Scout and NegaScout

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

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

Abstract

- **It looks like alpha-beta pruning is the best we can do for an exact 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.**
    - *Alpha-beta cut algorithm is one way to make sure of this by returning an exact value.*
    - *Is there a way to search a tree by only returning a bound?*
    - *Is searching with a bound faster than searching exactly?*
- **Similar intuition holds for a MIN node.**

SCOUT procedure

- **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 [Judea Pearl 1980].**
- **High level idea:**
  - **While searching a branch T_i of a MAX node, if we have already obtained a lower bound v_ℓ.**
    - ▷ ***First TEST whether it is possible for T_i to return something greater than v_ℓ.***
    - ▷ ***If FALSE, then there is no need to search T_i.***
      ⇒ This is called fails the test.
    - ▷ ***If TRUE, then search T_i.***
      ⇒ This is called passes the test.
  - **While searching a branch T_j of a MIN node, if we have already obtained an upper bound v_u**
    - ▷ ***First TEST whether it is possible for T_j to return something smaller than v_u.***
    - ▷ ***If FALSE, then there is no need to search T_j.***
      ⇒ This is called fails the test.
    - ▷ ***If TRUE, then search T_j.***
      ⇒ This is called passes the test.

How to TEST $> v$

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

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

- determine the successor positions $p_1, \ldots, p_b$ of p
- if $b = 0$, then // terminal
  - ▷ *if $f(p) > v$ then // f(): evaluation function*
  - ▷ *return TRUE*
  - ▷ *else return FALSE*
- if p is a MAX node, then
  - for $i := 1$ to b 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 b 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 $> v$

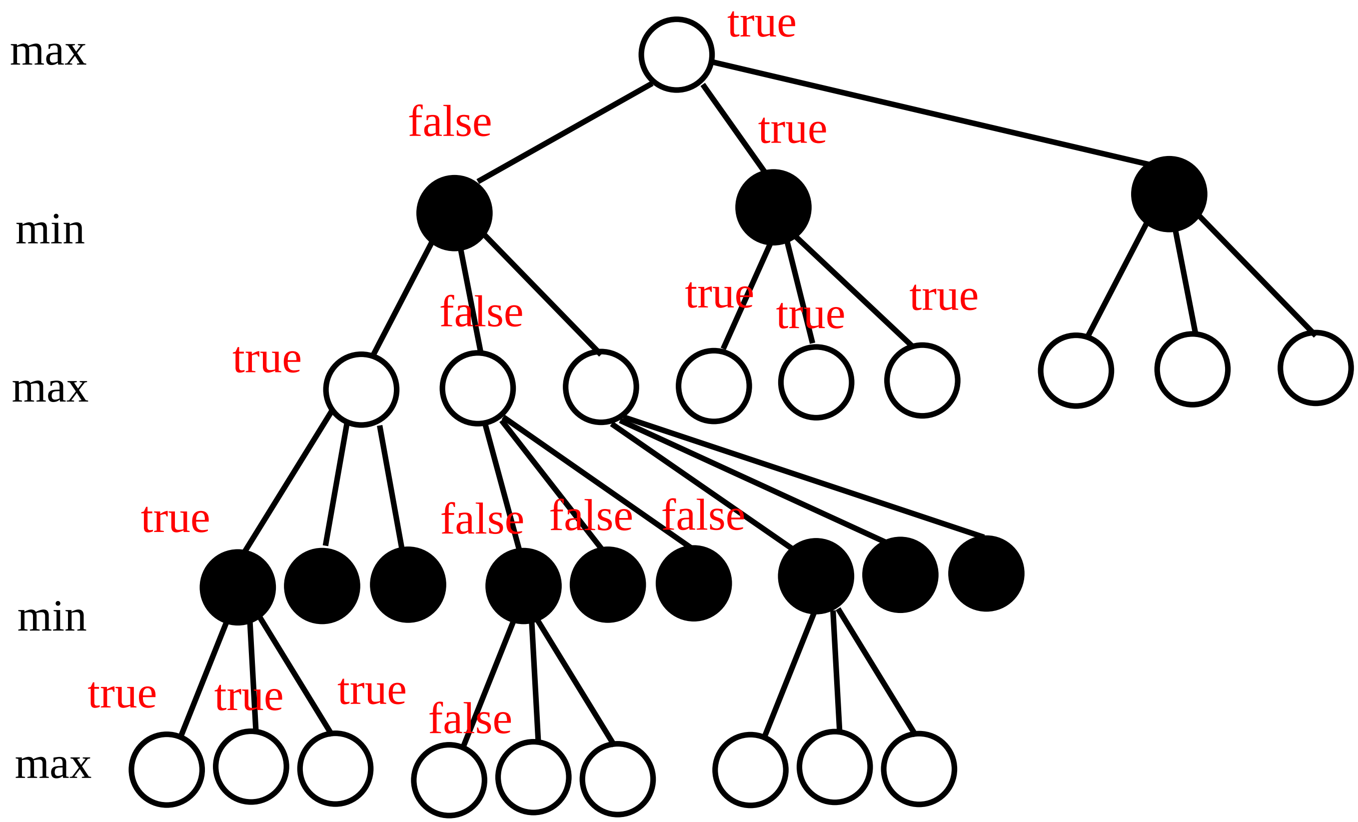
How to TEST $< v$

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

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

- **determine the successor positions $p_1, \ldots, p_b$ of p**
- **if $b = 0$, then // terminal**
  - ▷ ***if $f(p) < v$ then // f(): evaluation function***
  - ▷ ***return TRUE***
  - ▷ ***else return FALSE***
- **if p is a MAX node, then**
  - **for $i := 1$ to b do**
    - ▷ ***if TEST$_<$(p_i, v) is FALSE, then***
      return FALSE // fail 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 b 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 $\geq v$**

Illustration of TEST$_{>}$



Short circuit operations for TEST$_{>}$

- **For a MAX node:**
  - if a branch is TRUE, then there is no need to do further testing;
  - if a branch is FALSE, then we need to do more testing on other branches.
  - It is better to test branches with better probabilities of being TRUE first.
- **For a MIN node:**
  - if a branch is FALSE, then there is no need to do further testing;
  - if a branch is TRUE, then we need to do more testing on other branches.
  - It is better to test branches with better probabilities of being FALSE first.

How to TEST — Discussions

- **Sometimes it may be needed to test for "$\geq v$", or "$\leq v$".**
  - **$\boxed{\text{TEST}_{>}(p,v) \text{ is TRUE}} \equiv \boxed{\text{TEST}_{\leq}(p,v) \text{ is FALSE}}$**
  - **$\boxed{\text{TEST}_{>}(p,v) \text{ is FALSE}} \equiv \boxed{\text{TEST}_{\leq}(p,v) \text{ is TRUE}}$**
  - **$\boxed{\text{TEST}_{<}(p,v) \text{ is TRUE}} \equiv \boxed{\text{TEST}_{\geq}(p,v) \text{ is FALSE}}$**
  - **$\boxed{\text{TEST}_{<}(p,v) \text{ is FALSE}} \equiv \boxed{\text{TEST}_{\geq}(p,v) \text{ is TRUE}}$**
- **Practical consideration:**
  - **Set a depth limit and evaluate the position's value when the limit is reached.**

Main SCOUT procedure

Algorithm SCOUT(position p)

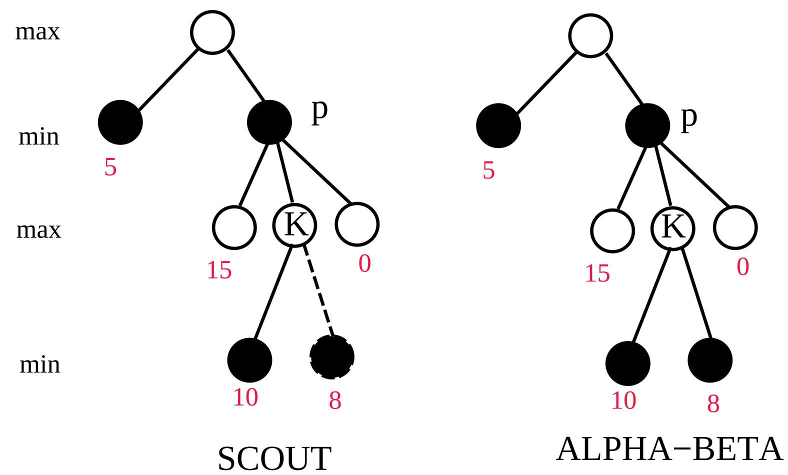
- **determine the successor positions** $p_1, \ldots, p_b$
- **if** $b = 0$**, then return** $f(p)$
- **else** $v = SCOUT(p_1)$ **// SCOUT the first branch**
- **if** p **is a MAX node**
  - **for** $i := 2$ **to** b **do**
    - ▷ ***if TEST***$_>(p_i, v)$ ***is TRUE, // TEST first for the rest of the 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** b **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

Discussions for SCOUT (1/3)

- **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 that 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 that p_i passes the test if TEST$_<$(p_i, v) is TRUE.**

Discussions for SCOUT (2/3)

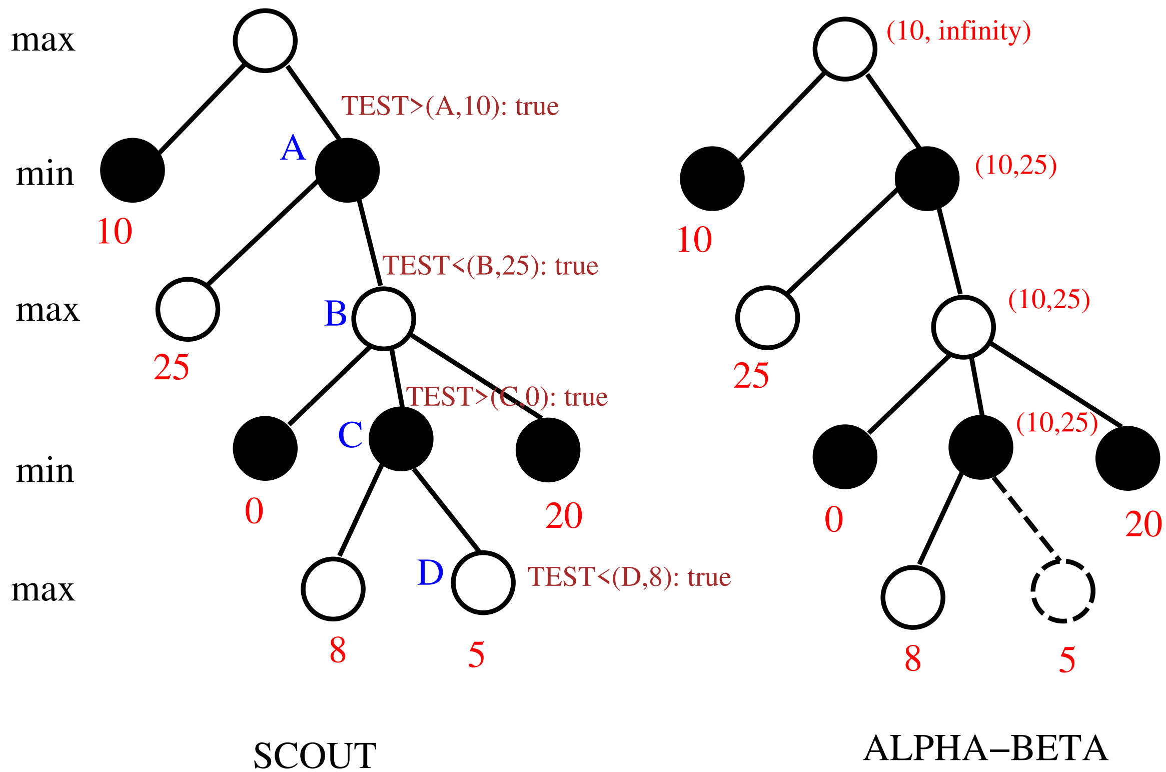
- **TEST which is called by SCOUT may visit less nodes than that of alpha-beta.**



- **Assume TEST$_>$(p,5) is called by the root after the first branch of the root 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 rooted p.***
- **Alpha-beta needs to visit K's second branch.**

Discussions for SCOUT (3/3)

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



Number of nodes visited (1/4)

- **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 b, the number of nodes evaluated is $\Omega(b^{\ell/2})$ where ℓ is the depth of the tree.*
- **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 b, the number of nodes evaluated is $\Omega(b^{\ell/2})$.*

Number of nodes visited (2/4)

- **Assumptions:**
  - **Assume a full complete b-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 $\frac{b^{\ell+1}-1}{b-1}$.**
- H_1**: the minimum number of nodes visited by TEST when it returns TRUE.**

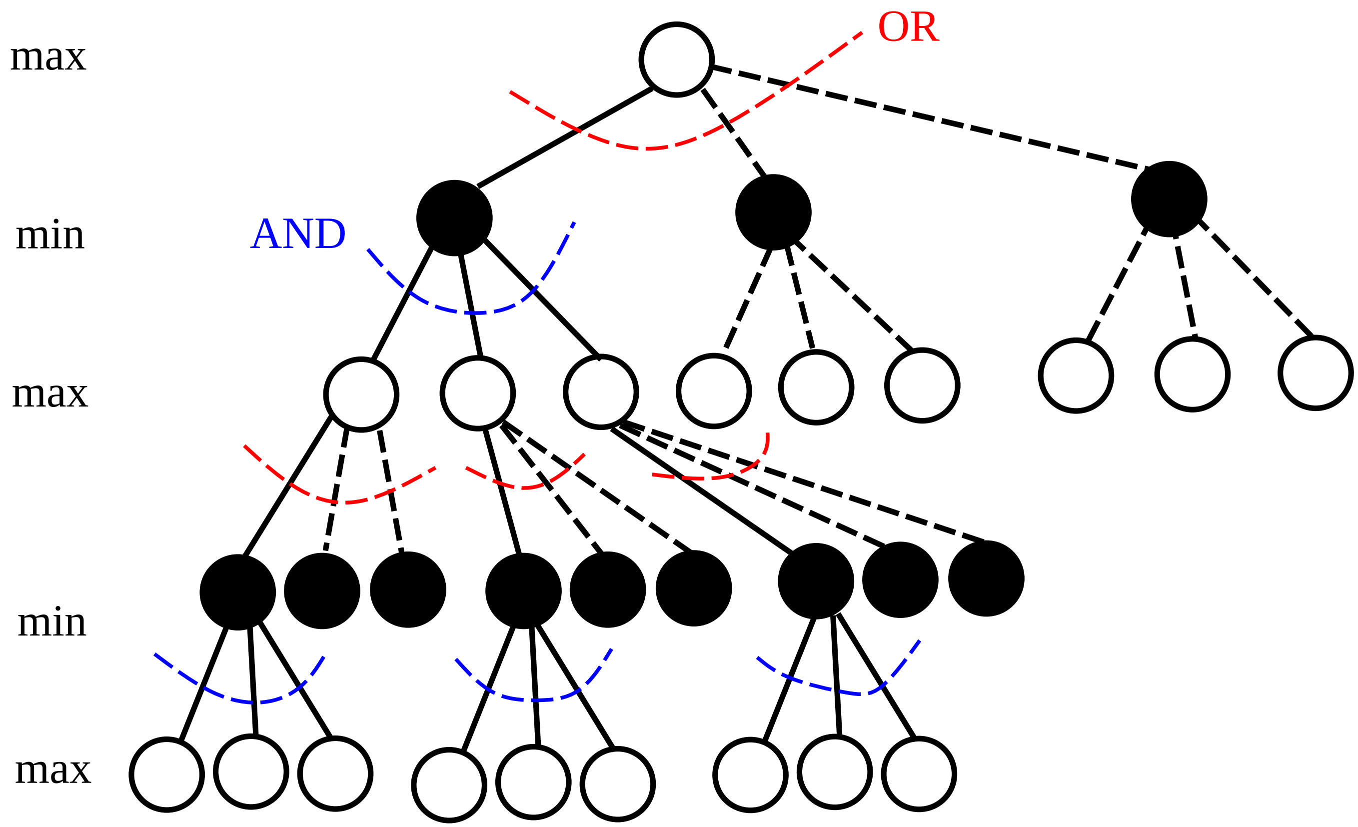
$$
\begin{aligned}
H_1 = &\ 1 + 1 + b + b + b^2 + b^2 + b^3 + b^3 + \cdots + b^{\ell/2-1} + b^{\ell/2-1} + b^{\ell/2} \\
= &\ 2 \cdot (b^0 + b^1 + \cdots + b^{\ell/2}) - b^{\ell/2} \\
= &\ 2 \cdot \frac{b^{\ell/2+1}-1}{b-1} - b^{\ell/2}
\end{aligned}
$$

Number of nodes visited (3/4)

- **Assumptions:**
  - **Assume a full complete b-ary tree with depth ℓ where ℓ is even.**
  - **The depth of the root, which is a MAX node, is 0.**
- H_2: **the minimum number of nodes visited by alpha-beta.**

$$
\begin{aligned}
H_2 &= \sum_{i=0}^{\ell}(b^{\lceil i/2\rceil} + b^{\lfloor i/2\rfloor} - 1) \\
&= \sum_{i=0}^{\ell} b^{\lceil i/2\rceil} + \sum_{i=0}^{\ell} b^{\lfloor i/2\rfloor} - (\ell+1) \\
&= \sum_{i=0}^{\ell} b^{\lceil i/2\rceil} + H_1 - (\ell+1) \\
&= (1 + b + b + \cdots + b^{\ell/2-1} + b^{\ell/2} + b^{\ell/2}) + H_1 - (\ell+1) \\
&= (H_1 - 1 + b^{\ell/2}) + H_1 - (\ell+1) \\
&= 2\cdot H_1 + b^{\ell/2} - (\ell+2) \\
&\geq 2\cdot H_1 \text{ if } b > 3
\end{aligned}
$$

Number of nodes visited (4/4)



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 pruning as long as some relative move orderings are "good."**
    - ▷ *The moving orders of your children and the children of your ancestor who is odd level up "together" decide a cut-off.*
  - **The bounds are updated during searching.**
    - ▷ *Sometimes, a deep alpha-beta cut-off occurs because a bound found from your ancestor a distance away.*

Performance of SCOUT (1/3)

- **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.**
  - **For every node v in a branch that is not the first visited child of its parent with a depth[1] of ℓ,**
    - ▷ ***every ancestor of v may initiate a TEST to visit v.***
    - ▷ ***It can be visited ℓ times by TEST.***

[1] The depth of the root is defined to be 0.

Performance of SCOUT (2/3)

- **Show great improvements on $depth > 3$ over brute-force methods 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.**
- **Show good improvement over alpha-beta on game trees with certain characteristics.**
- **Experimental data on the game of Kalah show [UCLA Tech Rep UCLA-ENG-80-17, A comparison of the Alpha-Beta and SCOUT algorithms using the game of Kalah, Noe 1980]:**
  - **SCOUT favors "skinny" game trees, that are game trees with high depth-to-width ratios.**
    - ▷ ***Q: why?***
  - **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.***

Performance of SCOUT (3/3)

- **Comparing alpha-beta pruning and SCOUT [Pearl 1984] on uniform game trees:**
  - **Alpha-beta is always better than SCOUT in the experiments using random game trees.**
    - ▷ *In theory, when both are in their best cases, SCOUT cuts out more, but this rarely happens in practice.*
  - **Let $r_{b,d} = \frac{N_{scout}}{N_{AB}}$ where N_{scout} is the nodes searched using SCOUT and N_{AB} is the nodes searched using alpha-beta on depth-d random-valued game trees with a uniform branching factor of b.**
    - ▷ $1 \leq r_{b,d} \leq 1.275$ *for any positive integers* b *and* d.
    - ▷ $r_{b_1,d} \geq r_{b_2,d}$ *if* $b_1 \leq b_2$: *ratio is closer when the branching factor is larger.*
    - ▷ $r_{b,d_1} \geq r_{b,d_2}$ *if* $d_1 \leq d_2$: *ratio is closer when the searching depth is larger.*
    - ▷ $r_{2,20} \sim 1.04$.
    - ▷ $r_{b,20} \sim 1$: *after depth .= 20, the two are almost the same.*

Comments

- **Q1:**
  - **Currently, we use a "feasible" test to decide whether we need to search this branch or not.**
    - ▷ *For example in searching a MAX node, if we have currently found a branch with the value v, then the return value is at least v.*
    - ▷ *If a new branch has a chance of larger than v, then we explore it in details. Otherwise, we skip it.*
    - ▷ *How about a hybrid approach? When to use one instead of the other?*
  - **How about using the idea of "infeasible" test?**
    - ▷ *If a new branch has no chance of larger than v, then we do not explore it in details. Otherwise, we do.*
- **Q2: What can we do with regard to the first branch?**
  - **Can some previous values of some previous positions be used?**

Alpha-beta revisited

- In an alpha-beta search with a window ($alpha$,$beta$):
  - **Failed-high** means it returns a value that is larger than or equal to its upper bound $beta$.
  - **Failed-low** means it returns a value that is smaller than or equal to its lower bound $alpha$.
- **Null** or **Zero window search**:
  - Using alpha-beta search with the window (m,$m+1$).
    - ▷ *Can never happen in a normal alpha-beta pruning when starts with* $(-\infty,\infty)$*.*
  - 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.*
- The above argument works for the original ($F1$), fail hard ($F2$) and fail soft ($F3$) versions of the alpha-beta algorithm.

Behaviors of Null window search

- **When $F1(p, m, m+1, \infty)$ returns $m+1$:**
  - **for the MAX node p, returns immediately after the first child p_i, namely the smallest index i, returning a value $\geq m+1$.**
  - **for the MIN node p_i, every child $p_{i,j}$ returns a value $\geq m+1$**
  - **for each MAX node $p_{i,j}$, returns immediately after the first child $r_{i,j,k}$, namely the smallest index k, returning a value $\geq m+1$.**
  - $\ldots$
- **Exactly like the OR-AND tree shown in TEST$_>$ when TEST is passed.**
- **We can observe similar behaviors when $F1(p, m, m+1, \infty)$ returns m as if TEST is failed.**

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 value used in TEST.***
  - **Use a fail soft version to get a better result when the returned value is out of the window.**

The NegaScout Algorithm – Mini-Max (1/2)

- **Algorithm** $F4'$(**position** p, **value** $alpha$, **value** $beta$, **integer** $depth$)
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // 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* b *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 *is max possible or* $m \geq beta$ *then return*(m) // *beta cut off*

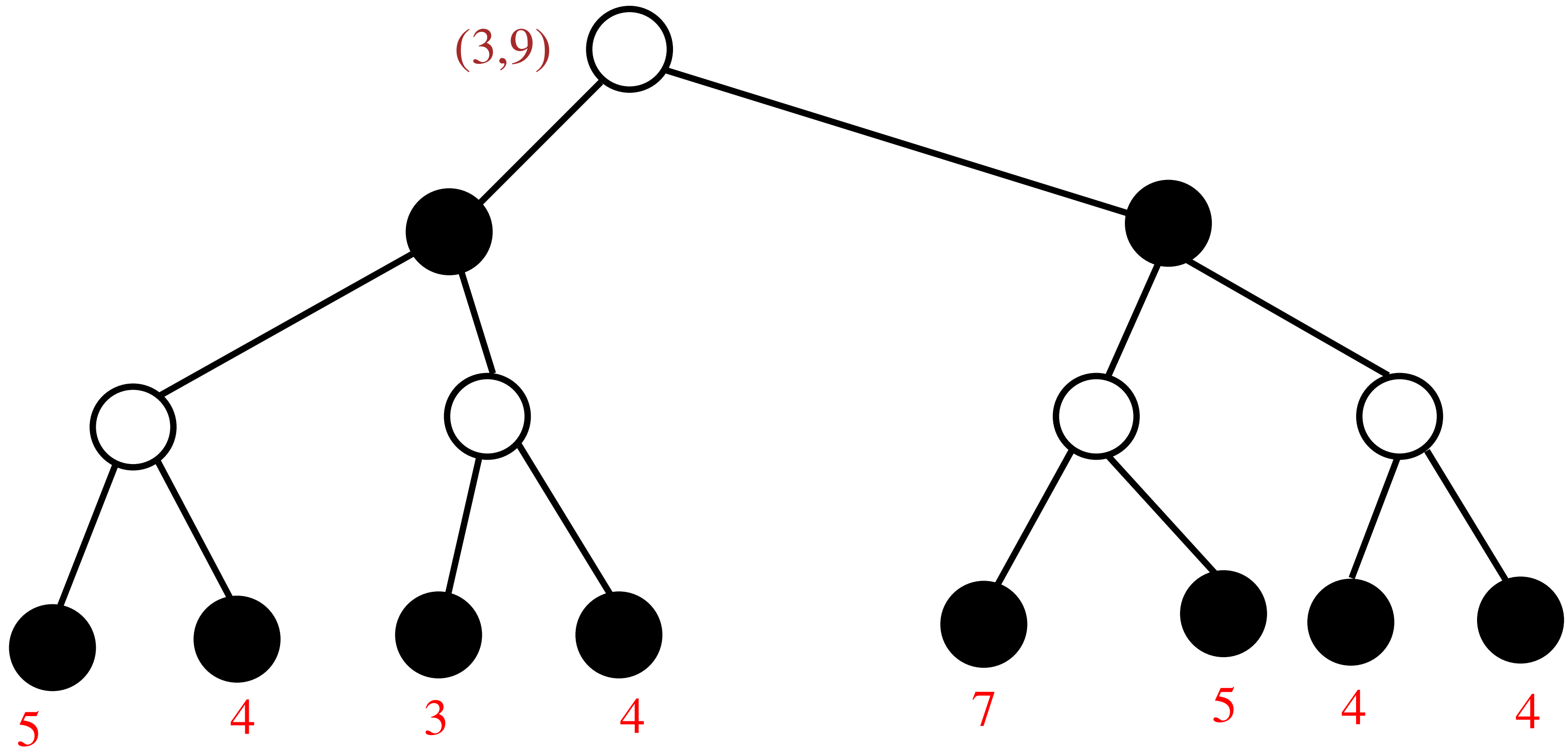
    end
  - return m

The NegaScout Algorithm – Mini-Max (2/2)

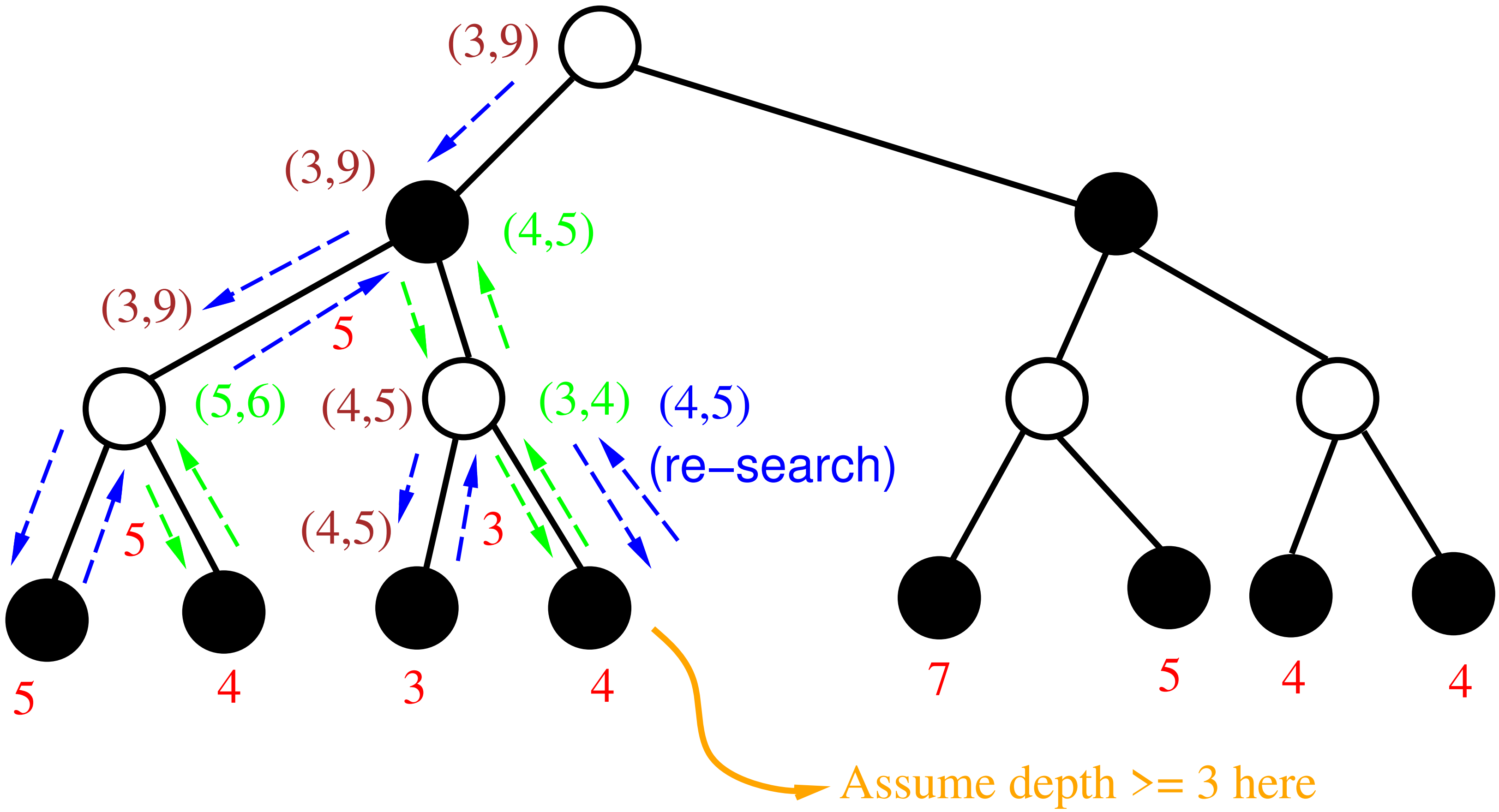
- **Algorithm $G4'$(position p, value $alpha$, value $beta$, integer $depth$)**
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // a terminal node
    or $depth = 0$ // $depth$ is the remaining depth to search
    or time is running up // from timing control
    or some other 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 b do
    - ▷ 9: $t := F4'(p_i, m - 1, m, depth - 1)$ // null window search
    - ▷ 10: if $t < m$ then // failed-low
      11: if ($depth < 3$ or $t \leq alpha$)
      12: then $m := t$
      13: else $m := F4'(p_i, alpha, t, depth - 1)$ // re-search
    - ▷ 14: if m is min possible or $m \leq alpha$ then return(m)// alpha cut off

    end
  - return m

NegaScout – Mini-Max version (1/2)



NegaScout – Mini-Max version (2/2)



The NegaScout Algorithm

- **Use Nega-MAX format.**
- **Algorithm $F4$(position p, value $alpha$, value $beta$, integer $depth$)**
  - **determine the successor positions $p_1, \ldots, p_b$**
  - **if $b = 0$ // 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*** b ***do***
    - ▷ $t := -F4(p_i, -n, -max\{alpha, m\}, depth - 1)$
    - ▷ ***if*** $t > m$ ***then***
      if $(n = beta$ ***or*** $depth < 3$ ***or*** $t \geq beta)$
      then $m := t$
      else $m := -F4(p_i, -beta, -t, depth - 1)$ ***// re-search***
    - ▷ ***if*** m ***is max possible or*** $m \geq beta$ ***then return***(m) ***// cut off***
    - ▷ $n := max\{alpha, m\} + 1$ ***// 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.**
- **Search the first child p_1 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, equivalent to*

      9: $t := -F4(p_i, -beta, -alpha, depth - 1)$

      searching with the normal window $(alpha, beta)$

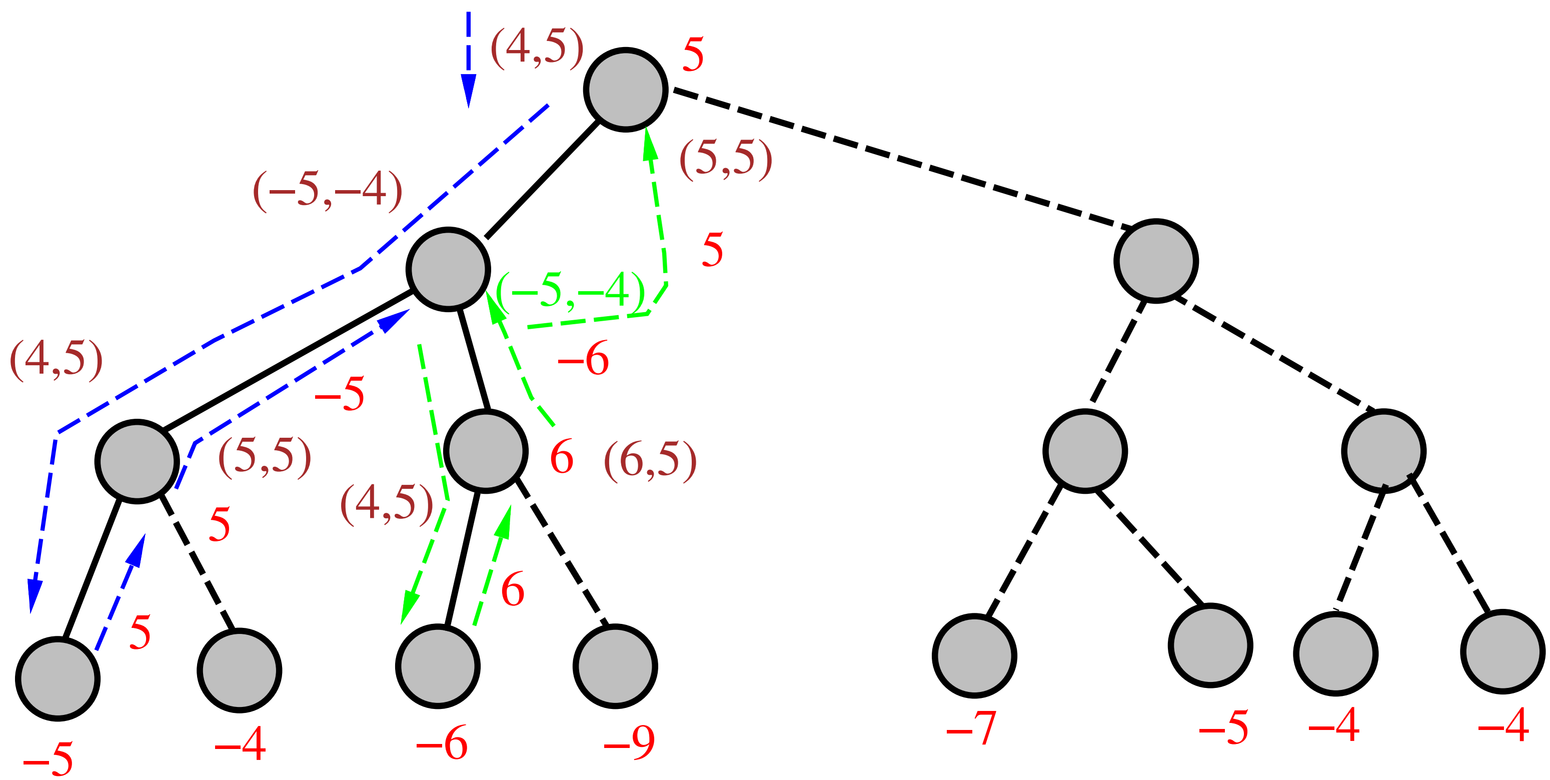
Search behaviors (2/3)

- **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 $(n-1, n)$ searches for the second child and beyond where $n = max\{alpha, m\} + 1$.**
    - ▷ ***m is best value obtained so far***
    - ▷ ***$alpha$ is the previous lower bound***
    - ▷ ***m's value will be first set at line 12 because $n = beta$***
    - ▷ ***The value of $n = max\{alpha, m\} + 1$ is set at line 15.***
  - **line 11:**
    - ▷ ***If $n = beta$, we are at the first iteration.***
    - ▷ ***If $depth < 3$, we are on a smaller depth subtree, i.e., depth at most 2, NegaScout always returns the best value.***
    - ▷ ***If $t \geq beta$, 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 some form of a transposition table which will be introduced later.**
  - **Order the moves in a priority list.**
  - **Reduce the number of re-searching's.**

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 reused.
- Using TEST in SCOUT to do the first search because it has a chance to visit less nodes than that of ALPHA-BETA.

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.
* Noe, T. A comparison of the Alpha-Beta and SCOUT algorithms using the game of Kalah Technical Report UCLA-ENG-80-17, Cognitive Systems Laboratory, University of California, Los Angeles, 1980.
* Pearl, Judea. Heuristics: intelligent search strategies for computer problem solving. Addison-Wesley Longman Publishing Co., Inc., 1984.