Scout, NegaScout and Proof-Number Search

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

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

Introduction

- It looks like alpha-beta pruning is the best we can do for a **generic searching procedure**.
  - What else can be done generically?
  - Alpha-beta pruning follows basically the "intelligent" searching behaviors used by human when domain knowledge is not involved.
  - Can we find some other "intelligent" behaviors used by human during searching?
- Intuition: MAX node.
  - Suppose we know currently we have a way to gain at least 300 points at the first branch.
  - If there is an efficient way to know the second branch is at most gaining 300 points, then there is no need to search the second branch in detail.
    - ▷ *Is there a way to search a tree approximately?*
    - ▷ *Is searching approximately faster than searching exactly?*
- Similar intuition holds for a MIN node.

SCOUT procedure

- Invented by Judea Pearl in 1980.
- It may be possible to verify whether the value of a branch is greater than a value v or not in a way that is faster than knowing its exact value.
- High level idea:
  - While searching a branch T_b of a MAX node, if we have already obtained a lower bound v_ℓ.
    - ▷ *First TEST whether it is possible for T_b to return something greater than v_ℓ.*
    - ▷ *If FALSE, then there is no need to search T_b.*
      *This is called **fails the test.***
    - ▷ *If TRUE, then search T_b.*
      *This is called **passes the test.***
  - While searching a branch T_c of a MIN node, if we have already obtained an upper bound v_u
    - ▷ *First TEST whether it is possible for T_c to return something smaller than v_u.*
    - ▷ *If FALSE, then there is no need to search T_c.*
      *This is called **fails the test.***
    - ▷ *If TRUE, then search T_c.*
      *This is called **passes the test.***

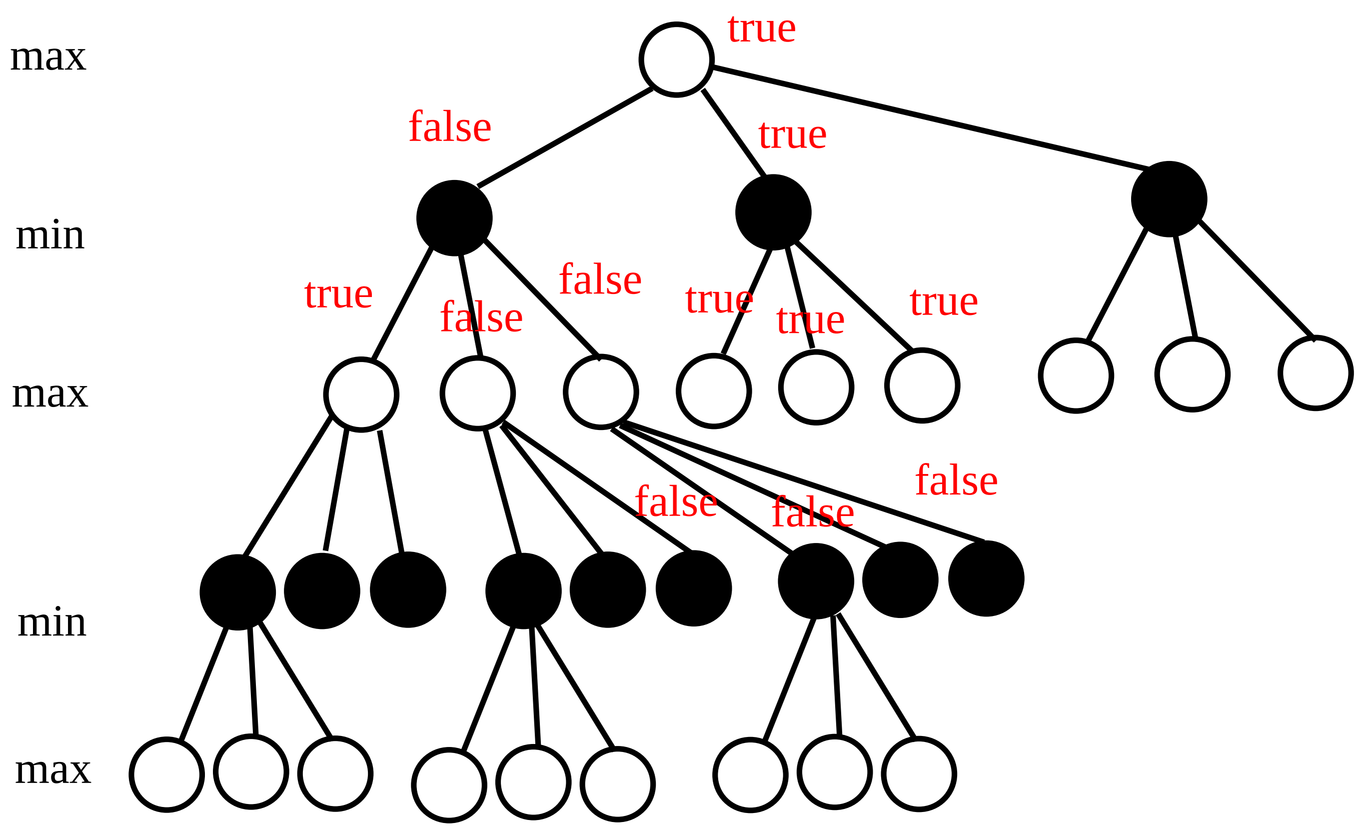
How to TEST $> v$

procedure TEST(position p, value v, condition $>$)

// test whether the value of the branch at p is $> v$

- **determine the successor positions $p_1, \ldots, p_d$ of p**
- **if $d = 0$, then // terminal**
  - ▷ ***if $f(p) > v$ then // f(): evaluating function***
  - ▷ ***return TRUE***
  - ▷ ***else return FALSE***
- **if p is a MAX node, then**
  - **for $i := 1$ to d do**
    - ▷ ***if TEST(p_i, v, $>$) is TRUE, then***
      return TRUE // succeed if a branch is $> v$
  - **return FALSE // fail only if all branches $\leq v$**
- **if p is a MIN node, then**
  - **for $i := 1$ to d do**
    - ▷ ***if TEST(p_i, v, $>$) is FALSE, then***
      return FALSE // fail if a branch is $\leq v$
  - **return TRUE // succeed only if all branches are $> v$**

Illustration of TEST



How to TEST — Discussions

- **Condition can be stated as $<$ by properly revising the algorithm.**
  - $\boxed{\textbf{TEST}(p,v,>) \textbf{ is TRUE}} \equiv \boxed{\textbf{TEST}(p,v,<=) \textbf{ is FALSE}}$
  - $\boxed{\textbf{TEST}(p,v,>) \textbf{ is FALSE}} \equiv \boxed{\textbf{TEST}(p,v,<=) \textbf{ is TRUE}}$
  - $\boxed{\textbf{TEST}(p,v,<) \textbf{ is TRUE}} \equiv \boxed{\textbf{TEST}(p,v,>=) \textbf{ is FALSE}}$
  - $\boxed{\textbf{TEST}(p,v,<) \textbf{ is FALSE}} \equiv \boxed{\textbf{TEST}(p,v,>=) \textbf{ is TRUE}}$
- **Practical consideration:**
  - **Set a depth limit and evaluate the position's value when the limit is reached.**

How to TEST $< v$

procedure TEST(position p, value v, condition $<$)

// test whether the value of the branch at p is $< v$

- **determine the successor positions $p_1, \ldots, p_d$ of p**
- **if $d = 0$, then // terminal**
  - ▷ ***if $f(p) < v$ then // f(): evaluating function***
  - ▷ ***return TRUE***
  - ▷ ***else return FALSE***
- **if p is a MAX node, then**
  - **for $i := 1$ to d do**
    - ▷ ***if TEST(p_i, v, $<$) is FALSE, then***
      return FALSE // succeed if a branch is $\geq v$
  - **return TRUE // succeed only if all branches $< v$**
- **if p is a MIN node, then**
  - **for $i := 1$ to d do**
    - ▷ ***if TEST(p_i, v, $<$) is TRUE, then***
      return TRUE // succeed if a branch is $< v$
  - **return FALSE // fail only if all branches are $\geq v$**

Main SCOUT procedure (1/2)

Algorithm SCOUT(position p)

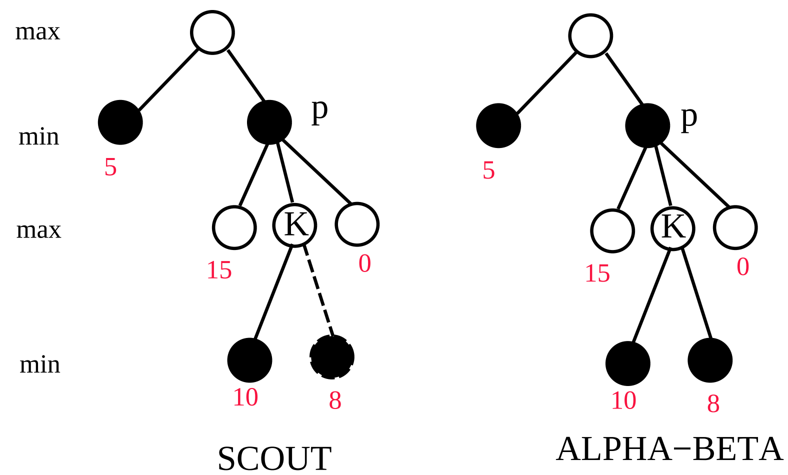
- **determine the successor positions** $p_1, \ldots, p_d$
- **if** $d = 0$, **then return** $f(p)$
- **else** $v = SCOUT(p_1)$ **// SCOUT the first branch**
- **if** p **is a MAX node**
  - **for** $i := 2$ **to** d **do**
    - ▷ ***if TEST(p_i, v, $>$) is TRUE, // TEST first for the rest of the branches***
      then $v = SCOUT(p_i)$ ***// find the value of this branch if it can be*** $> v$
- **if** p **is a MIN node**
  - **for** $i := 2$ **to** d **do**
    - ▷ ***if TEST(p_i, v, $<$) is TRUE, // TEST first for the rest of the branches***
      then $v = SCOUT(p_i)$ ***// find the value of this branch if it can be*** $< v$
- **return** v

Main SCOUT procedure (2/2)

- **Note that v is the current best value at any moment.**
- **MAX node:**
  - **For any $i > 1$, if TEST(p_i, v, $>$) is TRUE,**
    - ▷ ***then the value returned by $SCOUT(p_i)$ must be greater than v.***
  - **We say the p_i passes the test if TEST(p_i, v, $>$) is TRUE.**
- **MIN node:**
  - **For any $i > 1$, if TEST(p_i, v, $<$) is TRUE,**
    - ▷ ***then the value returned by $SCOUT(p_i)$ must be smaller than v.***
  - **We say the p_i passes the test if TEST(p_i, v, $<$) is TRUE.**

Discussions for SCOUT (1/2)

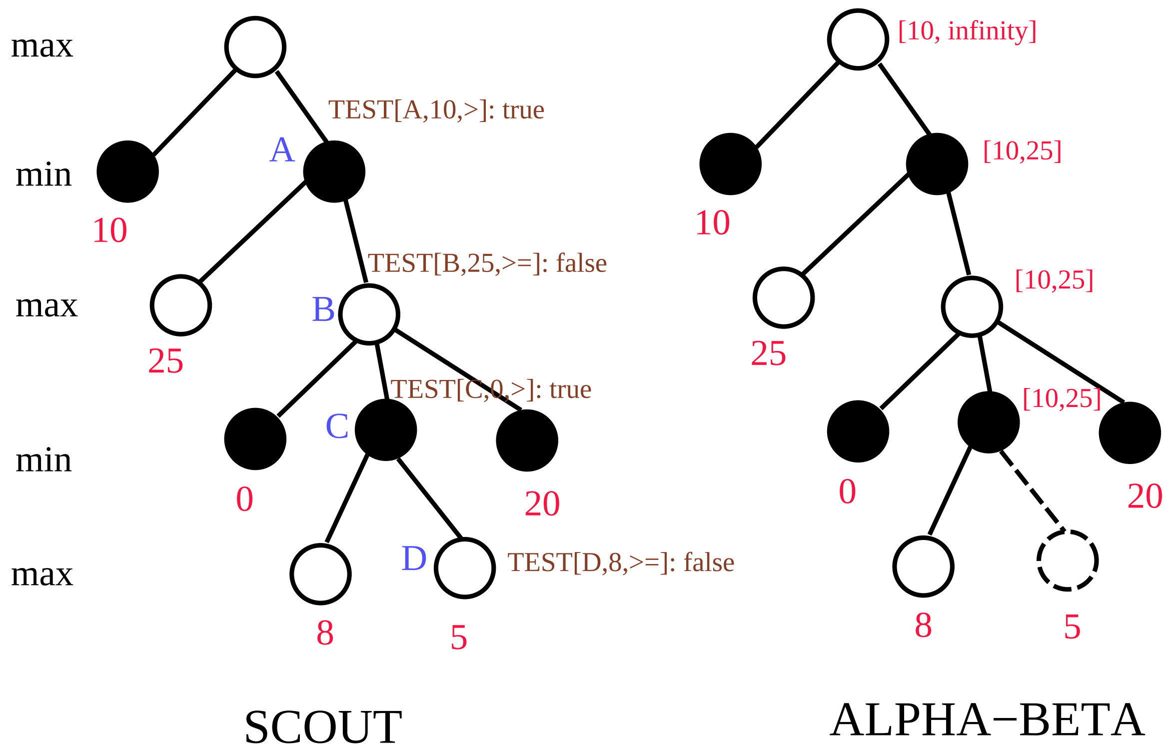
- **TEST who is called by SCOUT may visit less nodes than alpha-beta.**



  - **Assume $TEST(p, 5, >)$ is called by the root after the first branch is evaluated.**
    - ▷ ***It calls $TEST(K, 5, >)$ which skips K's second branch.***
    - ▷ ***$TEST(p, 5, >)$ is FALSE, i.e., fails the test, after returning from the 3rd branch.***
    - ▷ ***No need to do SCOUT for the branch p.***
  - **Alpha-beta needs to visit K's second branch.**

Discussions for SCOUT (2/2)

- **SCOUT may pay many visits to a node that is cut off by alpha-beta.**



Number of nodes visited (1/3)

- **For TEST to return TRUE for a subtree T, it needs to evaluate at least**
  - *one child for a MAX node in T, and*
  - *and all of the children for a MIN node in T.*
  - *If T has a fixed branching factor b and uniform depth d, the number of nodes evaluated is $\Omega(b^{d/2})$.*
- **For TEST to return FALSE for a subtree T, it needs to evaluate at least**
  - *one child for a MIN node in T, and*
  - *and all of the children for a MAX node in T.*
  - *If T has a fixed branching factor b and uniform depth d, the number of nodes evaluated is $\Omega(b^{d/2})$.*

Number of nodes visited (2/3)

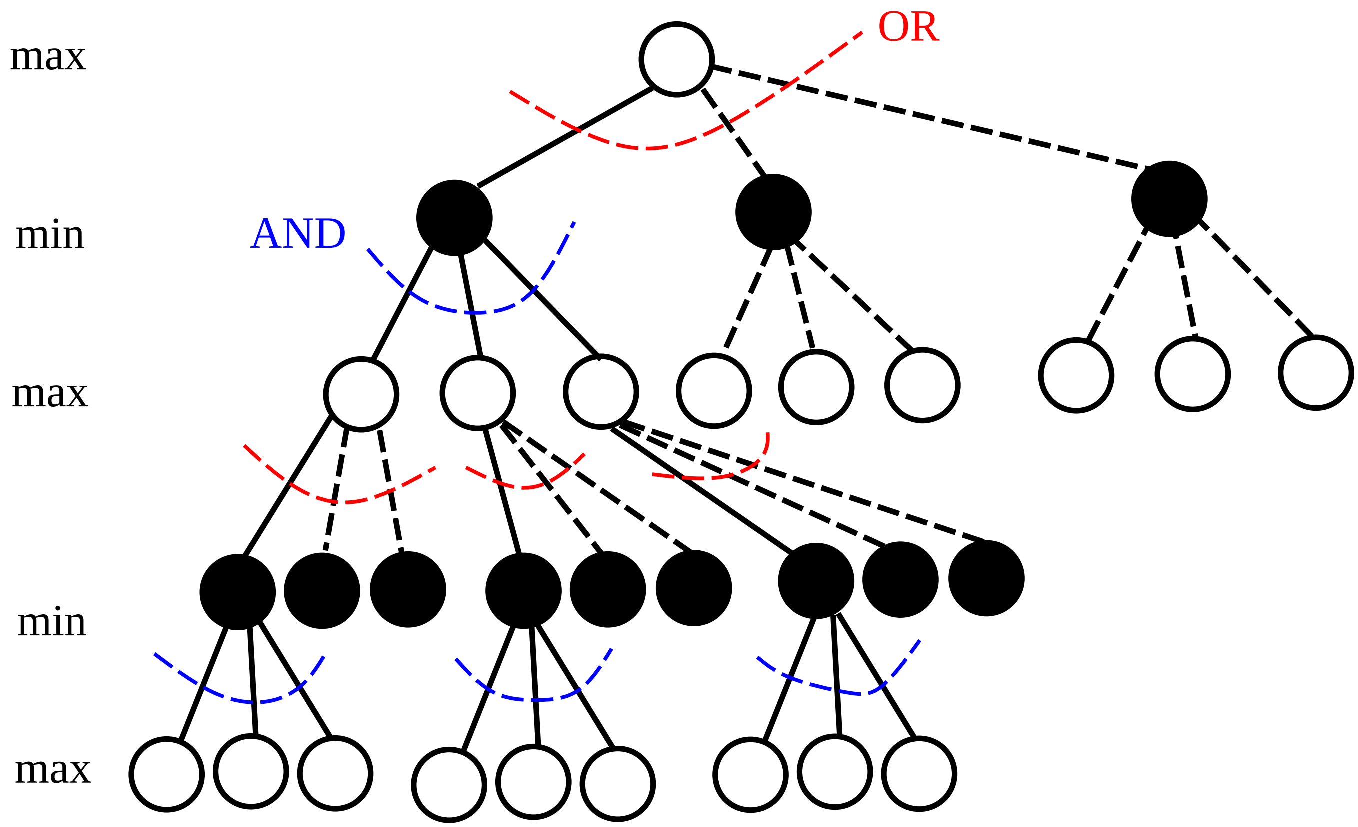
- **Assumptions:**
  - **Assume a full complete d-ary tree with depth ℓ where ℓ is even.**
  - **The depth of the root, which is a MAX node, is 0.**
- **The total number of nodes in the tree is $\frac{d^{\ell+1}-1}{d-1}$.**
- **The minimum number of nodes visited by TEST when it returns TRUE.**

$$
\begin{aligned}
&= 1+1+d+d+d^2+d^2+d^3+d^3+\cdots+d^{\ell/2-1}+d^{\ell/2-1}+d^{\ell/2} \\
&= 2\cdot(d^0+d^1+\cdots+d^{\ell/2})-d^{\ell/2} \\
&= 2\cdot\frac{d^{\ell/2+1}-1}{d-1}-d^{\ell/2}
\end{aligned}
$$

- **The minimum number of nodes visited by alpha-beta.**

$$
\begin{aligned}
&= \sum_{i=0}^{\ell} d^{\lceil i/2\rceil}+d^{\lfloor i/2\rfloor}-1 \\
&= \sum_{i=0}^{\ell} d^{\lceil i/2\rceil}+\sum_{i=0}^{\ell} d^{\lfloor i/2\rfloor}-(\ell+1) \\
&= (1+d+d+\cdots+d^{\ell/2}+d^{\ell/2})+ \\
&\quad (1+1+d+d+\cdots+d^{\ell/2-1}+d^{\ell/2-1}+d^{\ell/2})-(\ell+1)
\end{aligned}
$$

Number of nodes visited (3/3)



Comparisons

- **When the first branch of a node has the best value, then TEST scans the tree fast.**
  - **The best value of the first $i-1$ branches is used to test whether the ith branch needs to be searched exactly.**
  - **If the value of the first $i-1$ branches of the root is better than the value of ith branch, then we do not have to evaluate exactly for the ith branch.**
- **Compared to alpha-beta pruning whose cut off comes from bounds of search windows.**
  - **It is possible to have some cut-off for alpha-beta as long as there are some relative move orderings are "good."**
    - ▷ *The moving orders of your children and the children of your ancestor who is odd level up decide a cut-off.*
  - **The search bound is updated during the searching.**
    - ▷ *Sometimes, a deep alpha-beta cut-off occurs because a bound found from your ancestor a distance away.*

Performance of SCOUT (1/2)

- **A node may be visited more than once.**
  - **First visit is to TEST.**
  - **Second visit is to SCOUT.**
    - ▷ *During SCOUT, it may be TESTed with a different value.*
  - **Q: Can information obtained in the first search be used in the second search?**
- **SCOUT is a recursive procedure.**
  - **A node in a branch that is not the first child of a node with a depth of ℓ.**
    - ▷ *Note that the depth of the root is defined to be 0.*
    - ▷ *Every ancestor of you may initiate a TEST to visit you.*
    - ▷ *It can be visited ℓ times by TEST.*

Performance of SCOUT (2/2)

- **Show great improvements on $depth > 3$ for games with small branching factors.**
  - **It traverses most of the nodes without evaluating them preciously.**
  - **Few subtrees remained to be revisited to compute their exact mini-max values.**
- **Experimental data on the game of Kalah show [UCLA Tech Rep UCLA-ENG-80-17, Noe 1980]:**
  - **SCOUT favors "skinny" game trees, that are game trees with high depth-to-width ratios.**
  - **On depth = 5, it saves over 40% of time.**
  - **Maybe bad for games with a large branching factor.**
  - **Move ordering is very important.**
    - ▷ ***The first branch, if is good, offers a great chance of pruning further branches.***

Alpha-beta revisited

- **In an alpha-beta search with a window [$alpha$,$beta$]:**
  - **Failed-high means it returns a value that is larger than its upper bound $beta$.**
  - **Failed-low means it returns a value that is smaller than its lower bound $alpha$.**
- **Null or Zero window search:**
  - **Using alpha-beta search with the window [m,$m+1$].**
  - **The result can be either failed-high or failed-low.**
  - **Failed-high means the return value is at least $m+1$.**
    - ▷ ***Equivalent to $TEST(p, m, >)$ is true.***
  - **Failed-low means the return value is at most m.**
    - ▷ ***Equivalent to $TEST(p, m, >)$ is false.***

Alpha-Beta + Scout

- **Intuition:**
  - Try to incooperate SCOUT and alpha-beta together.
  - The searching window of alpha-beta if properly set can be used as TEST in SCOUT.
  - Using a searching window is better than using a single bound as in SCOUT.
  - Can also apply alpha-beta cut if it applies.
- **Modifications to the SCOUT algorithm:**
  - Traverse the tree with two bounds as the alpha-beta procedure does.
    - ▷ *A searching window.*
    - ▷ *Use the current best bound to guide the TEST value.*
  - Use a fail soft version to get a better result when the returned value is out of the window.

The NegaScout Algorithm – MiniMax (1/2)

- **Algorithm $F4'$(position p, value $alpha$, value $beta$, integer $depth$)**
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$ // a terminal node
    or $depth = 0$ // $depth$ is the remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // apply heuristic here
  - then return $f(p)$ else
    begin
    - ▷ $m := -\infty$ // m *is the current best lower bound; fail soft*
      $m := \max\{m, G4'(p_1, alpha, beta, depth - 1)\}$ // *the first branch*
      if $m \geq beta$ *then return*(m) // *beta cut off*
    - ▷ *for* $i := 2$ *to* d *do*
    - ▷ $t := G4'(p_i, m, m + 1, depth - 1)$ // *null window search*
    - ▷ *if* $t > m$ *then* // *failed-high*
      - *if* ($depth < 3$ *or* $t \geq beta$)
      - *then* $m := t$
      - *else* $m := G4'(p_i, t, beta, depth - 1)$ // *re-search*
    - ▷ *if* $m \geq beta$ *then return*(m) // *beta cut off*

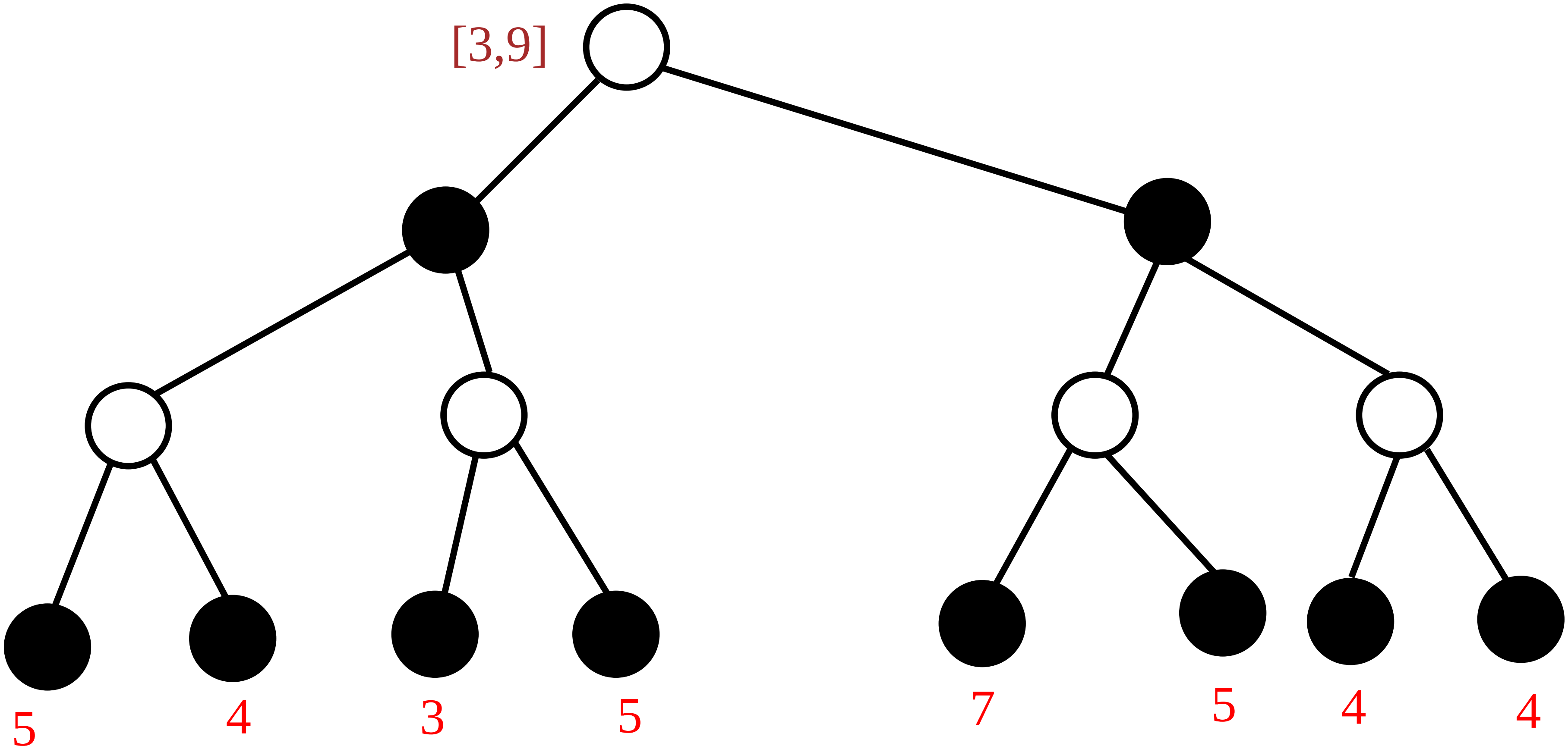
    end
  - return m

The NegaScout Algorithm – MiniMax (2/2)

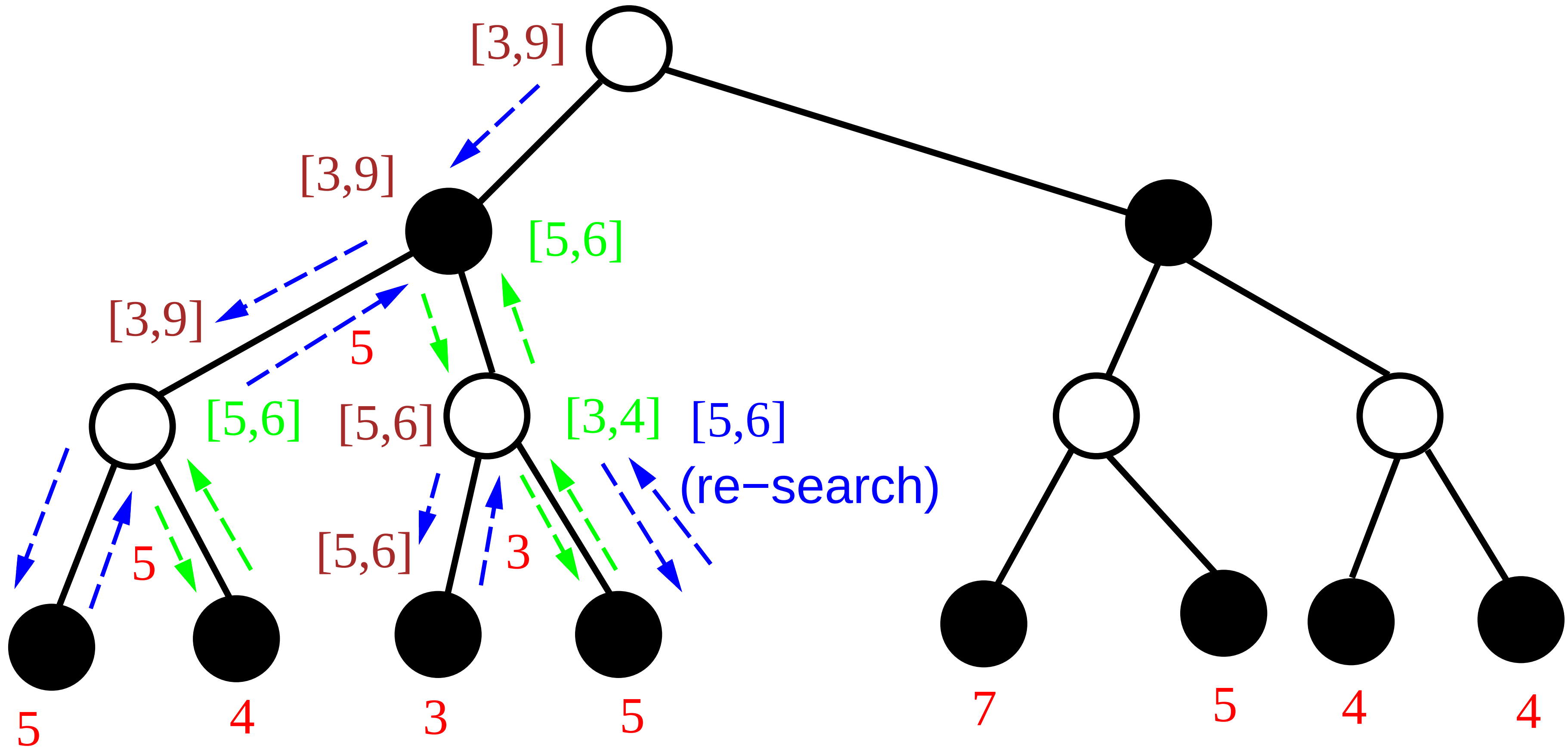
- **Algorithm** $G4'$(**position** p, **value** $alpha$, **value** $beta$, **integer** $depth$)
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$ // a terminal node
    or $depth = 0$ // $depth$ is the remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // apply heuristic here
  - then return $f(p)$ else
    begin
    - ▷ $m = \infty$ // m *is the current best upper bound; fail soft*
      $m := \min\{m, F4'(p_1, alpha, beta, depth - 1)\}$ // *the first branch*
      if $m \leq alpha$ *then return*(m) // *alpha cut off*
    - ▷ *for* $i := 2$ *to* d *do*
    - ▷ $t := F4'(p_i, m, m + 1, depth - 1)$ // *null window search*
    - ▷ *if* $t <= m$ *then* // *failed-low*
      if ($depth < 3$ *or* $t \leq alpha$)
      then $m := t$
      else $m := F4'(p_i, alpha, t, depth - 1)$ // *re-search*
    - ▷ *if* $m \leq alpha$ *then return*(m) // *alpha cut off*

    end
  - return m

NegaScout – MiniMax version (1/2)



NegaScout – MiniMax version (2/2)



The NegaScout Algorithm

- Use Nega-MAX format.
- Algorithm $F4$(position p, value $alpha$, value $beta$, integer $depth$)
  - determine the successor positions $p_1, \ldots, p_d$
  - if $d = 0$ // a terminal node
    or $depth = 0$ //$depth$ is the remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // apply heuristic here
  - then return $h(p)$ else
    - ▷ $m := -\infty$ // *the current lower bound; fail soft*
    - ▷ $n := beta$ // *the current upper bound*
    - ▷ ***for*** $i := 1$ ***to*** d ***do***
    - ▷ ***9:*** $t := -F4(p_i, -n, -max\{alpha, m\}, depth - 1)$
    - ▷ ***10:*** ***if*** $t > m$ ***then***
      11: ***if*** $(n = beta$ ***or*** $depth < 3$ ***or*** $t \geq beta)$
      12: ***then*** $m := t$
      13: ***else*** $m := -F4(p_i, -beta, -t, depth - 1)$ // ***re-search***
    - ▷ ***14:*** ***if*** $m \geq beta$ ***then return***(m) // ***cut off***
    - ▷ ***15:*** $n := max\{alpha, m\} + 1$ // ***set up a null window***
  - return m

Search behaviors (1/3)

- **If the depth is enough or it is a terminal position, then stop searching further.**
  - **Return $h(p)$ as the value computed by an evaluation function.**
  - **Note:**

$$h(p) = \begin{cases} f(p) & \text{if depth of } p \text{ is 0 or even} \\ -f(p) & \text{if depth of } p \text{ is odd} \end{cases}$$

- **Fail soft version.**
- **For the first child p_1, search using the normal alpha beta window..**
  - **line 9: normal window for the first child**
  - **the initial value of m is $-\infty$, hence $-max\{alpha, m\} = -alpha$**
    - ▷ *m is the current best value*
  - **that is, searching with the normal window $[alpha, beta]$**

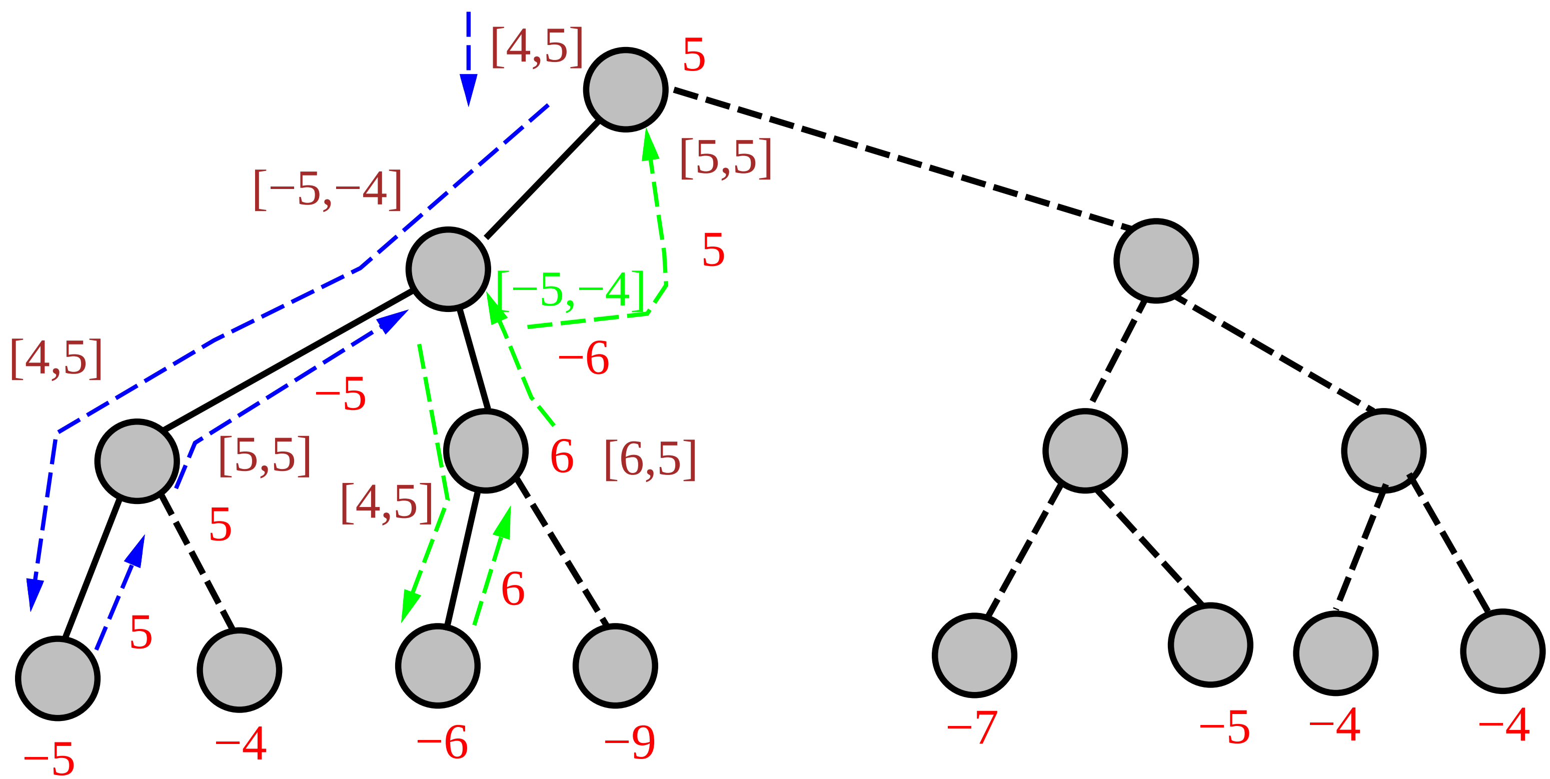
Search behaviors (2/3)

- **For the second child and beyond p_i, $i > 1$, first perform a null window search for testing whether m is the answer.**
  - **line 9: a null-window of $[m, m+1]$ searches for the second child and beyond.**
    - ▷ *m is best value obtained so far*
    - ▷ *m's value will be first set at line 12 because $n = beta$*
    - ▷ *The null window is set at line 15.*
  - **line 11:**
    - ▷ *$n = beta$: we are at first iteration.*
    - ▷ *$depth < 3$: on a smaller depth subtree, i.e., depth at most 2, NegaScout always returns the best value.*
    - ▷ *$t \geq beta$: we have obtained a good enough value from the failed-soft version to guarantee a beta cut.*

Search behaviors (3/3)

- **For the second child and beyond p_i, $i > 1$, first perform a null window search for testing whether m is the answer.**
  - **line 11: on a smaller depth subtree, i.e., depth at most 2, NegaScout always returns the best value.**
    - ▷ ***Normally, no need to do alpha-beta or any enhancement on very small subtrees.***
    - ▷ ***The overhead is too large on small subtrees.***
  - **line 13: re-search when the null window search fails high.**
    - ▷ ***The value of this subtree is at least t.***
    - ▷ ***This means the best value in this subtree is more than m, the current best value.***
    - ▷ ***This subtree must be re-searched with the the window $[t, beta]$.***
  - **line 14: the normal pruning from alpha-beta.**

Example for NegaScout



Refinements

- When a subtree is re-searched, it is best to use information on the previous search to speed up the current search.
  - Restart from the position that the value t is returned.
- Maybe want to re-search using the normal alpha-beta procedure.
- $F4$ runs much better with a good move ordering and transposition tables.
  - Order the moves in a priority list.
  - Reduce the number of re-searches.

Performances

- **Experiments done on a uniform random game tree [Reinefeld 1983].**
  - **Normally superior to alpha-beta when searching game trees with branching factors from 20 to 60.**
  - **Shows about 10 to 20% of improvement.**

Comments

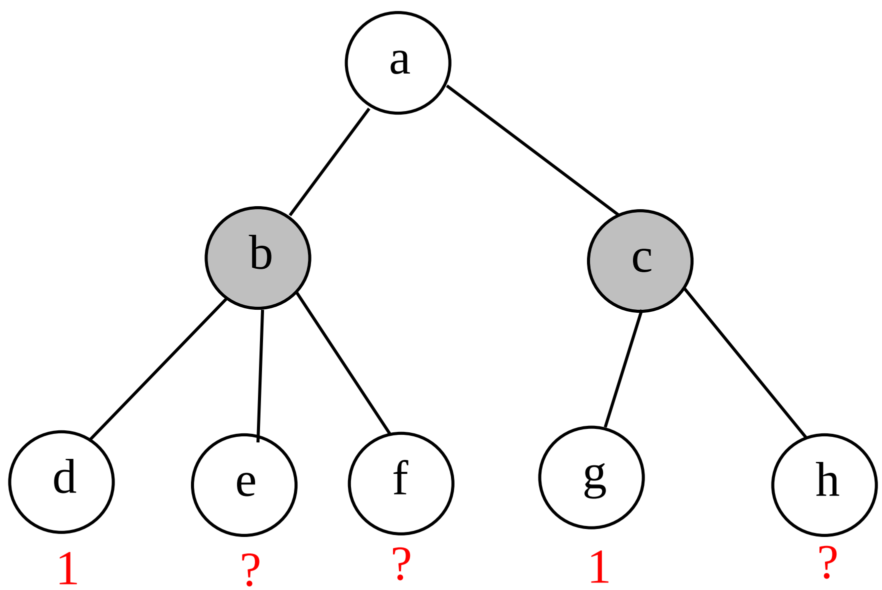
- **Incooperating both SCOUT and alpha-beta.**
- **Used in state-of-the-art game search engines.**
- **The first search, though maybe unsuccessful, can provide useful information in the second search.**
  - **Information can be stored and then be reused.**

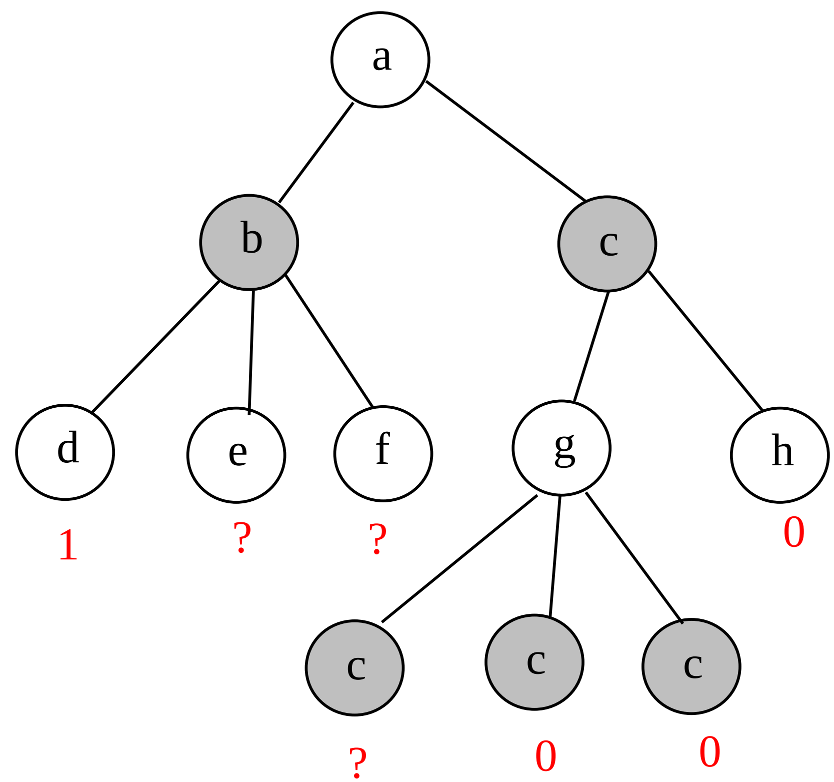
Ideas for new search methods

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





Proof or Disproof Number

- **Assign a proof number and a disproof number to each node u in a binary valued game tree.**
  - $proof(u)$**: 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.**

Proof Number: Definition

- u **is a leaf:**
  - **If** $value(u)$ **is unknown, then** $proof_v(u)$ **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 children** $u_1, \ldots, u_k$**:**
  - **if** u **is a MAX node,**

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

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

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

Disproof Number: Definition

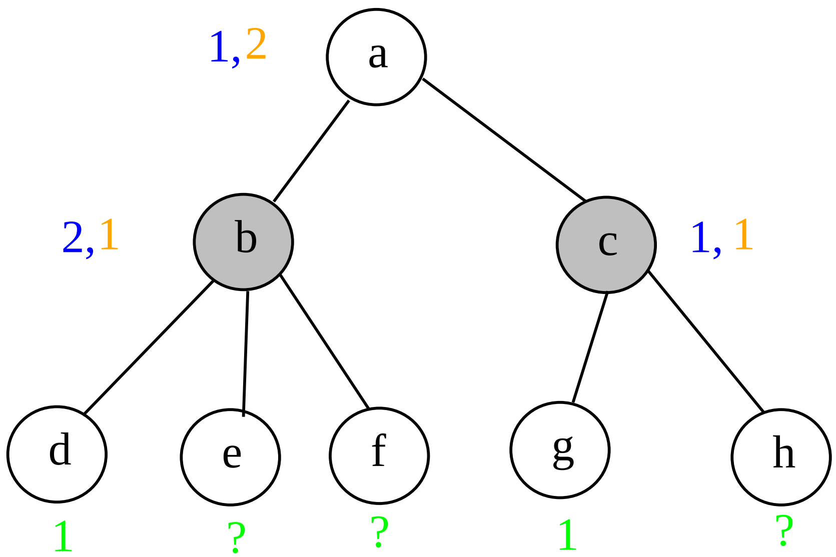
- u **is a leaf:**
  - **If** $value(u)$ **is unknown, then** $proof_v(u)$ **is cost of evaluating** u**.**
  - **If** $value(u)$ **is 1, then** $disproof(u) = \infty$**.**
  - **If** $value(u)$ **is 0, then** $disproof(u) = 0$**.**
- u **is an internal node with children** $u_1, \ldots, u_k$**:**
 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

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

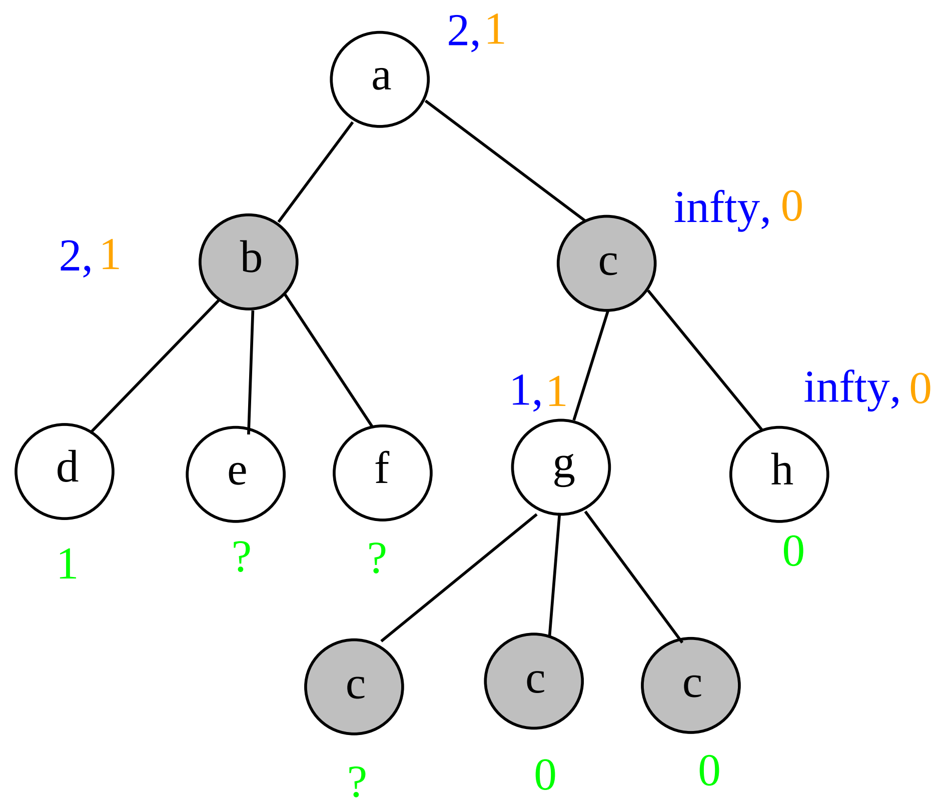
  - **if** u **is a MIN node,**

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

Illustrations



proof number, disproof number



proof number, disproof number

How to Use these Numbers

- **If the numbers are known in advance, then from the root, we search a child u with the value equals to $\min\{proof(root), disproof(root)\}$.**
  - **Then we find a path from the root towards a leaf recursively as follows,**
    - *if we try to prove it, then pick a child with the least proof number for a MAX node, and pick any node that has a chance to be proved for a MIN node.*
    - *if we try to disprove it, then pick a child with the least disproof number for a MIN node, and pick any node that has a chance to be disproved for a MAX node.*
- **Assume each leaf takes a lot of time to evaluate.**
  - **For example, the game tree represents an open game tree or an endgame tree.**
  - **Depends on the results we have so far, pick the next leaf to prove or disprove.**
- **Need to able to update these numbers on the fly.**

PN-search: algorithm

- *loop:* **Compute or update proof and disproof numbers for each node in a bottom up fashion.**
  - **If** $proof(root) = 0$ **or** $disproof(root) = 0$**, then we are done, otherwise**
    - $proof(root) \le disproof(root)$: ***we try to prove it.***
    - $proof(root) > disproof(root)$: ***we try to disprove it.***
- $u \leftarrow root$; $\{*$ **find the 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;***
    - ***if current is a MIN node, then***
      $u \leftarrow$ ***leftmost child of*** u ***with a non-zero proof number;***
  - **if we try to disprove, then**
    - ***while*** u ***is not a leaf do***
    - ***if*** u ***is a MAX node, then***
      $u \leftarrow$ ***leftmost child of*** u ***with a non-zero disproof number;***
    - ***if current is a MIN node, then***
      $u \leftarrow$ ***leftmost child of*** u ***with the smallest non-zero disproof number;***
- **Prove or disprove** u**; go to** $loop$**;**

Multi-Valued game Tree

- **The values of the leaves may not be binary.**
  - **Assume the values are non-negative integers.**
  - **Note: it can be in any finite countable domain.**
- **Revision of the proof and disproof numbers.**
  - $proof_v(u)$**: the minimum number of leaves needed to visited in order for the value of** u **to** $\geq v$**.**
    - ▷ $proof(u) = proof_1(u)$.
  - $disproof_v(u)$**: the minimum number of leaves needed to visited in order for the value of** u **to** $< v$**.**
    - ▷ $disproof(u) = disproof_1(u)$.

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 children** $u_1, \ldots, u_k$**:**
  - **if** u **is a MAX node,**

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

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

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

Multi-valued Disproof Number

- u **is a leaf:**
  - **If** $value(u)$ **is unknown, then** $proof_v(u)$ **is cost of evaluating** u**.**
  - **If** $value(u) \geq v$ **is 1, then** $disproof_v(u) = \infty$**.**
  - **If** $value(u) < v$ **is 0, then** $disproof_v(u) = 0$**.**
- u **is an internal node with children** $u_1, \ldots, u_k$**:**
  - **if** u **is a MAX node,**

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

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

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

Revised PN-search(v): algorithm

- *loop:* **Compute or update proof$_v$ and disproof$_v$ numbers for each node in a bottom up fashion.**
  - **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 the 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;***
    - ▷   ***if current is a MIN node, then***
      $u \leftarrow$ ***leftmost child of*** u ***with a non-zero proof$_v$ number;***
  - **if we try to disprove, then**
    - ▷ ***while*** u ***is not a leaf do***
    - ▷   ***if*** u ***is a MAX node, then***
      $u \leftarrow$ ***leftmost child of*** u ***with a non-zero disproof$_v$ number ;***
    - ▷   ***if current is a MIN node, then***
      $u \leftarrow$ ***leftmost child of*** u ***with the smallest non-zero disproof$_v$ number;***
- **Prove or disprove** u**; go to** $loop$**;**

Multi-valued PN-search: algorithm

- **When the values of the leaves are not binary, use an open value binary search to find an upper bound of the value.**
  - **Set the initial value of v to be 1.**
  - $loop$**: PN-search(**v**)**
    - ▷ ***Prove the value of the search tree is*** $\geq v$ ***or disprove it by showing it is*** $< v$***.***
  - **If it is proved, then double the value of** v **and go to** $loop$ **again.**
  - **If it is disproved, then the true value of the tree is between** $\lfloor v/2 \rfloor$ **and** $v - 1$**.**
  - $\{*$ **Use a binary search to find the exact returned value of the tree.** $*\}$
  - $low \leftarrow \lfloor v/2 \rfloor$**;** $high \leftarrow v - 1$**;**
  - **while** $low \leq high$ **do**
    - ▷ ***if*** $low = high$***, then return*** low ***as the tree value***
    - ▷ $mid \leftarrow \lfloor (low + high)/2 \rfloor$
    - ▷ ***PN-search(***mid***)***
    - ▷ ***if it is disproved, then*** $high \leftarrow mid - 1$
    - ▷ ***else if it is proved, then*** $low \leftarrow mid$

Comments

- **Appears to be good for certain searching certain game trees.**
  - **Find the easiest way to prove or disprove a conjecture.**
  - **A dynamic strategy depends on work has been done so far.**
- **Take into consideration the fact that some nodes may need more time to process than the other nodes.**

References and further readings

* **J. Pearl. Asymptotic properties of minimax trees and game-searching procedures.** *Artificial Intelligence*, **14(2):113–138, 1980.**
* **A. Reinefeld. An improvement of the scout tree search algorithm.** *ICCA Journal*, **6(4):4–14, 1983.**
* **L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search.** *Artificial Intelligence*, **66(1):91–124, 1994.**