Scout and NegaScout

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: One a 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

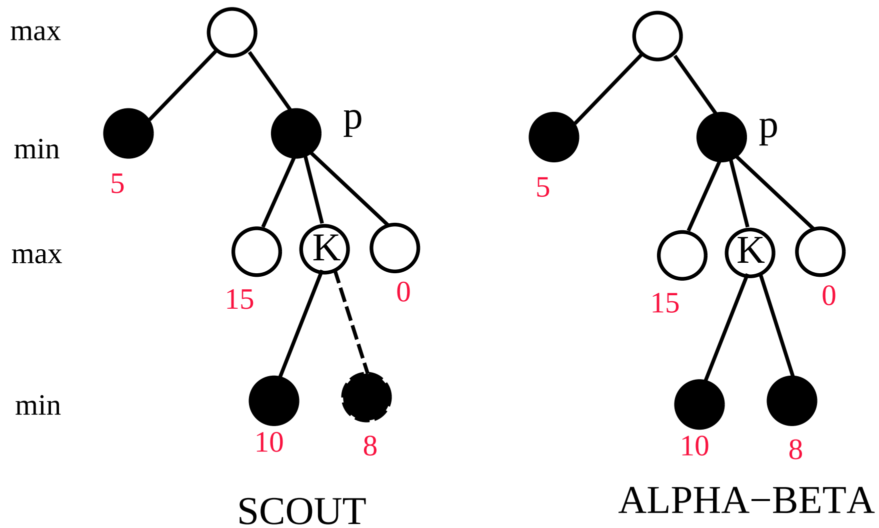
- **procedure TEST(position p, value v, condition $>$)**
  // test whether the value of the branch at p is $> v$
  - **determine the successor positions $p_1, \ldots, p_d$ of p**
  - **if $d = 0$, then // terminal**
    - ▷ ***return TRUE if $f(p) > v$ // f(): evaluating function***
    - ▷ ***return FALSE otherwise***
  - **for $i := 1$ to d do**
    - ▷ ***if p is a MAX node and TEST(p_i, v, $>$) is TRUE, then return TRUE***
    - ▷ ***if p is a MIN node and TEST(p_i, v, $>$) is FALSE, then return FALSE***
  - **if p is a MAX node, then return FALSE**
  - **if p is a MIN node, then return TRUE**
- **Condition can be stated as $\geq$ by properly revising the algorithm.**
  - **For the condition to be $<$ or $\leq$, we need to switch conditions for the MAX and MIN nodes.**
- **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 $p_1, \ldots, p_d$**
  - **if $d = 0$, then return $f(p)$**
  - **else $v = SCOUT(p_1)$ // SCOUT the first branch**
  - **for $i := 2$ to d do**
    // TEST first for the rest of the branches
    - ▷ ***if p is a MAX node and TEST(p_i, v, $>$) is TRUE,***
      then $v = SCOUT(p_i)$ // find the value of this branch
    - ▷ ***if p is a MIN node and TEST(p_i, v, $\geq$) is FALSE,***
      then $v = SCOUT(p_i)$ // find the value of this branch
  - **return v**
- **Note that v is the current best value at any moment.**
  - **for a MAX node, p,**
    - ▷ ***For any $i > 1$, if TEST(p_i, v, $>$) is TRUE, then the value returned by $SCOUT(p_i)$ must be greater than v for a MAX node.***
    - ▷ ***We say the p_i passes the test if TEST(p_i, v, $>$) is TRUE.***
  - **for a MIN node, p,**
    - ▷ ***For any $i > 1$, if TEST(p_i, v, $\geq$) is FALSE, 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, $\geq$) is FALSE.***

Discussions for SCOUT (1/2)

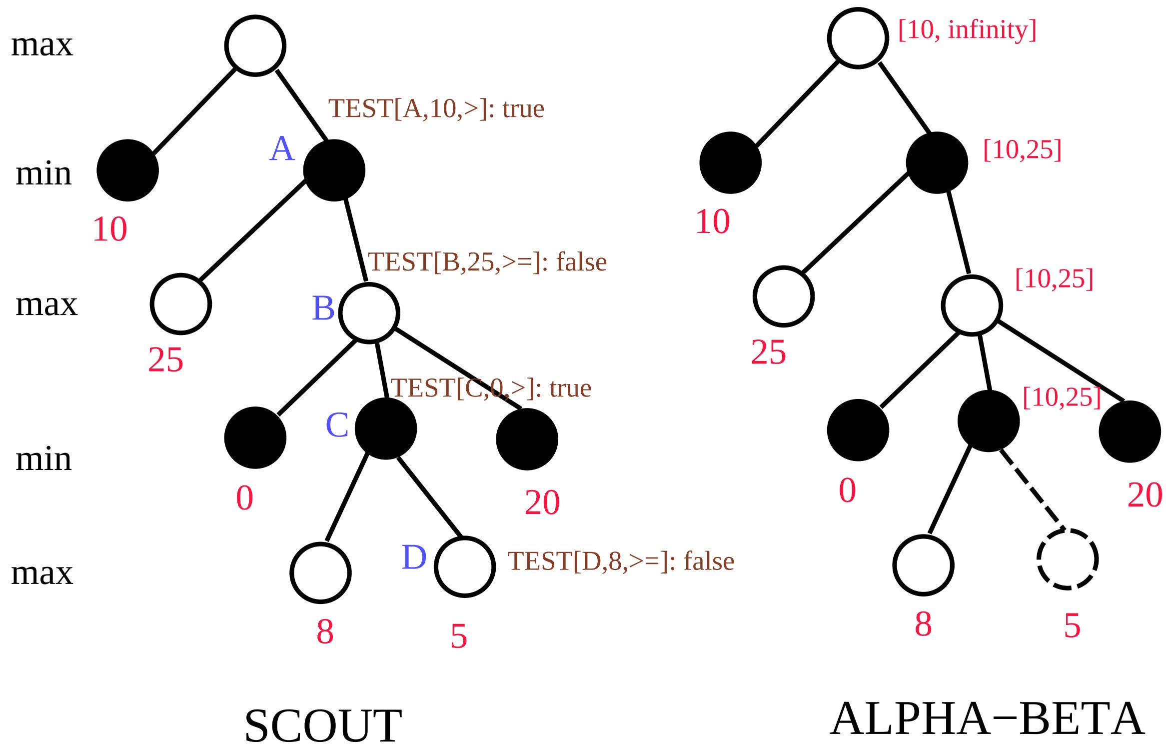
- **TEST may visit less nodes than alpha-beta.**



  - **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 visit a node that is cut off by alpha-beta.**



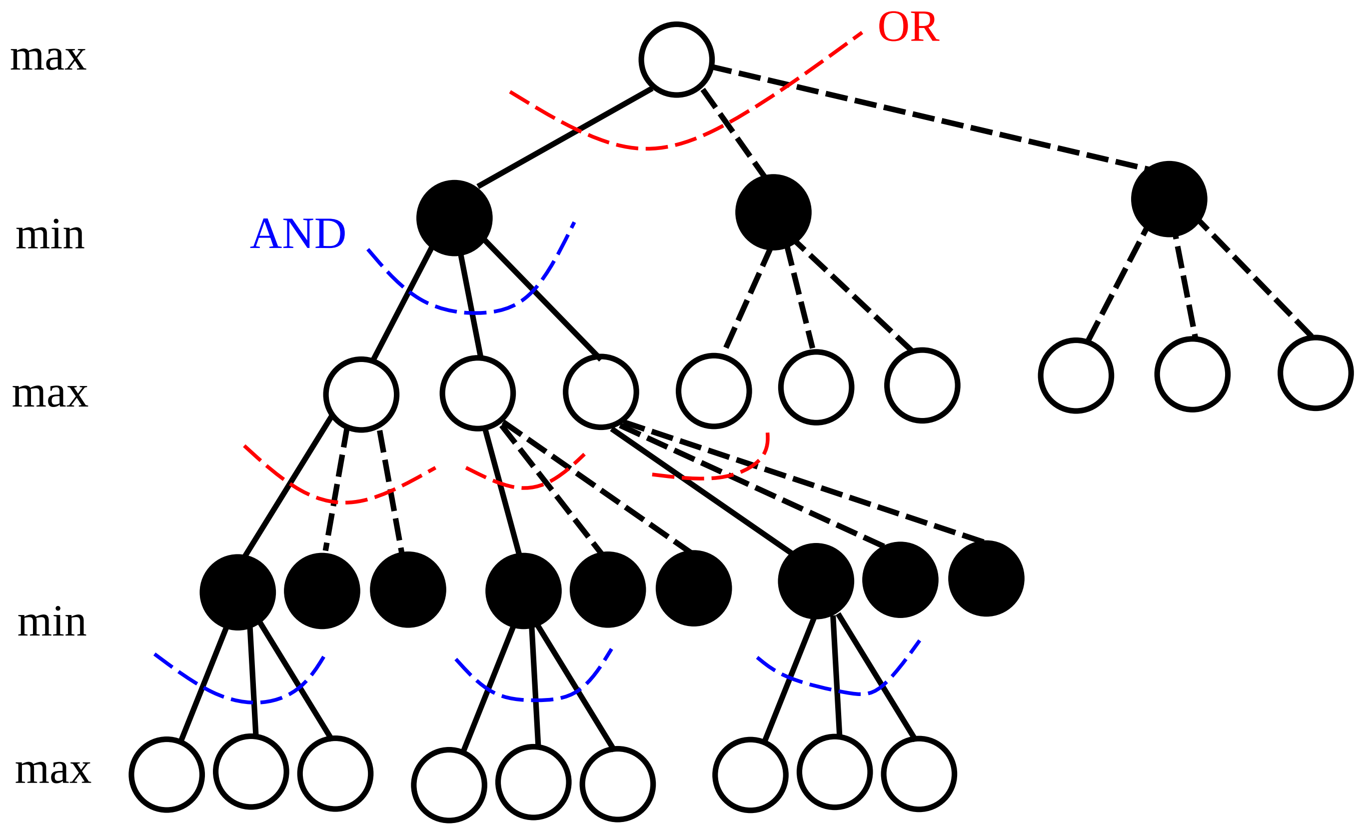
Number of nodes visited (1/3)

- **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.**
  - **Assume 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.**
  - ▷ ***It is*** $1+1+d+d+d^2+d^2+d^3+d^3+\cdots+d^{\ell/2-1}+d^{\ell/2-1}+d^{\ell/2}$.
  - ▷ ***It is*** $2\cdot(d^0+d^1+\cdots+d^{\ell/2})-d^{\ell/2}=2\cdot\frac{d^{\ell/2+1}-1}{d-1}-d^{\ell/2}$.
- **The minimum number of nodes visited by alpha-beta.**
  - ▷ ***It is*** $\sum_{i=0}^{\ell} d^{\lceil i/2\rceil}+d^{\lfloor i/2\rfloor}-1$.
  - ▷ ***It is*** $1+d+(2d-1)+(d^2+d-1)+\cdots+(d^{\ell/2}+d^{\ell/2-1}-1)+(2\cdot d^{\ell/2}-1)$.

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 a 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.**
    - ▷ *Every ancestor of you may initiate a TEST to visit you.*
    - ▷ *It can be visited ℓ times by TEST.*
    - ▷ *Every ancestor of you may pass the TEST and decides to SCOUT you.*
    - ▷ *It can be visited ℓ times by SCOUT.*

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 show [Pearl 1980]:**
  - **SCOUT favors "skinny" games, that are games 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*
    - ▷ *9:* $t := G4'(p_i, m, m + 1, depth - 1)$ // *null window search*
    - ▷ *10:* *if* $t > m$ *then* // *failed-high*
      11: *if* ($depth < 3$ *or* $t \geq beta$)
      12: *then* $m := t$
      13: *else* $m := G4'(p_i, t, beta, depth - 1)$ // *re-search*
    - ▷ *14:* *if* $m \geq beta$ *then return*(m) // *beta cut off*

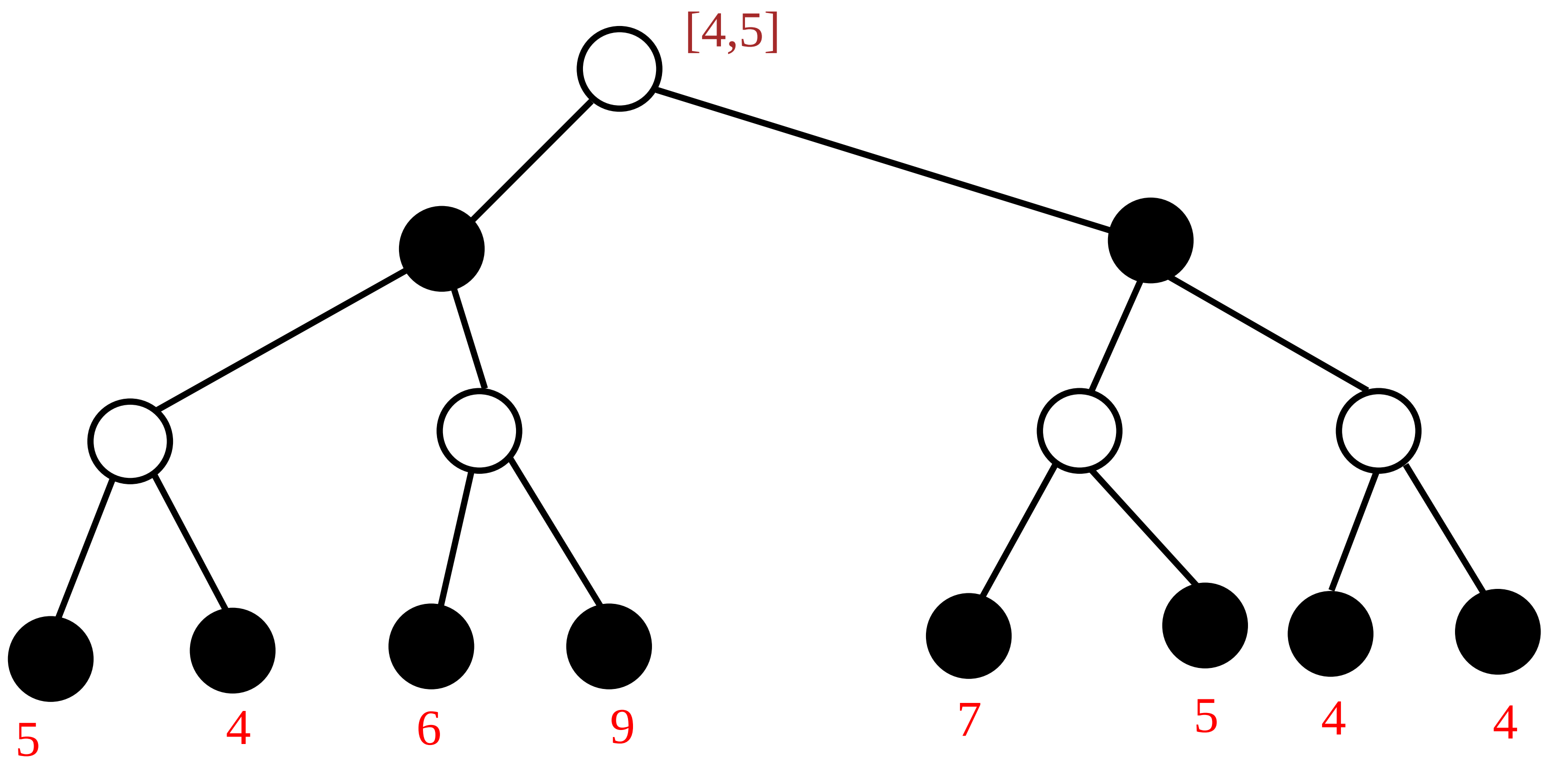
    end
  - return m

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 $g(p)$ 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*
    - ▷ *9:* $t := F4'(p_i, m, m + 1, depth - 1)$ // null window search
    - ▷ *10:* *if* $t <= m$ *then* // failed-low
      11: *if* ($depth < 3$ *or* $t \leq alpha$)
      12: *then* $m := t$
      13: *else* $m := G4'(p_i, alpha, t, depth - 1)$ // re-search
    - ▷ *14:* *if* $m \leq alpha$ *then return*(m) // alpha cut off

    end
  - return m

Example for NegaScout – MiniMax version



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 $f(p)$ else**
    - ▷ $m := -\infty$ ***// the current lower bound; fail soft***
    - ▷ $n := beta$ ***// the current upper bound***
    - ▷ ***for*** $i := 1$ ***to*** d ***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*** $m \geq beta$ ***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 $f(p)$ as the value computed by an evaluation function.**
- **Fail soft version.**
- **For the first child p_1, a normal alpha beta searching window is used.**
  - **line 9: normal alpha-beta search for the first child**
  - **the initial value of m is $-\infty$, hence $-max\{alpha, m\} = -alpha$**
    - ▷ m ***is current best value***
  - **that is, searching with the normal window $[alpha, beta]$**

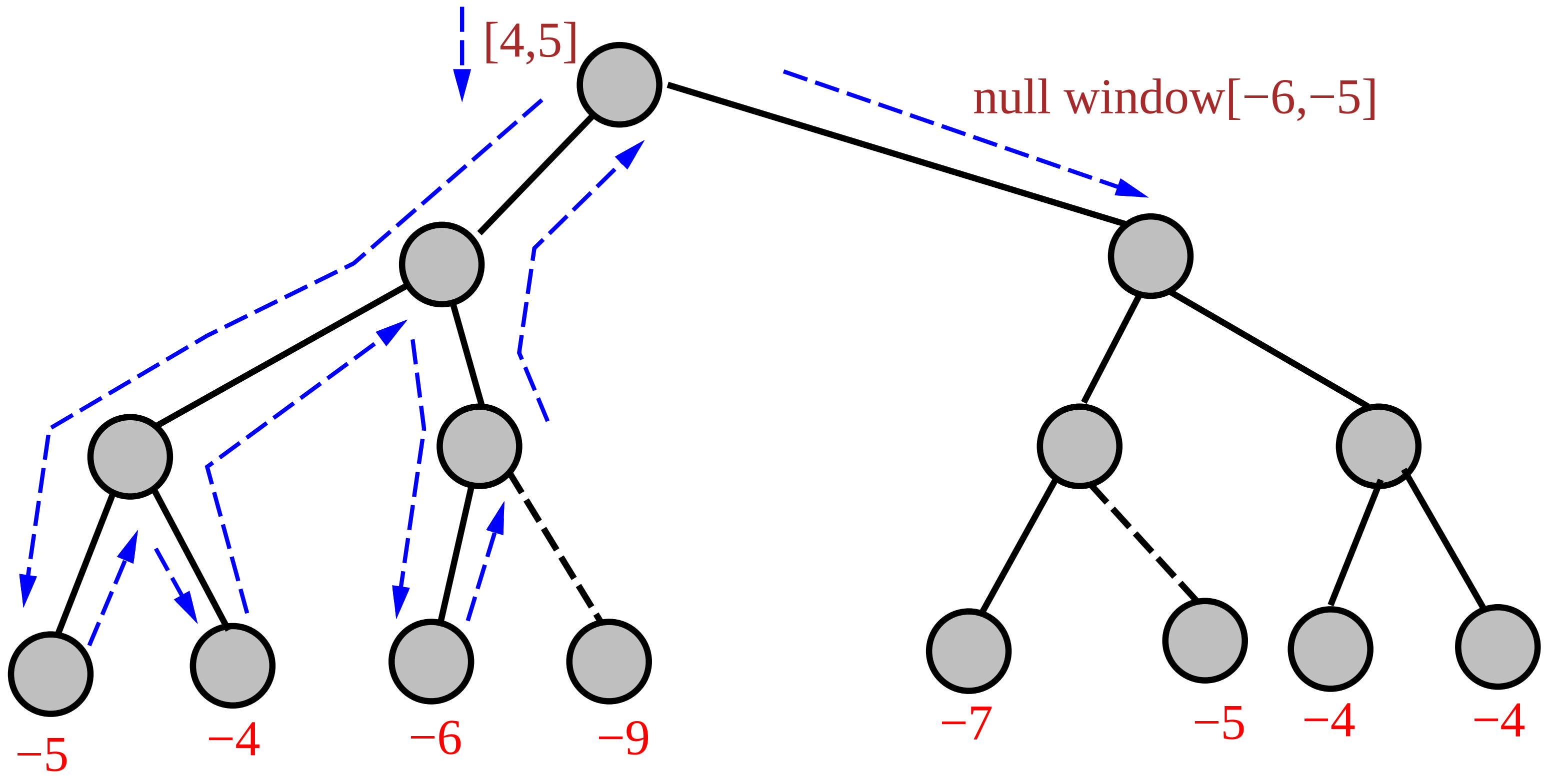
Search behaviors (2/3)

- **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 best-first 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 tree 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.

References and further readings

* A. Reinefeld. An improvement of the scout tree search algorithm. *ICCA Journal*, 6(4):4–14, 1983.
* J. Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14(2):113–138, 1980.