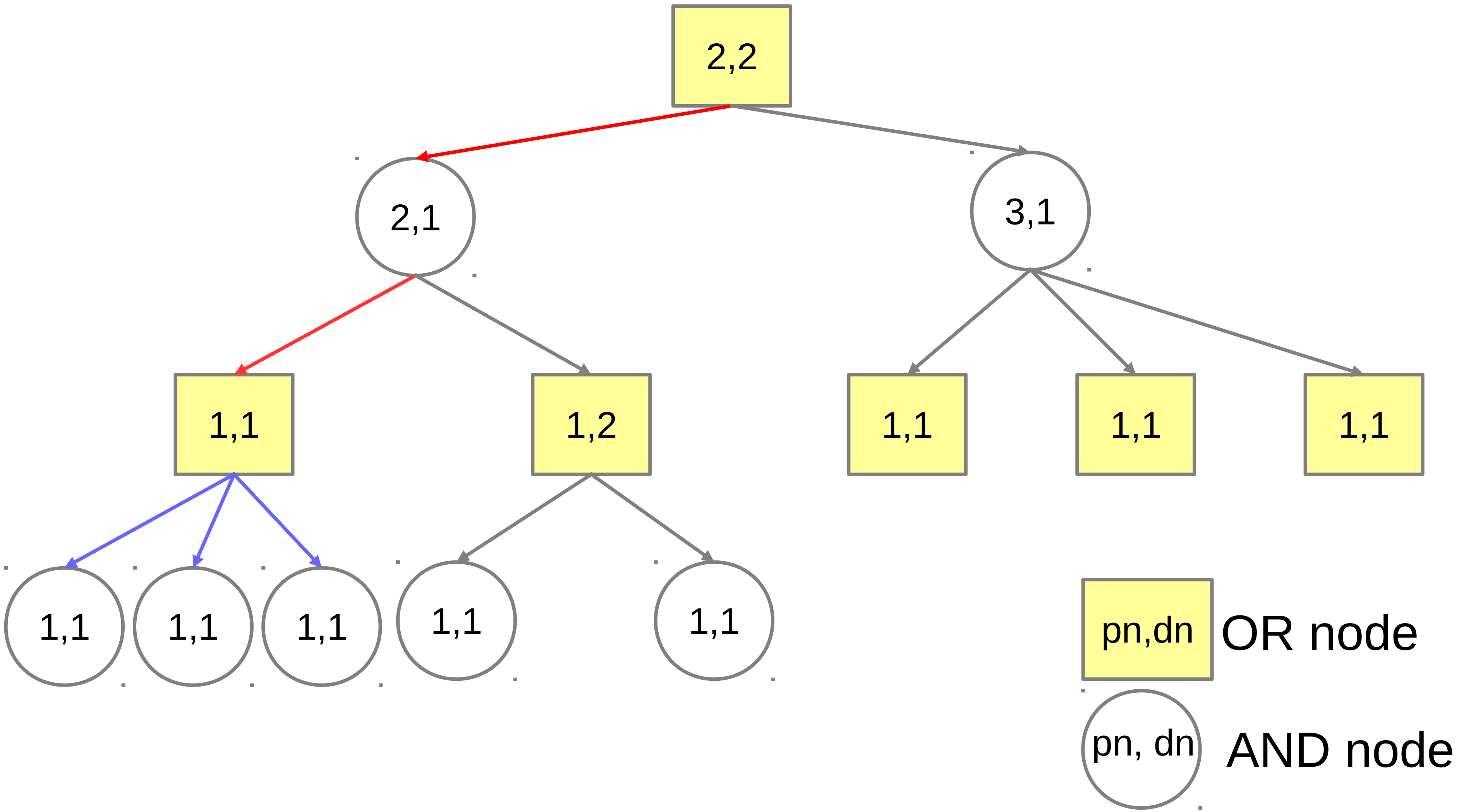
# Example of PNS (1 / 4)

MPN selection



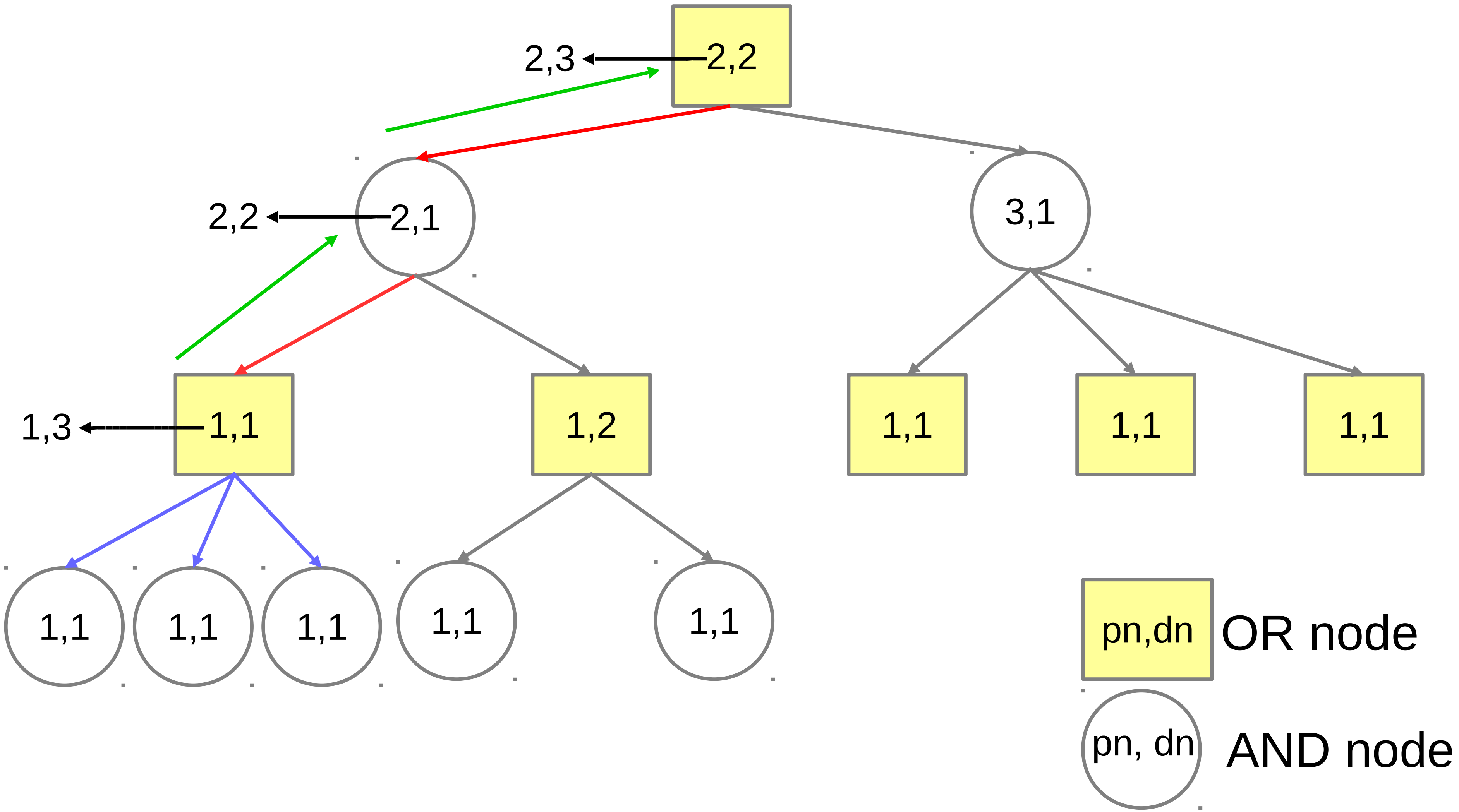
# Example of PNS (2 / 4)

MPN expansion



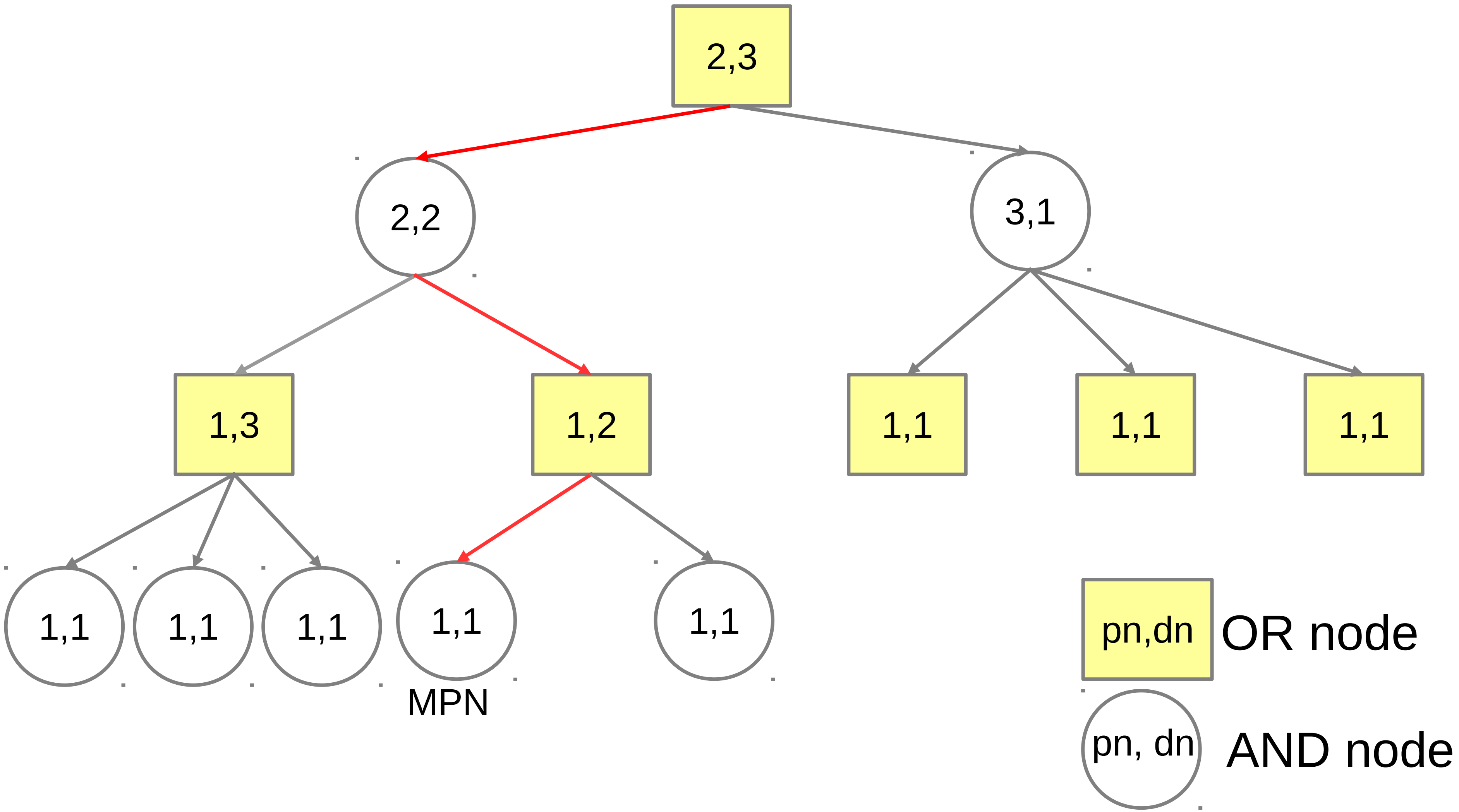
# Example of PNS (3 / 4)

Back propagation of proof and disproof numbers



# Example of PNS (4 / 4)

MPN selection



# 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
- Non-negligible Interior node re-expansion (depth-first proof-number search is better)
- No guarantee on finding short win or small proof tree – ignores cost of proof so far
- Behaves more like “pure heuristic search” in single-agent search than like optimal A*

# Reducing Memory Usage (1 / 2)

- PN² Search [Allis, 1994]
  - Perform two levels of proof-number search

  a) Run one step of PNS

  b) Run another, limited PNS to evaluate leaf nodes
    - E.g. Limit to $1/\left(1+e^{(a-x)/b}\right)$ where *x* is the tree size of first search and *a* and *b* are empirically tuned parameters [Breuker, 98]

  c) Throw away the second search (wasteful?)

  d) Repeat a)

# Reducing Memory Usage (2 / 2)

- Transposition table + efficient pruning techniques to discard least useful existing TT entries when TT is filled up
    - SmallTreeGC: garbage collect nodes with small subtrees [Nagai, 1999]
    - SmallTreeReplacement: hashing with open addressing, try multiple entries (e.g. 10), replace one with smallest subtree [Nagai, 2002]
    - Alternative: hashing with chaining – store more than one entry at one location
    - Can run with (incredibly) little memory
    - Can be combined with PNS, but typically combined with depth-first proof-number search (df-pn)

Remains an open question which performs better, $PN^2$ or TT+SmallTreeGC?

# Depth-First Proof-Number Search [Nagai, 2002]

- Basic PNS always propagates proof and disproof numbers of leaf all the way back to root
- Incurs high overhead to expand new leaf
  - E.g. Expanding only one new leaf that is 100 steps away from root requires to re-expand 100 internal nodes
- Df-pn significantly reduces node re-expansion overhead
  - Uses thresholds of proof and disproof numbers to control search

    C.f. Korf's Recursive Best-First Search for single-agent search
  - Uses transposition table to save previous search effort
  - Empirically ratio of re-expansion is about 30% in Go/shogi
- Df-pn finds MPN as basic PNS does
  - If search space is tree

# Main Idea of Df-pn's Threshold Controlling Techniques (1 / 2)

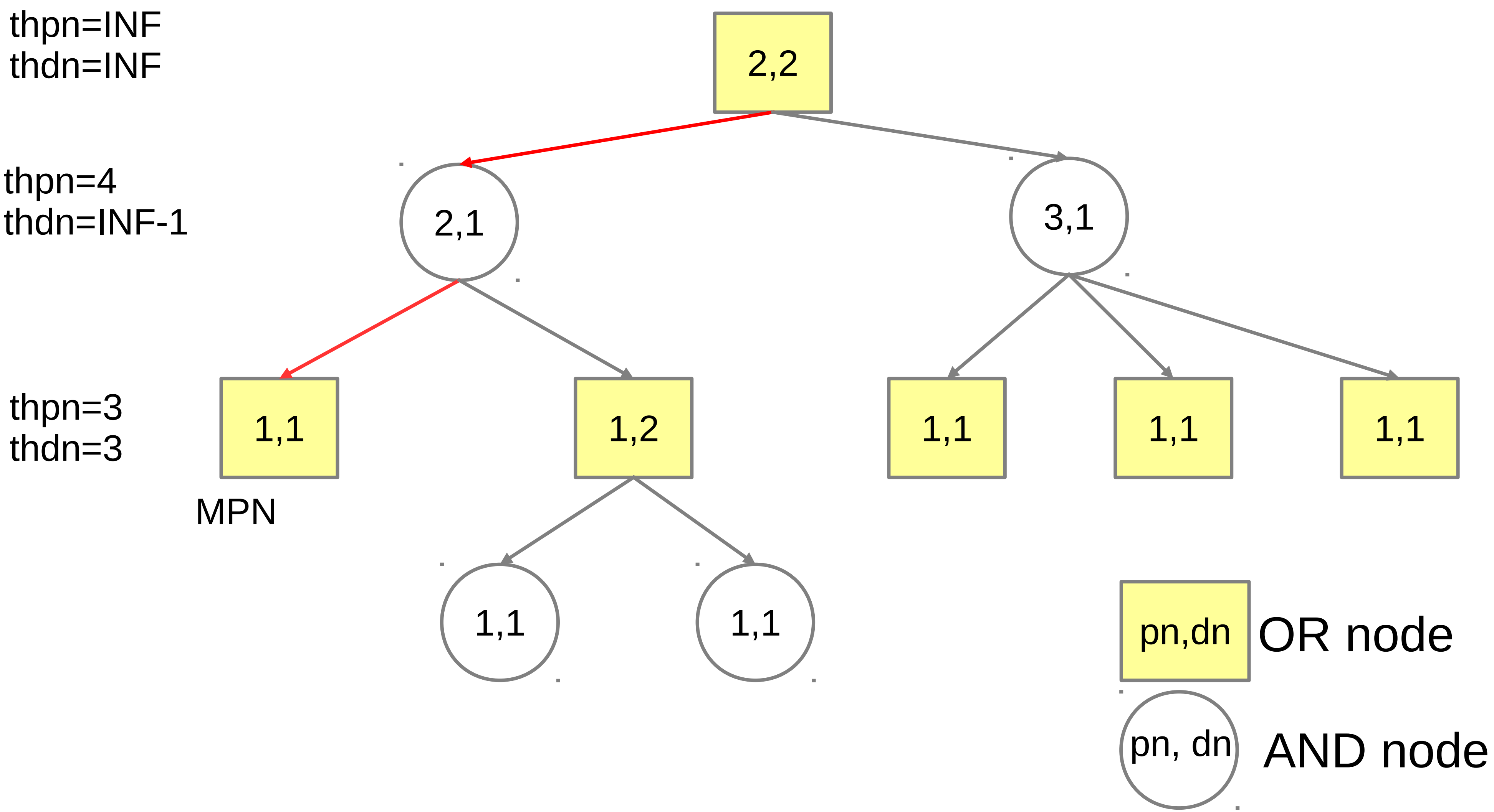
- PNS search often stays in one subtree for a long time
- As long as we can determine MPN, we don't care about proof and disproof numbers – can delay updates
- Example: pn(n) = min(100,90,20,60,50)=20 at OR node n
- Locally stay in subtree with pn=20 until its proof number *exceeds* smallest proof number among other children pn2 (50 in example)
- Globally, must also check if move decision would change higher up in the tree. Can pass down a condition of such change from parent as a threshold parameter
- Formula for new threshold: min(pn(parent), pn2+1)

# Main Idea of Df-pn's Threshold Controlling Technique (2 / 2)

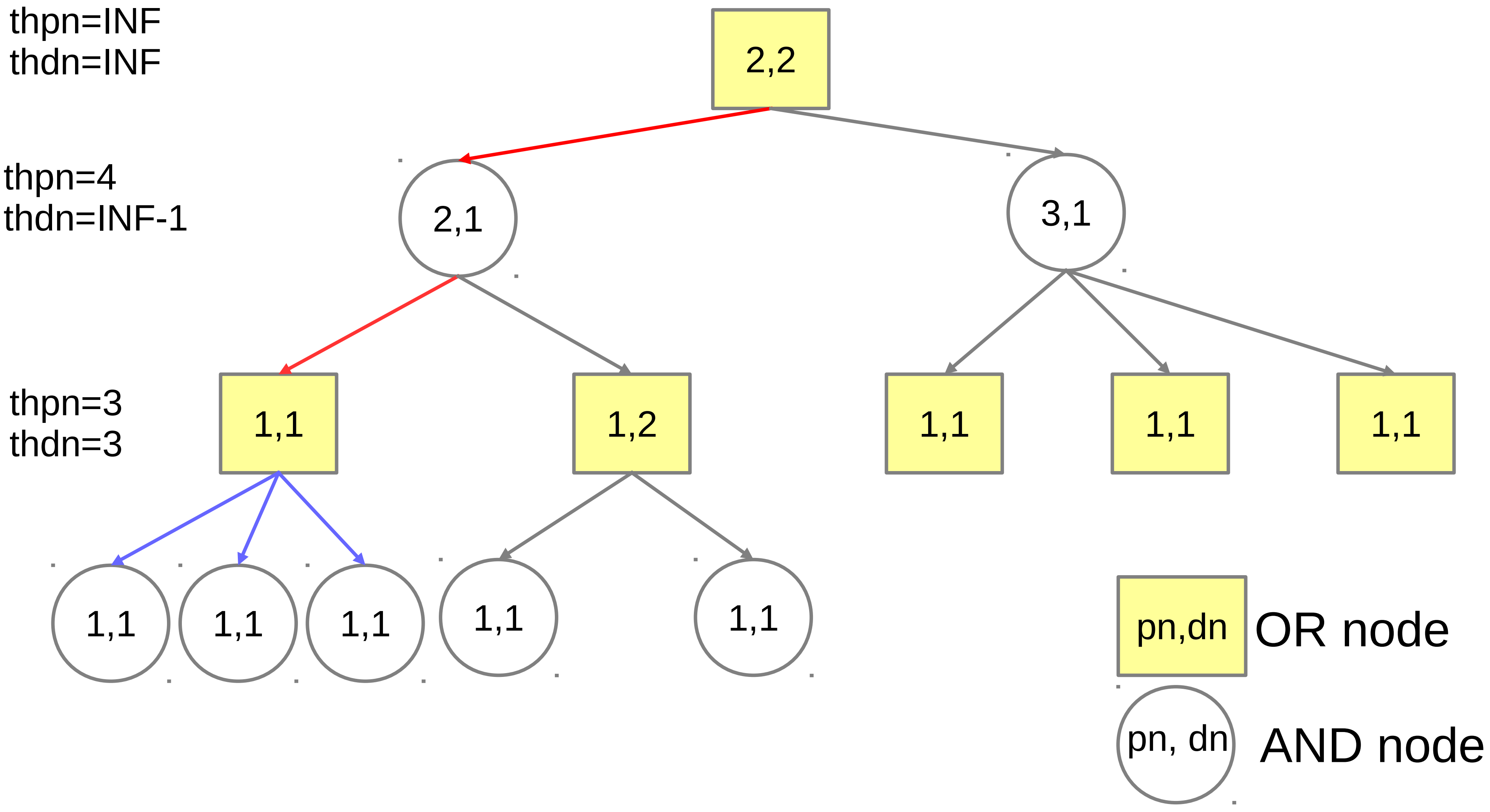
- pn(n) = pn(c1) + … + pn(ck) where c1,...,ck are n's children and n is an AND node
- Assume we have threshold for node n, n.thpn
- Say we are working on cj.  How long?
- Answer: until n.pn >= n.thpn, or increase cj exceeds difference n.thpn – n.pn. So set

  cj.thpn = pn(cj) + (n.thpn – n.pn)
- Apply same rules to set threshold for disproof number

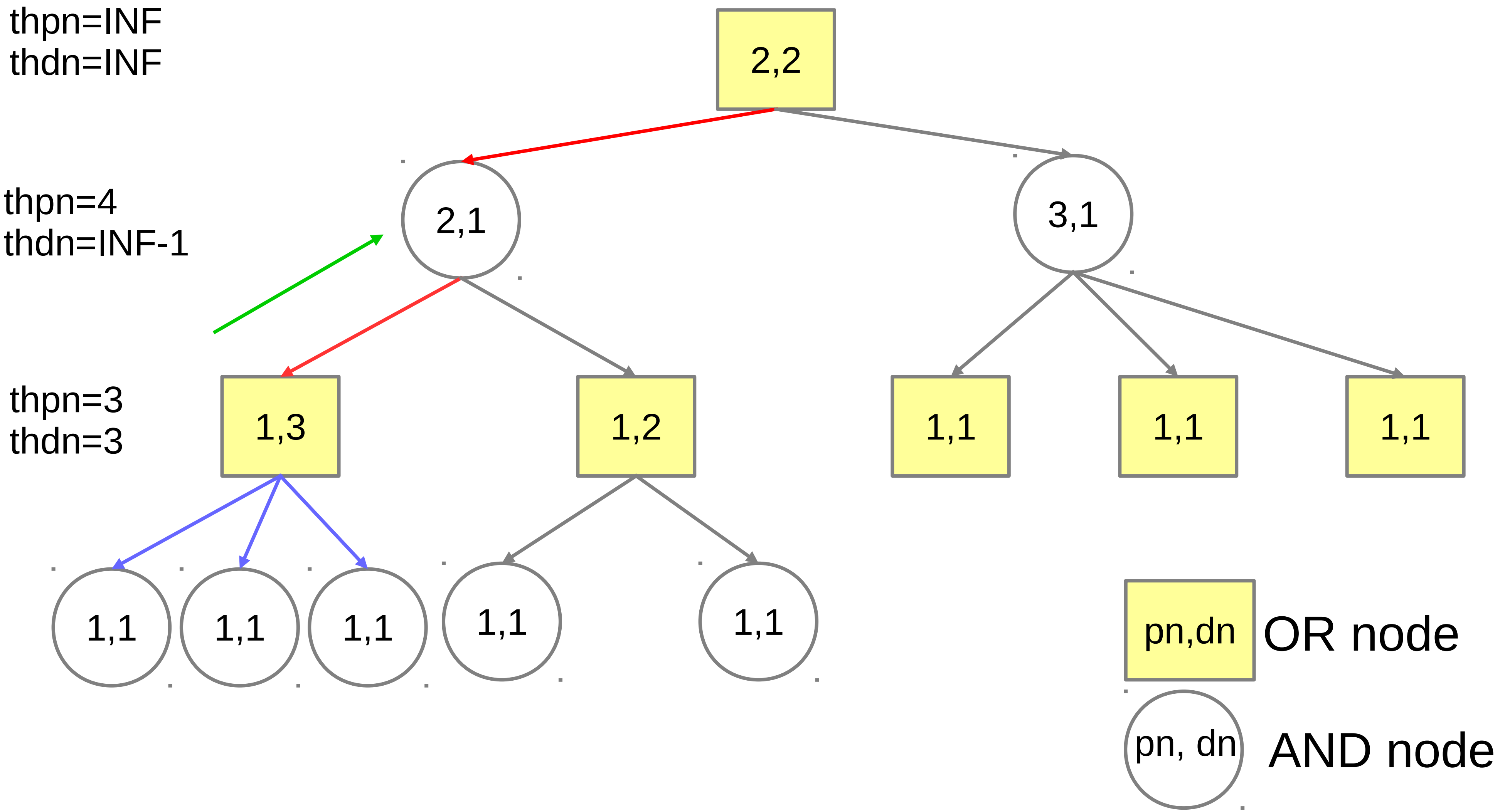
# Example of Df-pn (1 / 4)



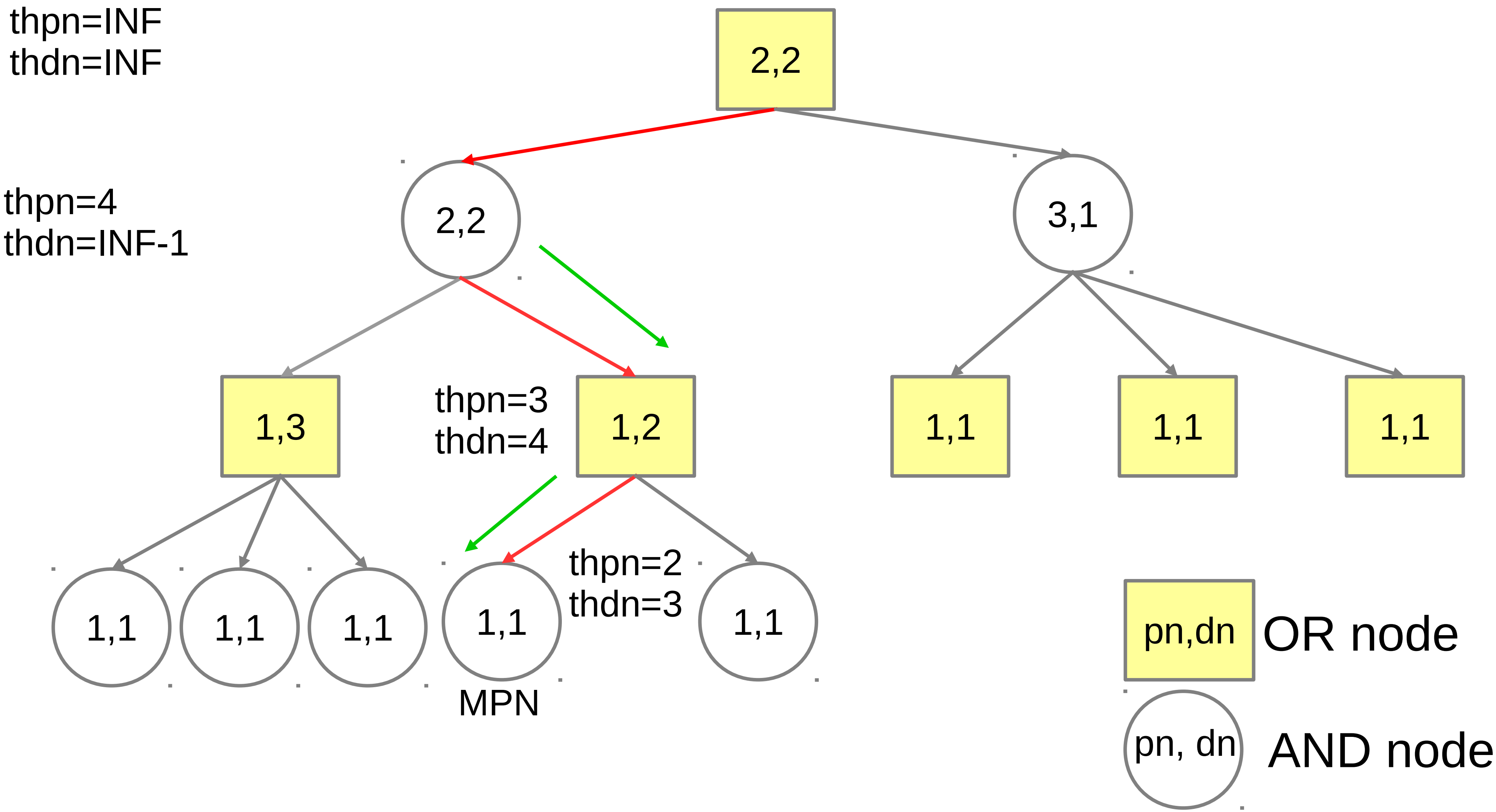
# Example of PNS (2 / 4)



# Example of Df-pn (3 / 4)



# Example of Df-pn (4 / 4)



# Outline of Df-pn Algorithm

a) Set root.thpn = root.thdn=INF and set n=root

b) Recompute pn(n) and dn(n) by using n's children

c) If n.thpn<= pn(n) or n.thdn <= dn(n) return to n's parent

d) If n is an OR node, select and examine child cj with the smallest proof number and set the thresholds to:

   - cj.thpn=min(n.thpn,pn2+1), cj.thdn = dn(cj) + (n.thdn – n.dn)

e) If n is an AND node, select and examine cj with the smallest disproof number and set the thresholds to:

   - cj.thpn = pn(cj) + (n.thpn – n.pn), cj.thdn=min(n.thpn,dn2+1)

f) Repeat until root is solved

pn2, dn2: smallest (dis-)proof numbers of other children than cj

# PNS Variants in Practice

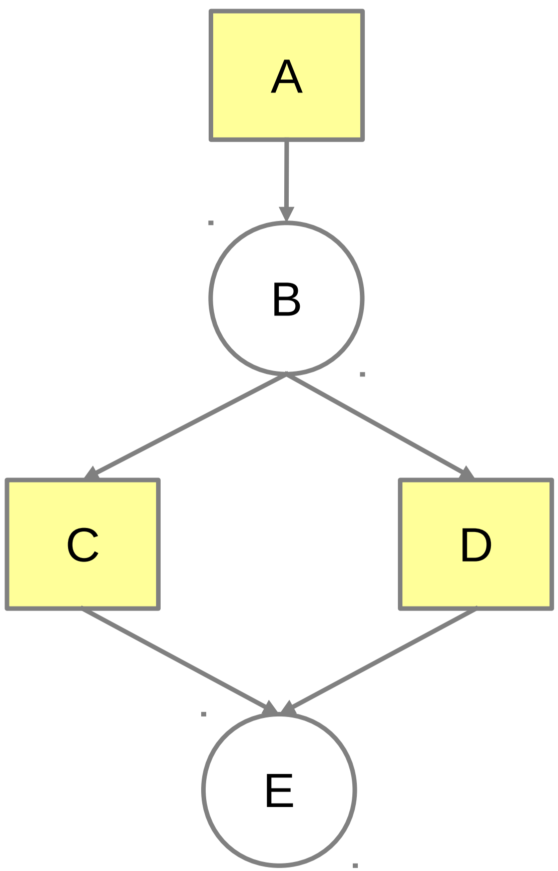
- Need to incorporate many techniques to make PNS work efficiently in practice
  - Problems in Directed Acyclic Graph (DAG) and Directed Cyclic Graph (DCG)
  - Search enhancements
  - Parallelization

# PNS on a DAG – Overcounting Proof and Disproof Numbers

- 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
- It happens in practice, e.g. tsume-shogi, Go
- NP-hard to compute accurate proof and disproof numbers
- Approximative approaches: Proof-Set Search, WPNS, and SNDA

# Example of Overcounting

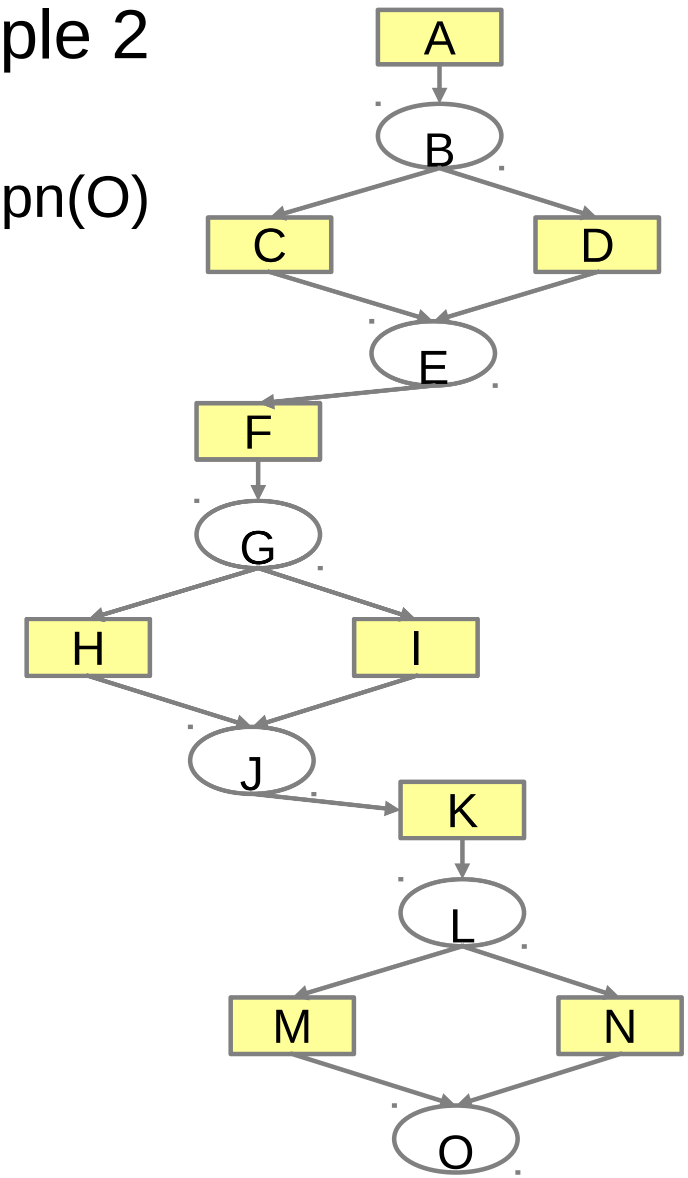
Example 1




pn(A)=pn(B)=pn(C)+pn(D)
= pn(E) + pn(E) = 2pn(E)

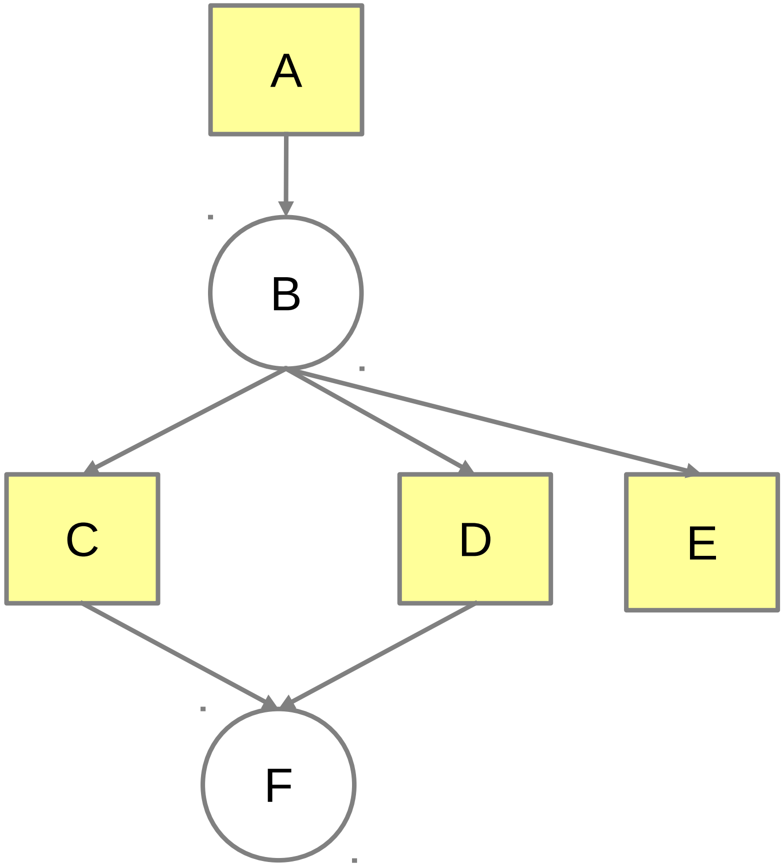
Example 2

pn(A)= 8pn(O)



# Proof-Set Search [Mueller, 2003]

- Use proof *sets* instead of proof “numbers”
- (Dis-)proof set of n = a set of leaf nodes that must be expanded to (dis-)prove
- Open question: How to implement time- and memory-efficient proof-set operations?

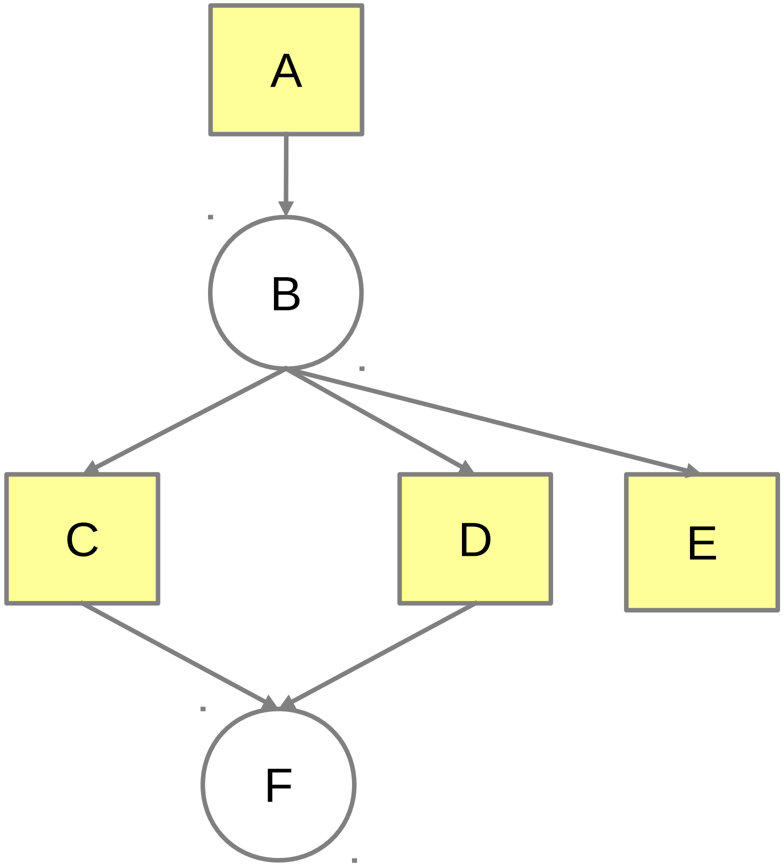



Example

$$\mathrm{pset}(B)=\mathrm{pset}(C)\cup\mathrm{pset}(D)\cup\mathrm{pset}(E)=\mathrm{pset}(F)\cup\mathrm{pset}(F)\cup\mathrm{pset}(E)$$
$$=\{F\}\cup\{F\}\cup\{E\}=\{E,F\}$$

# Weak Proof-Number Search (WPNS) [Ueda et al, 2008]

- Extension of [Okabe, 2005]
- Use standard formula for proof numbers at OR nodes and disproof numbers at AND nodes
- For AND node n, pn(n) = max(pn(c1),..pn(ck)) + k -1 where c1,...,ck are n's children
- Analogous computation for dn at OR node

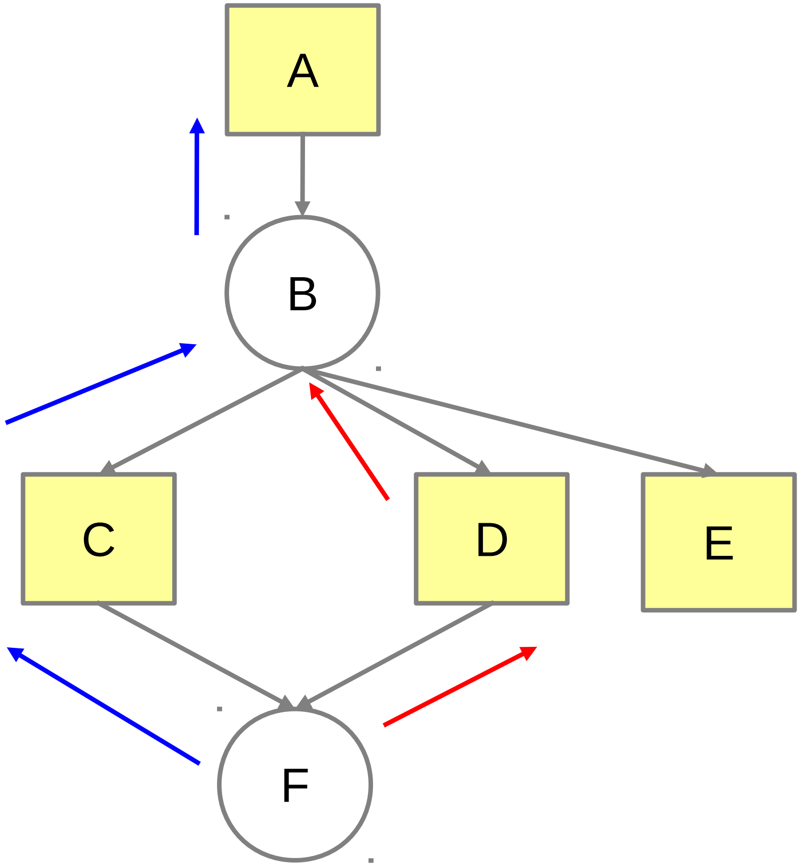



Example

pn(B)= max(pn(C), pn(D), pn(E)) + 2 = max(pn(F),pn(F),pn(E)) + 2
= max(pn(E),pn(F))+ 2

# Source Node Detection Algorithm (SNDA) [Kishimoto, 2010]

- Extension of [Nagai,2002]
- Keep a pointer to one parent p for each node
- Detect a source of DAG
- Take max instead of sum for nodes that may cause overcounting
- Take sum for others




Example

pn(B)= max(pn(C), pn(D)) + pn(E) = max(pn(F),pn(F)) + pn(E)
= pn(E)+pn(F)

# Comments on Solutions to Overcounting Problem

- There is always a trade-off between speed of search, accuracy of proof and disproof numbers and available memory
- Accurate proof and disproof numbers lead to reduction of node expansion but lead to reduction of node expansion rate too
    - E.g., WPNS tends to expand more nodes than SNDA but achieves comparable performance except for some very difficult problem instances in tsume-shogi [Kishimoto, 2010]

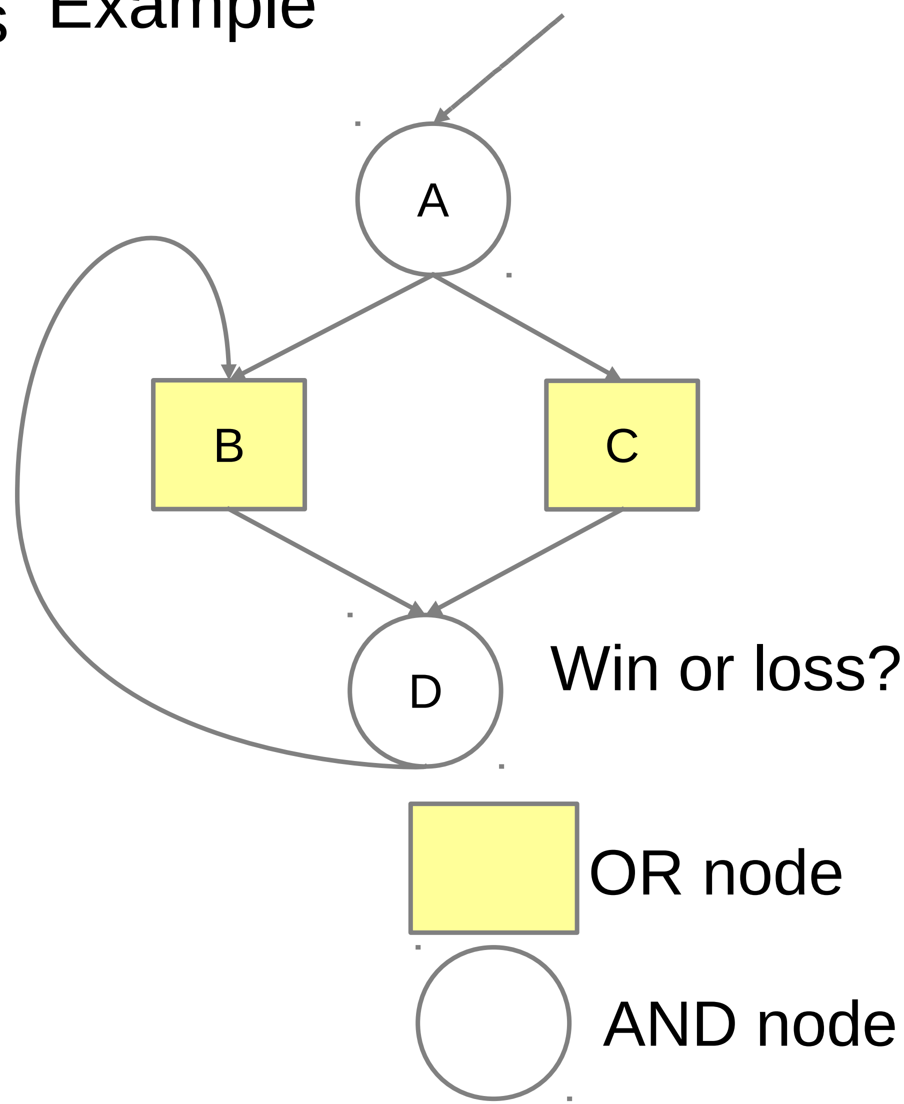
# PNS Variants on a DCG

- Need to address more issues
  - Graph History Interaction Problem
  - Infinite loop

# Graph-History Interaction (GHI) Problem [Palay,1983]

- Many games contain repetitions
- Outcome of repetition is determined by the rule of game
  - E.g., move leading to previous position is illegal in Go
- Transposition table ignores history
  - Never wants to give up using TT for performance reason
  - May contain incorrect results

Example

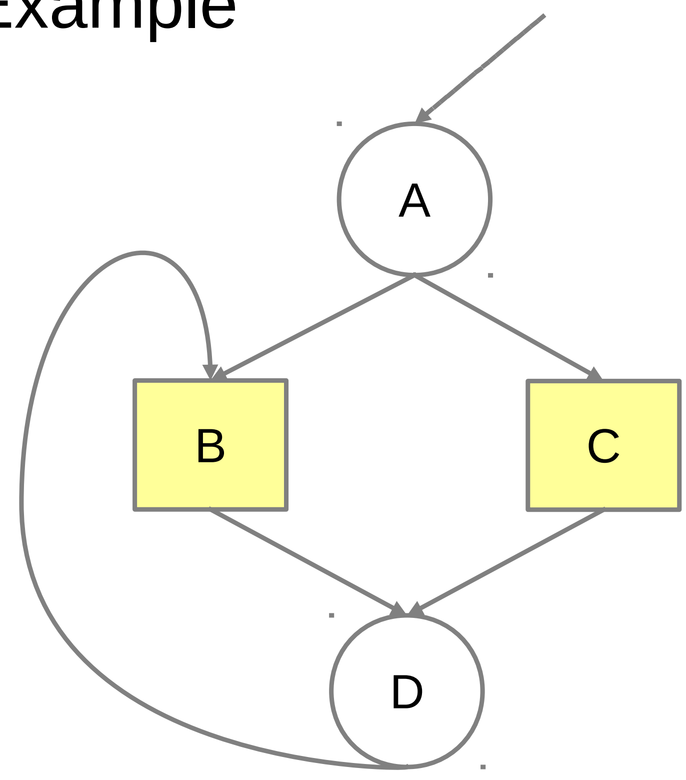


# General Solution to GHI [Kishimoto & Mueller, 2004]

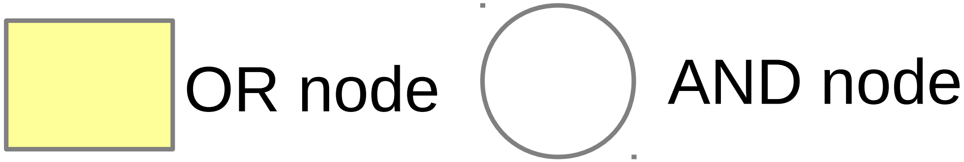
- Prepare encoded position and encoded path to transposition table entry
- Reuse proof and disproof numbers for unproven node
- Save win/loss via path if repetitions are involved
- Save win/loss with no condition if repetitions are not involved

Note: Works correctly with TT replacement schemes but need to reconstruct proof tree (See [Kishimoto, 2005])

Example




1. D via A->B->D Win
2. D via A->C->D Loss



# Infinite Loop Problem in Df-pn [Kishimoto & Mueller 2003, 2008]

- No new leaf is expanded
  - MPN property no longer holds
- Df-pn overcounts (dis-)proof numbers due to repetitions

Example (right-hand side)

dn(O)=dn(I) + dn(P) >= thdn(O)

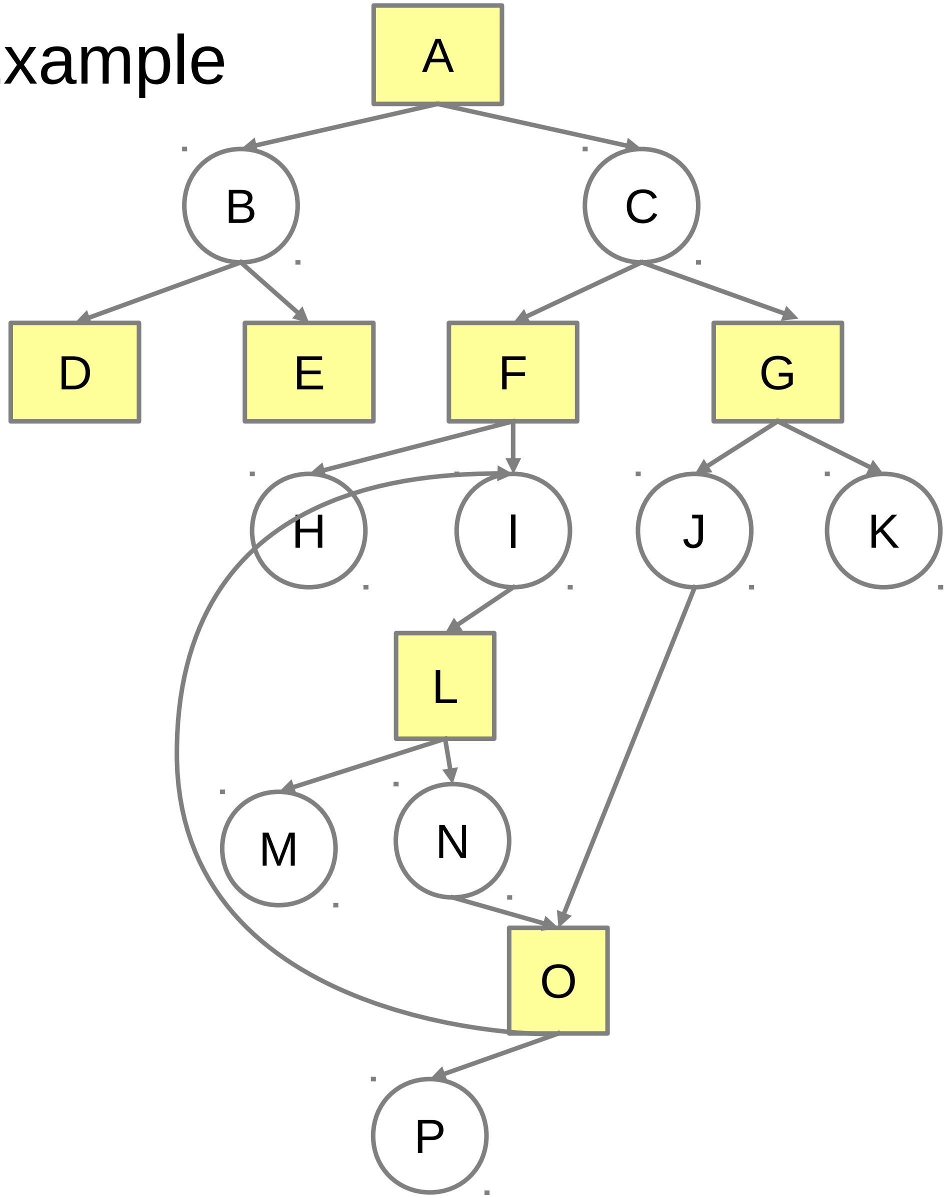
via A->C->G->J->O

dn(N)=dn(O) >= thdn(N)

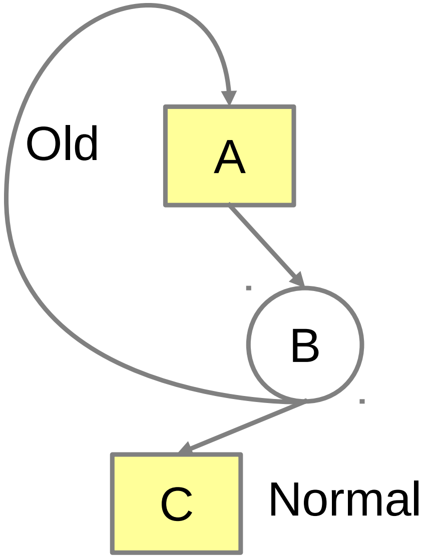
via A->C->F->I->L->N

Cycle A->C->G->J->O->I->L->N->O is never detected

Example



# Df-pn(r) (1 / 2)




- Keep the *minimal distance* from root
    - Normal child: has a larger minimal distance than parent
    - Old child: not normal child
- Modify computation of (dis-)proof number
    - AND node:

$$pn(n)= \begin{cases} \text{1. Ignore proof number of old nodes} \\ \text{(if unproven normal child exists)} \\ \text{2. Largest proof number of old node} \\ \text{(if all normal children are proven)} \end{cases}$$

- Analogous formula for dn(n) for OR node n
- Propagation of minimal distance to parent (see original paper)

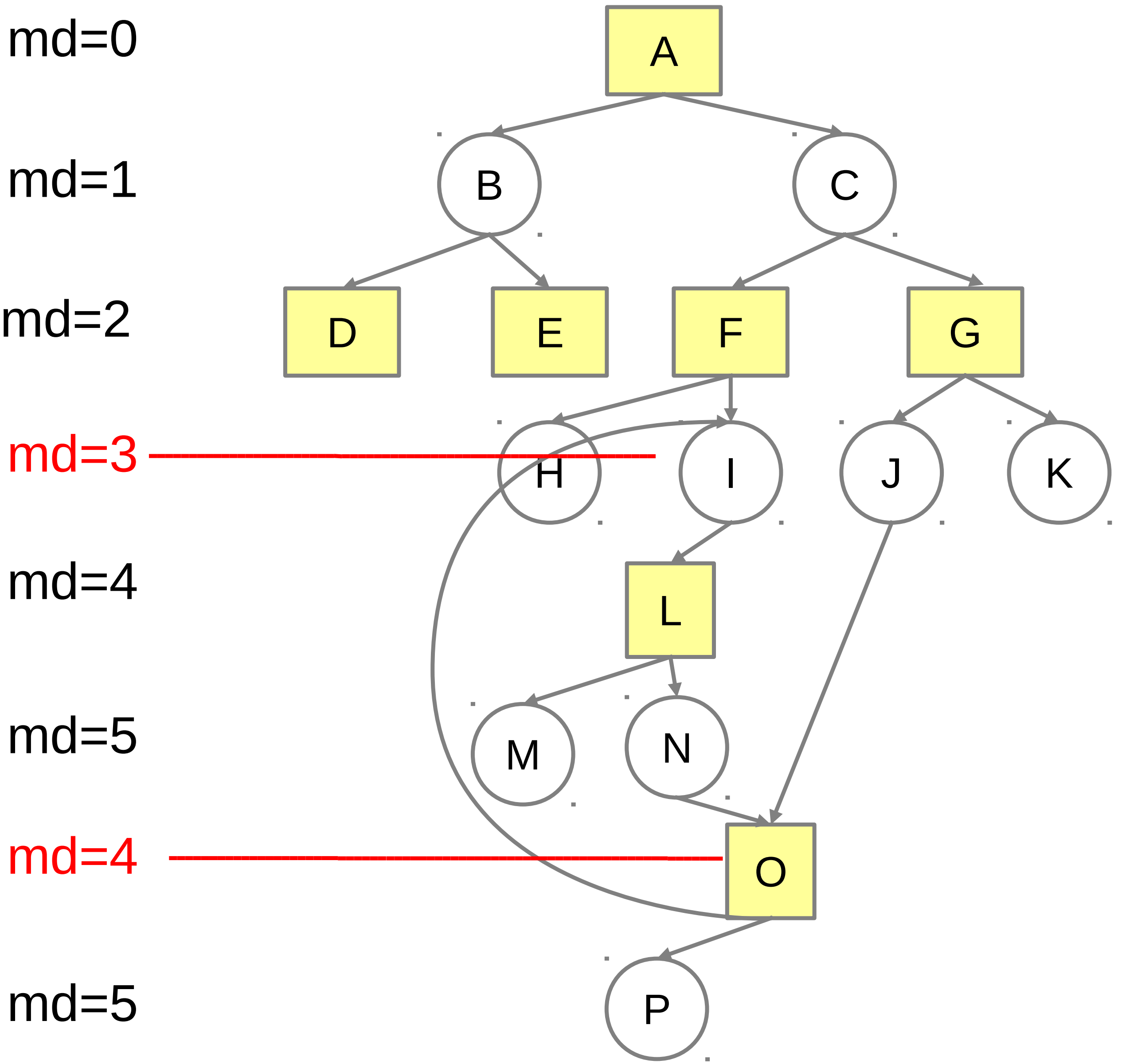
# Df-pn(r) (2 / 2)

- dn(O)=dn(P)

  if P is unproven

- dn(O)=dn(I)

  if P is disproven



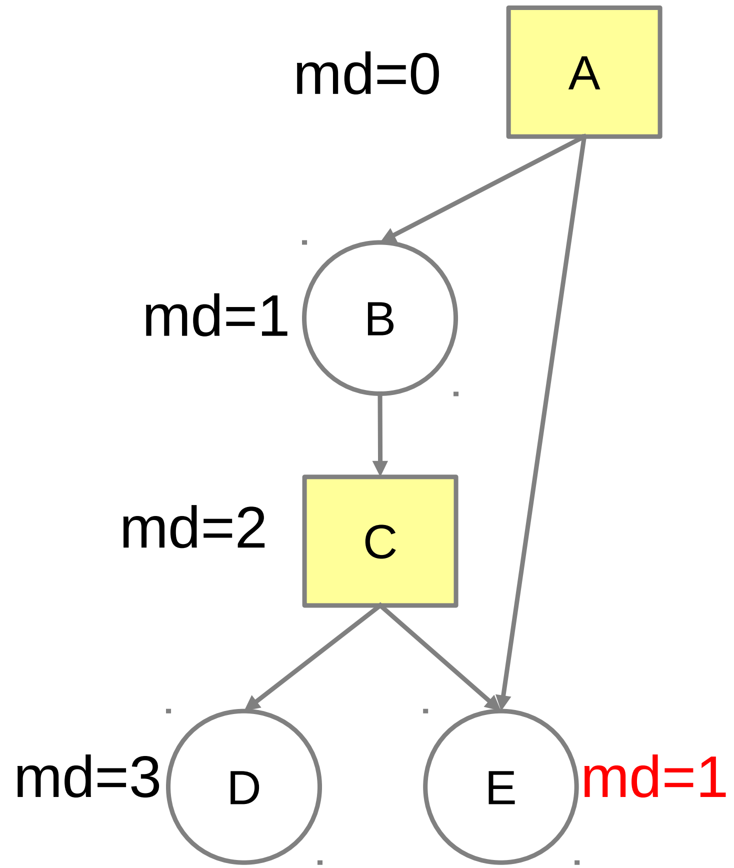
# Underestimation Problem of Df-pn(r) [Kishimoto, 2010]

- Df-pn(r) undercounts (dis-)proof number

Example (right-hand side)

- dn(C) must be dn(D)+dn(E)
- Df-pn(r) computes

  dn(C)=dn(D) (if D is unproven)



# Threshold Controlling Algorithm (TCA) [Kishimoto, 2010]

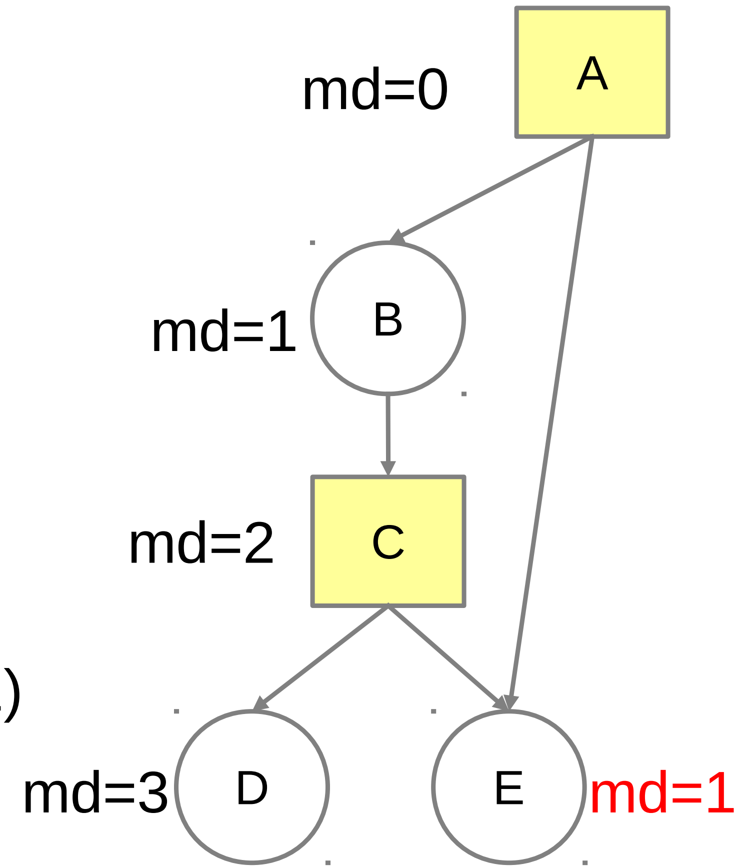
- Don't change the way of computing (dis-)proof number
- Increase threshold if n has old child

Example (right-hand side)

dn(C)=dn(D)+dn(E)

thdn(C)=max(thdn(C), dn(C)+1)

=max(thdn(C), dn(D) + dn(E) + 1)



# Search Enhancements for PNS Variants

- Heuristic initialization
- Modification to calculation of proof and disproof numbers
- Threshold control of df-pn
- Refining heuristic proofs
- Kawano's tree simulation
- Adding shallow depth-first search
- Early win/loss detection

# Heuristic Initialization

- Basics: pn, dn is lower bound on cost of solving node
- Initializing them with 1 is naïve
- Maybe we can find better estimates? e.g. depend on features of positions.
- Use domain-dependent evaluation functions evalpn(n), evaldn(n)
    - Manually tuned (e.g., df-pn+ [Nagai, 2002], [Kishimoto & Mueller, 2003], [Winands et al, 2011])
    - Machine learning such as support vector machine [Miwa et al,2005]
- Set pn(n)=evalpn(n) and dn(n) = evaldn(n) for leaf node n
- Large improvement in practice

# Modification to Calculation of Proof and Disproof Numbers

- Proof and disproof numbers often do not reflect the actual difficulty of solving a position
  - The number of legal moves is large and doesn't dramatically change between current and child nodes (e.g., Go and Hex)
  - Values of siblings are highly correlated, e.g. interposing piece drops in tsume-shogi, sacrificing pieces
- Many ways to modify pn & dn calculation schemes to reflect real difficulty of positions
  - Consider only a smaller number of best children [Yoshizoe, 2008][Arneson et al, 2011]
  - Detect “threats” [Nagai, 2002][Soeda et al, 2006][Yoshizoe et al, 2007]
  - Define domain-dependent rule (e.g., [Seo, 1995])

# Threshold Control of Df-pn

- Df-pn + heuristic initialization (called df-pn+ [Nagai, 2002]) increases overhead of re-expanding interior nodes
- Df-pn suffers from thrashing TT if more than one sibling exists and search space does not fit into TT
- Df-pn (or df-pn+) increments thresholds by the minimum possible amount
- Increase threshold increments over those of original df-pn
    - n.thpn = min(n.thpn(n), pn2 + δ) where δ > 1

      Constant δ [Nagai,2002], variable δ [Kishimoto & Mueller, 2005]
    - n.thpn = min(n.thpn(n), $\lceil pn2 \cdot (1+\epsilon) \rceil$) [Pawlewicz & Lew, 2007]

# Refining Heuristic Proofs [Schaeffer et al, 2005, 2007]

- Checkers solution by Schaeffer et al
- Evaluation of position is accurate – high-performance alpha-beta search with depth of 17-23
- Pseudo-proofs: assume everything with evaluation > 150 is win, < -150 is loss. Create proof tree.
- After 150 is proven, change bound to 200/-200. Then 250/-250, etc. Once bound reaches INF/-INF, proof is complete

# Other Search Enhancements

- Tree simulation [Kawano, 1996]
  - Try to construct (dis-)proof tree if “similar” positions are (dis-)proven
- Shallow depth-first iterative deepening search at leaf nodes
  - Pseudo one move look ahead, e.g. [Allis et al, 1994] [Breuker et al, 1998][Winands, 2004]
  - 3-ply search at non-terminal OR leaf in tsume-shogi [Kaneko et al, 2005]
- Early win/loss detection
  - E.g., retrograde analysis, domain-dependent, static analysis of positions, dominance relations

# Parallel PNS Variants (1 / 3)

- Achieving reasonable parallel performance is difficult as in parallel alpha-beta
    - Search, communication and synchronization overhead
    - Sharing TT information among processors in distributed memory environments
- Moreover, PNS variants construct more unbalanced trees than alpha-beta
- Unbalanced trees often make PNS variants explore different but still promising portions of search space

# PNS Variants (2 / 3)

- Shared-memory parallel df-pn [Kaneko, 2010]
    - Share transposition table among threads
    - Use virtual proof and disproof numbers, vpn(n), vdn(n) (c.f., Coulom's virtual loss in MCTS)
    - For OR node n, vpn(n) = pn(n) + k where pn(n) is proof number for n and k is the number of threads that enter n
    - Define analogously vdn(n) for AND node n
    - Select child with best child ci with smallest vpn(ci) at OR node n (and analogously at AND node n)
    - Similar idea (but slightly different calculation scheme) is used to solve Hex [Pawlewicz et al, 2013]

# Parallel PNS Variants (3 / 3)

- Master-slave framework in distributed memory environments, e.g., [Schaeffer et al, 2007][Wu et al, 2011][Saffidine et al, 2012]
    - One master manages a subtree of the root node and coordinates work to slaves
    - Master preserves the most important search results
    - Slaves independently examine assigned work until condition determined by master is satisfied
    - Several strategies to initiate parallelism are proposed

    E.g., virtual loss, semi-automatic selection of candidates

# Extension to Multi-Valued Cases

- Series of Boolean searches, e.g., binary search for sequential search
- Each search uses a bound on leaf value as in null-window search
- How to reuse search results from previous searches
    - Previous (dis-)proof was with harder bound than current one – can just take old result [Moldenhauer, 2009]
    - Unproven (dis-)proof numbers from previous search, e.g., proving “win by Ko”/seki in Go [Kishimoto & Mueller, 2003] [Niu et al, 2006]
- Multi-Outcome Proof-Number Search determines multi-valued case with one search [Saffidine & Cazenave, 2012]

# Comments on PNS Variants

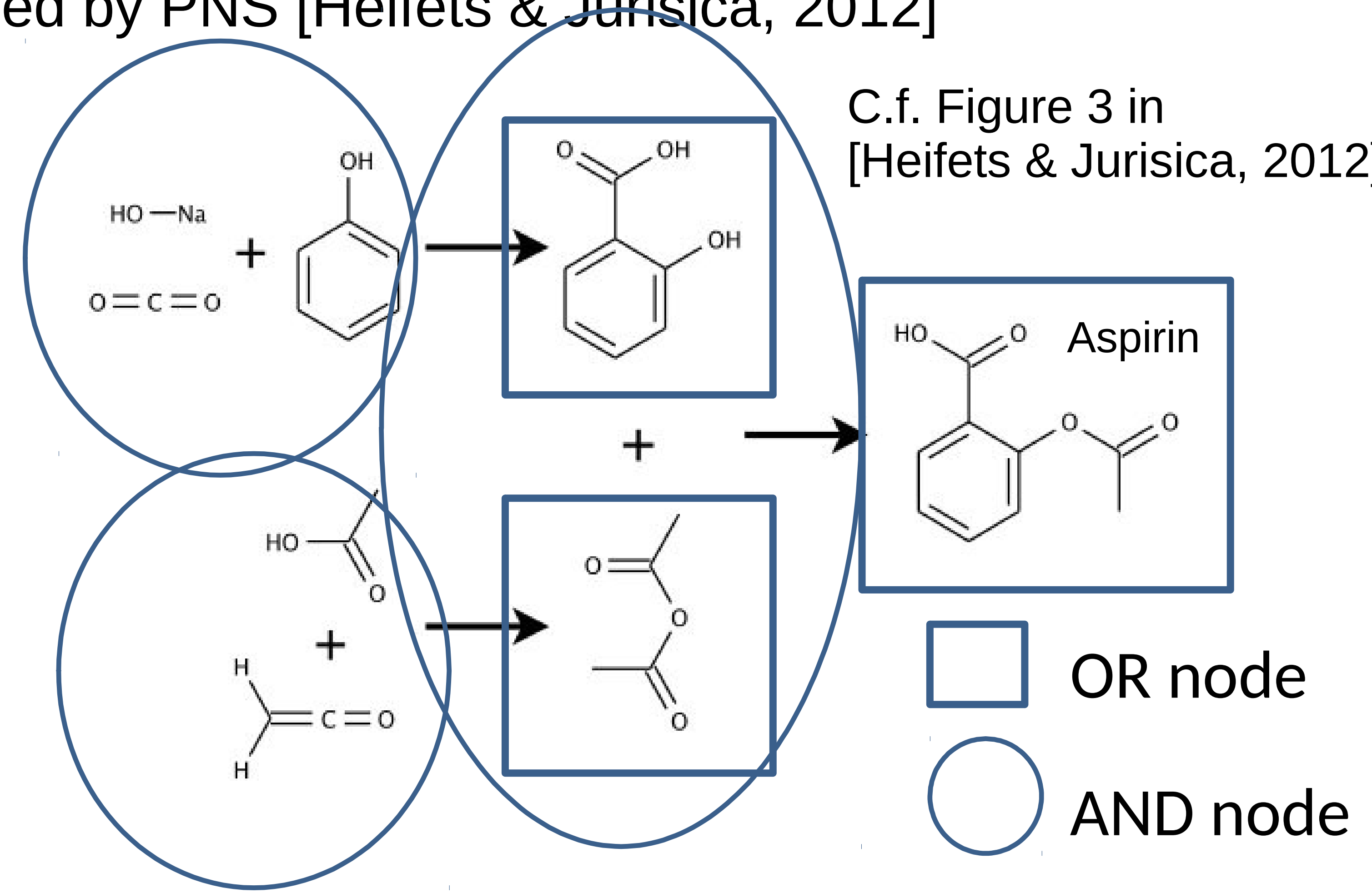
- Good cases
  - Uneven branching factor
  - Early wins/losses found in some branches
  - Number of moves correlated with winning chance
- Bad cases
  - “Everything looks the same”
  - Uniform branching factor, no early wins/losses
- Really bad cases
  - Proof numbers actively misleading
  - Lots of “forcing moves”, but they don't work. Only a “quiet” move works

# Applications (1 / 3)

- PNS variants are used to solve games/game positions
  - Checkers [Schaeffer et al, 2007], tsume-shogi (e.g. [Seo et al, 2001][Nagai, 2002]), tsume-Go [Kishimoto & Mueller, 2005] etc
- PNS variants are recently applied to other kinds of game
  - Multi-player games [Saito & Winands, 2010], two-player game with imperfect information [Sakuta, 2001], moving target search [Moldenhauer, 2009]
- Proof numbers were applied to theorem proving in 80s [Elkan, 1989]

# Applications (2 / 3)

- The problem of chemical synthesis from given simpler molecules was formulated as AND/OR graph search and solved by PNS [Heifets & Jurisica, 2012]




C.f. Figure 3 in [Heifets & Jurisica, 2012]

# Application (3 / 3)

- Optimally solving Maximum a Posteriori (MAP) task defined over graphical model can be modeled as AND/OR graph search [Dechter & Mateescu, 2007]
  - OR node: Assign value to variable
  - AND node: Select one variable
- RBFOO, which has commonalities with PNS but includes several modifications, is empirically shown to be efficient (See [Kishimoto & Marinescu, 2014] for details)

# Conclusions

- Gave an overview about PNS variants that are commonly used to solve games/game positions
  - Basic ideas of PNS and df-pn
  - Issues to be resolved, e.g. memory, DAG, DCG
  - Search enhancements
  - Parallelism
  - Multi-valued scenario
  - Applications