Transposition Table, History Heuristic, and other Search Enhancements

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

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

Abstract

- Introduce heuristics for improving the efficiency of alpha-beta based searching algorithms.
    - Re-using information: Transposition table.
    - Adaptive searching window size.
    - Better move ordering.
    - Dynamically adjust searching depth.
- Study the effect of combining multiple heuristics.
    - Each enhancement should not be taken in isolation.
    - Try to find the combination that provides the greatest reduction in tree size.
- Be careful on artificial game trees.
- Be careful on the type of game trees that you do experiments on.
    - Depth, width and leaf-node evaluation time.
    - A heuristic that is good on the current experiment setup may not be good some years in the future because of the the game tree can be evaluated much deeper in the the same time using faster CPU's.

Enhancements and heuristics

- **Always used enhancements**
  - Iterative deepening
  - Alpha-beta, NegaScout or Monte-Carlo search algorithm
  - Transposition table
- **Frequently used heuristics**
  - Knowledge heuristic: using domain knowledge to enhance evaluating functions or move ordering.
  - Aspiration search
  - Refutation tables
  - Killer heuristic
  - History heuristic
- **Some techniques about aggressive forward pruning**
  - Null move pruning
  - Late move reduction
- **Search depth extension**
  - Conditional depth extension: to check doubtful positions.
  - Quiescent search: to check forceful variations.

Transposition tables

- **We are searching a game graph, not a game tree.**
  - **Interior nodes of game trees are not necessarily distinct.**
  - **It may be possible to reach the same position by more than one path.**
- **How to use information in the transposition table?**
  - **Suppose p is searched again with the depth limit d'.**
  - **If $d \geq d'$, then no need to search anymore.**
    - ▷ ***Just retrieve the result from the table.***
  - **If $d < d'$, then use the best move stored as the starting point for searching.**
- **Need to be able to locate p in a large table efficiently.**

Transposition tables: contents

- **What are recorded in an entry of a transposition table?**
  - **The position p.**
    - ▷ ***Note: the position describes who the next player is.***
  - **Searching depth d.**
  - **Best value in this subtree.**
    - ▷ ***Can be an exact value when the best value is found.***
    - ▷ ***Maybe a value that causes a cutoff.***
      $\rightarrow$ ***In a MAX node, it says at least v when a $beta$ cut off occurred.***
      $\rightarrow$ ***In a MIN node, it says at most v when an $alpha$ cut off occurred.***
  - **Best move, or the move caused a cut off, for this position.**

Transposition tables: updating rules

- **It is usually the case that at most one entry of information for a position is kept in the transposition table.**
- **When it is decided that we need to record information about a position p into the transposition table, we may need to consider the followings.**
  - **If p is not currently recorded, then just store it into the transposition table.**
    - ▷ ***Be aware of the fact that p's information may be stored in a place that previously occupied by another position q such that $p \neq q$.***
    - ▷ ***In most cases, we simply overwrite.***
  - **If p is currently recorded in the transposition table, then we need a good updating rule.**
    - ▷ ***Some programs simply overwrite with the latest information.***
    - ▷ ***Some programs compares the depth, and use the one a deeper searching depth.***
    - ▷ ***When the searching depths are the same, we normally favor one with the latest information.***

NegaScout with memory

- **Algorithm** $F4.1'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **check whether a value of** p **has been recorded in the transposition table**
  - **if yes, then**

    found an old HASH entry

    do procedures for hash hit, compare depth,

    $\cdots$
  - **determine the successor positions** $p_1, \ldots, p_d$
  - $\cdots$

    begin

    ▷ $m := -\infty$ ***or*** m' ***if exists// m is the current best lower bound; fail soft***

    ▷ $\cdots$

    if $m \geq beta$ ***then update this value as a lower bound into the transposition table; return*** m ***// beta cut off***

    ▷ ***for*** $i := 2$ ***to*** d ***do***

    ▷ $\cdots$

    ▷ ***14: if*** $m \geq beta$ ***then update this value as a lower bound into the transposition table; return*** m ***// beta cut off***

    end
  - **update this value as an exact value into the transposition table; return** m

Zobrist's hash function

- **Find a hash function $hash(p)$ so that with a very high probability that two distinct positions will be mapped into distinct locations in the table.**
- **Using XOR to achieve fast computation:**
  - **associativity: x XOR (y XOR z) = (x XOR y) XOR z**
  - **commutativity: x XOR y = y XOR x**
  - **x XOR x = 0**
    - ▷ *x XOR 0 $= x$*
    - ▷ *(x XOR y) XOR $y = x$ XOR (y XOR y) $= x$ XOR 0 $= x$*
  - **x XOR y is random if x and y are also random**

Hash function

- **Assume there are k different pieces and each piece can be placed into r different locations.**
  - **Obtain $k \cdot r$ random numbers in the form of $s[piece][location]$**
  - **$hash(p) = s[p_1][l_1]$ XOR $\cdots$ XOR $x[p_x][l_x]$ where p_i is the ith piece and l_i is the location of p_i.**
- **This value can be computed incrementally.**
  - **Assume the original hash value is h.**
  - **A piece p_{x+1} is placed at location l_{x+1}, then**
    - ▷ ***new hash value* $= h$ *XOR* $s[p_{x+1}][l_{x+1}]$.**
  - **A piece p_y is removed from location l_y, then**
    - ▷ ***new hash value* $= h$ *XOR* $s[p_y][l_y]$.**
  - **A piece p_y is moved from location l_y to location l'_y then**
    - ▷ ***new hash value* $= h$ *XOR* $s[p_y][l_y]$ *XOR* $s[p_y][l'_y]$.**
  - **A piece p_y is moved from location l_y to location l'_y and capture the piece p'_y at l'_y then**
    - ▷ ***new hash value* $= h$ *XOR* $s[p_y][l_y]$ *XOR* $s[p_y][l'_y]$ *XOR* $s[p'_y][l'_y]$.**
- **It is also easy to undo a move.**

Clustering of errors

- **Though the hash codes are uniformly distributed, the idiosyncrasies of a particular problem may produce an unusual number of clashes.**
  - **if $hash(p^*) = hash(p^+)$, then**
    - ▷ ***adding the same pieces at the same locations to positions p^* and p^+ produce the same clashes;***
    - ▷ ***removing the same pieces at the same locations from positions p^* and p^+ produce the same clashes.***

Practical issues (1/2)

- **Normally, design a hash table of 2^n entries, but with key length $n+m$ bits.**
  - **That is, each $s[piece][location]$ is a random value of $n+m$ bits.**
  - **Hash index = $hash(p) \mod 2^n$.**
  - **Store the hash key to compare when there is a hash hit.**
- **How to store a hash entry:**
  - **Store it when the entry is empty.**
  - **Replace the old entry if the current result comes from a deeper subtree.**
- **How to match an entry:**
  - **First compute $i = hash(p) \mod 2^n$**
  - **Compare $hash(p)$ with the stored key in the ith entry.**
  - **Since the error rate is very small, there is no need to store the exact position and then make a comparison.**

Practical issues (2/2)

- **Errors:**
  - **Assume this hash function is uniformly distributed.**
  - **The chance of error for hash clash is $\frac{1}{2^{n+m}}$.**
  - **Assume during searching, 2^w nodes are visited.**
  - **The chance of no clash in these 2^w visits is**

$$P = (1 - \frac{1}{2^{n+m}})^{2^w} \simeq (\frac{1}{e})^{2^{-(n+m-w)}}.$$

    - ***When $n+m-w$ is 5, $P \simeq 0.96924$.***
    - ***When $n+m-w$ is 10, $P \simeq 0.99901$.***
    - ***When $n+m-w$ is 20, $P \simeq 0.99999904632613834096$.***
    - ***When $n+m-w$ is 32, $P \simeq 0.99999999976716935638$.***
  - **Currently (2012):**
    - $n + m = 64$
    - $n \leq 32$
    - $w \leq 32$

Intuitions for possible enhancements

- **The size of the search tree built by a depth-first alpha-beta search largely depends on the order in which branches are considered at interior nodes.**
  - **It looks good if one can search the best possible subtree first in each interior node.**
  - **A better move ordering normally means a better way to prune a tree using alpha-beta search.**
- **Enhancements to the alpha-beta search have been proposed based on one or more of the following principles:**
  - **knowledge;**
  - **window size;**
  - **better move ordering;**
  - **forward pruning;**
  - **dynamic search extension;**
  - $\cdots$

Knowledge heuristic

- **Use game domain specified knowledge to obtain good**
  - **move ordering;**
  - **evaluating function.**
- **Example from chess like games for a good move ordering**
  - **Moves to avoid being checking or captured**
  - **Checking moves**
  - **Capturing moves**
    - ▷ *Favor capturing pieces of important*
    - ▷ *favor capturing pieces using pieces as little as possible*
  - **Moving of pieces with large material values**

Aspiration search

- **The normal alpha-beta search usually starts with a $(-\infty, \infty)$ search window.**
- **If some idea of the range of the search will fall is available, then tighter bounds can be placed on the initial window.**
  - **The tighter the bound, the faster the search.**
  - **Some possible guesses:**
    - ▷ ***During iterative deepening, assume the previous best value is x, then use $(x - threshold, x + threshold)$ as the initial window size where $threshold$ is a small value.***
- **If the value falls within the window then the original window is adequate.**
- **Otherwise, one must re-search with a wider window depending on whether it fails high or fails low.**
- **Reported to be at least 15% faster than the original alpha-beta search.**

Aspiration search — Algorithm

- **Iterative deepening with aspiration search.**
  - p **is the current board**
  - $limit$ **is the limit of searching depth, assume** $limit > 3$
  - $threshold$ **is the initial window size**
- **Algorithm IDAS(**p,$limit$,$threshold$**)**
  - $best := F4(p,-\infty,+\infty,3)$ **// initial value**
  - $current\_depth\_limit := 4$
  - **while** $current\_depth\_limit <= limit$ **do**
    - ▷ $m := F4(p,best - threshold,best + threshold,current\_depth\_limit)$
    - ▷ **if** $m \leq best - threshold$ **then** **// failed low**
      $m := F4(p,-\infty,m,current\_depth\_limit)$
    - ▷ **else if** $m \geq best + threshold$ **then** **// failed high**
      $m := F4(p,m,\infty,current\_depth\_limit)$
    - ▷ **else** $best := m$ **// found**
    - ▷ **if the time is used up then return** $best$
    - ▷ $current\_depth\_limit := current\_depth\_limit + 1$
  - **return** $best$

IDAS: comments

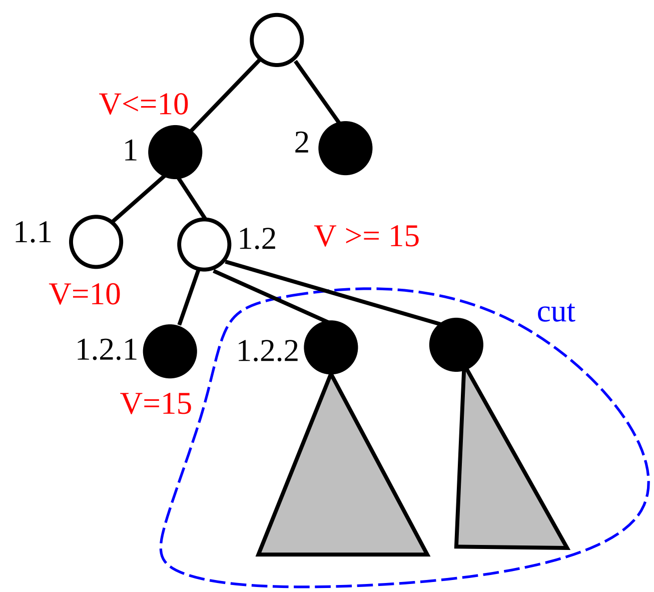
- **May want to try multiple window sizes.**
  - **For example: try** $[best - t_1, best + t_1]$ **first.**
  - **If failed low, try** $[best - t_1 - t_2, best - t_1]$**.**
  - **If failed high, try** $[best + t_1, best + t_1 + t_2]$**.**
  - $\cdots$
  - **Need to decide various** t_i **via experiments.**
- **Aspiration search is better to be used together with a transposition table so that information from the previous search can be reused later.**
- **Ideas here may also be helpful in designing better progressive pruning policy for Monte-Carlo-based search.**

Better move ordering

- **Intuition: the game evolves continuously.**
  - **What are considered good or bad in previous plys cannot be off too much in this ply.**
  - **If iterative deepening or aspiration search is used, then what are considered good or bad in the previous iteration cannot be off too much now.**
- **Techniques:**
  - **Refutation table.**
  - **Killer heuristic.**
  - **History heuristic.**

What moves are good?

- In alpha-beta search, a sufficient, or good, move at an interior node is defined as
  - one causes a cutoff, or
    - ▷ *Remark: this move is potentially good for its parent, though a cutoff happens may depends on the value of its older siblings.*
  - if no cutoff occurs, the one yielding the best minimax score, or
  - the one that is a sibling of the chosen yielding the best minimax score and has the same best score.



PV path

- **For each iteration, the search yields a path for each move from the root to a leaf node that results in either the correct minimax value or an upper bound on its value.**
  - **This path is often called principle variation (PV) or principle continuation.**
- **Q: What moves are considered good in the context of Monte-Carlo simulation?**
  - **There is currently no equivalent ideas for iterative deepening.**
    - ▷ *Need other techniques for better timing control.*
  - **Can information in the previous-ply Monte-Carlo search be used in searching this ply?**

Refutation tables

- **Assume using iterative deepening with an increasing $current\_depth\_limit$ being bounded by $limit$.**
  - **Store the current best principle variation at $P_{current\_depth\_limit,i}$ for each depth i at the current depth limit $current\_depth\_limit$.**
- **The PV path from the $current\_depth\_limit = d-1$ ply search can be used as the basis for the search to $current\_depth\_limit = d$ ply at the same depth.**
- **Searching the previous iteration's path or refutation for a move as the initial path examined for the current iteration will prove sufficient to refute the move one ply deeper.**
  - **When searching a new node at depth i for the current depth limit $current\_depth\_limit$,**
    - ▷ ***try the move made by this player at $P_{current\_depth\_limit-1,i}$ first;***
    - ▷ ***then try moves made by this player at $P_{current\_depth\_limit-2,i}$;***
    - ▷ $\cdots$

How to store the PV path

- **Algorithm** $F4.2'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_d$
  - **if** $d = 0$ **// a terminal node**
    $\ldots$
  - **then return** $f(p)$ **else**
    begin
    - $m := -\infty$ ***// m is the current best lower bound; fail soft***
      $m := \max\{m, G4.2'(p_1, alpha, beta, depth - 1)\}$ ***// the first branch***
      if $m \geq beta$ ***then*** $PV[current\_depth\_limit, depth] := p_1$***; return***$(m)$ ***// beta cut off***
    - ***for*** $i := 2$ ***to*** d ***do***
    - ***9:*** $t := G4.2'(p_i, m, m + 1, depth - 1)$ ***// null window search***
    - ***10:*** ***if*** $t > m$ ***then*** ***// failed-high***
      11: ***if*** $(depth < 3$ ***or*** $t \geq beta)$
      12: ***then*** $PV[current\_depth\_limit, depth] := p_i$***;*** $m := t$
      13: ***else*** $m := G4.2'(p_i, t, beta, depth - 1)$ ***// re-search***
    - ***14:*** ***if*** $m \geq beta$ ***then*** $PV[current\_depth\_limit, depth] := p_i$***; return***$(m)$ ***// beta cut off***

    end
  - **return** m **//PV entry is recorded in line 12**

How to use the PV

- **Use the PV information to do a better move ordering**
  - **Assume the current depth limit from iteration deepening is** $current\_depth\_limit$**.**
- **Algorithm** $F4.2.1'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_d$
  - **// get a better move ordering by using information stored in PV**
  - $k = 0$**;**
  - **for** $i = current\_depth\_limit - 1$ **downto** 1 **do**
    if $PV[i, depth] = p_x$ **and** $d \geq x > k$**, then**
    - ▷ ***swap*** p_x ***and*** p_k***; // make this move as the*** k***th move t be considered***
    - ▷ $k := k + 1$
  - $\cdots$

Killer heuristic

- **A compact refutation table.**
- **Storing at each depth of search the moves which seem to be causing the most cutoffs, i.e., so called killers.**
  - **Currently, store two most recent cutoffs at this depth.**
- **The next time the same depth in the tree is reached, the killer move is retrieved and used, if valid in the current position.**
- **Comment:**
  - **It is plausible to record more than one killers. However, the time to maintain them may be too much.**

History heuristic

- **Intuition:**
  - **A move M may be shown to be best in one position.**
  - **Later on in the search tree a similar position may occur, perhaps only differing in the location of one piece.**
    - *A position p and a position p' obtained from p by making one or two moves are likely to share important features.*
  - **Minor difference between p and p' may not change the position enough to alter move M from still being best.**
- **Recall: In alpha-beta search, a sufficient, or good, move at an interior node is defined as**
  - **one causes a cutoff, or**
  - **if no cutoff occurs, the one yielding the best minimax score, or**
  - **a move that is "equivalent" to the best move.**

Implementation (1/2)

- **Keep track of the history on what move are good.**
  - **Assume the board has q different locations.**
  - **Assume each time only a piece can be moved.**
  - **There are only q^2 possible moves.**
  - **Including more context information, e.g., the piece moving, did not significantly increase performance.**
    - ▷ ***If you carry the idea of including context to the extreme, the result is a transposition table.***
- **The history table.**
  - **In each entry, use a counter to record the weight or chance that this entry becomes a good move during searching.**
  - **Be careful for a possible counter overflow.**

Implementation (2/2)

- Each time when a move is good, increases its counter by a certain weight.
  - During move generation, pick one with the largest counter value.
    - ▷ *Need to access the history table and then sort the weights in the move queue.*
  - The deeper the subtree searched, the more reliable the minimax value except in pathological trees, rarely seen in practice.
  - The deeper the search tree, and hence larger, the greater the differences between two arbitrary positions in the tree and less they may have in common.
  - By experiment: let weight $= 2^{depth}$, where $depth$ is the depth of the subtree searched.
    - ▷ *Several other weights, such as 1 and $depth$, were tried and found to be experimentally inferior to 2^{depth}.*
- Killer heuristic is a special case of the history heuristic.
  - Killer heuristic only keeps track of one or two successful moves per depth of search.
  - History heuristic maintains good moves for all depths.
- History heuristic is very dynamic.

History heuristic: counter updating

- **Algorithm** $F4.3'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_d$
  - **if** $d = 0$ **// a terminal node**
    $\ldots$
  - **then return** $f(p)$ **else**
    begin
      - ▷ $m := -\infty$ ***// m is the current best lower bound; fail soft***
        $m := \max\{m, G4.3'(p_1, alpha, beta, depth - 1)\}$ ***// the first branch***
        if $m \geq beta$ ***then*** $HT[p_1] = HT[p_1] + weight$***; return(***$m$***)*** ***// beta cut off***
      - ▷ ***for*** $i := 2$ ***to*** d ***do***
      - ▷ ***9:*** $t := G4.3'(p_i, m, m + 1, depth - 1)$ ***// null window search***
      - ▷ ***10:*** ***if*** $t > m$ ***then*** ***// failed-high***
        11: ***if (***$depth < 3$ ***or*** $t \geq beta$***)***
        12: ***then*** $HT[p_i] = HT[p_i] + weight$***;*** $m := t$
        13: ***else*** $m := G4.3'(p_i, t, beta, depth - 1)$ ***// re-search***
      - ▷ ***14:*** ***if*** $m \geq beta$ ***then*** $HT[p_i] = HT[p_i] + weight$***; return(***$m$***)*** ***// beta cut off***

    end
  - **return** m

History heuristic: usage of the counter

- **Algorithm** $F4.3.1'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_d$
  - **order the moves in** $p_1, \ldots, p_d$ **according to the weights in HT[]**
  - $\cdots$

Comments: better move ordering

- **Need to take care of the case for the chance of a counter overflow.**
  - **Need to perform counter aging periodically.**
  - **That is, discount the value of the current counter as the game goes.**
  - **This also makes sure that the counter value reflects the "current" situation better, and to make sure it won't be overflowed.**
- **Ideas here may also be helpful in designing better node expansion policy for Monte-Carlo-based search.**

Experiments: Setup

- **Try out all possible combinations of heuristics.**
  - **6 parameters with 64 different combinations.**
    - ***Transposition table***
    - ***Knowledge heuristic***
    - ***Aspiration search***
    - ***Refutation tables***
    - ***Killer heuristic***
    - ***History heuristic***
- **Searching depth from 2 to 5 for all combinations.**
  - **Applying searching upto the depth of 6 to 8 when a combination showed significant reductions in search depth of 5.**
- **A total of 2000 VAX11/780 equivalent hours are spent to perform the experiments.**

Experiments: Results

- **Using a single parameter:**
  - *History heuristic performs well, but its efficiency appears to drop after depth 7.*
  - *Knowledge heuristic adds an additional 5% time, but performs about the same with the history heuristic.*
  - *The effectiveness of transposition tables increases with search depth.*
  - *Refutation tables provide constant performance, regardless of depth, and appear to be worse than transposition tables.*
  - *Aspiration and minimal window search provide small benefits.*
- **Using two parameters**
  - *Transposition tables plus history heuristic provide the best combination.*
- **Combining three or more heuristics do not provide extra benefits.**

Comments

- **Combining two best heuristics may not give you the best.**
- **Need to weight the amount of time spent in realizing a heuristic and the benefits it can bring.**
- **Need to be very careful in setting up the experiments.**

Dynamically adjusting searching depth

- **Aggressive forward pruning: do not search branches that seem to be excellent or hopeless too deep.**
  - **Null move pruning**
  - **Late move reduction**
- **Search depth extension: search a branch deeper if a side is in "danger".**
  - **Conditional depth extension: to check doubtful positions.**
  - **Quiescent search: to check forceful variations.**

Null move pruning

- **In general, if you forfeit the right to move and can still maintain the current advantage in a small number of plys, then it is usually true you can maintain the advantage in a larger number of plys.**
- **Algorithm:**
  - **It's your turn to move; the searching depth for this node is d.**
  - **During searching, an upper bound of $beta$ is obtained.**
  - **Make a null move, i.e., assume you do not move and let the opponent move again.**
    - ▷ ***Perform an alpha-beta search with a reduced depth $d - R$, where R is a constant decided by experiments.***
    - ▷ ***If the returned value v is at least $beta$, then apply a $beta$ cutoff and return v as the value.***
    - ▷ ***If the returned value v does not produce a cutoff, then do the normal alpha-beta search.***

Null move pruning: analysis

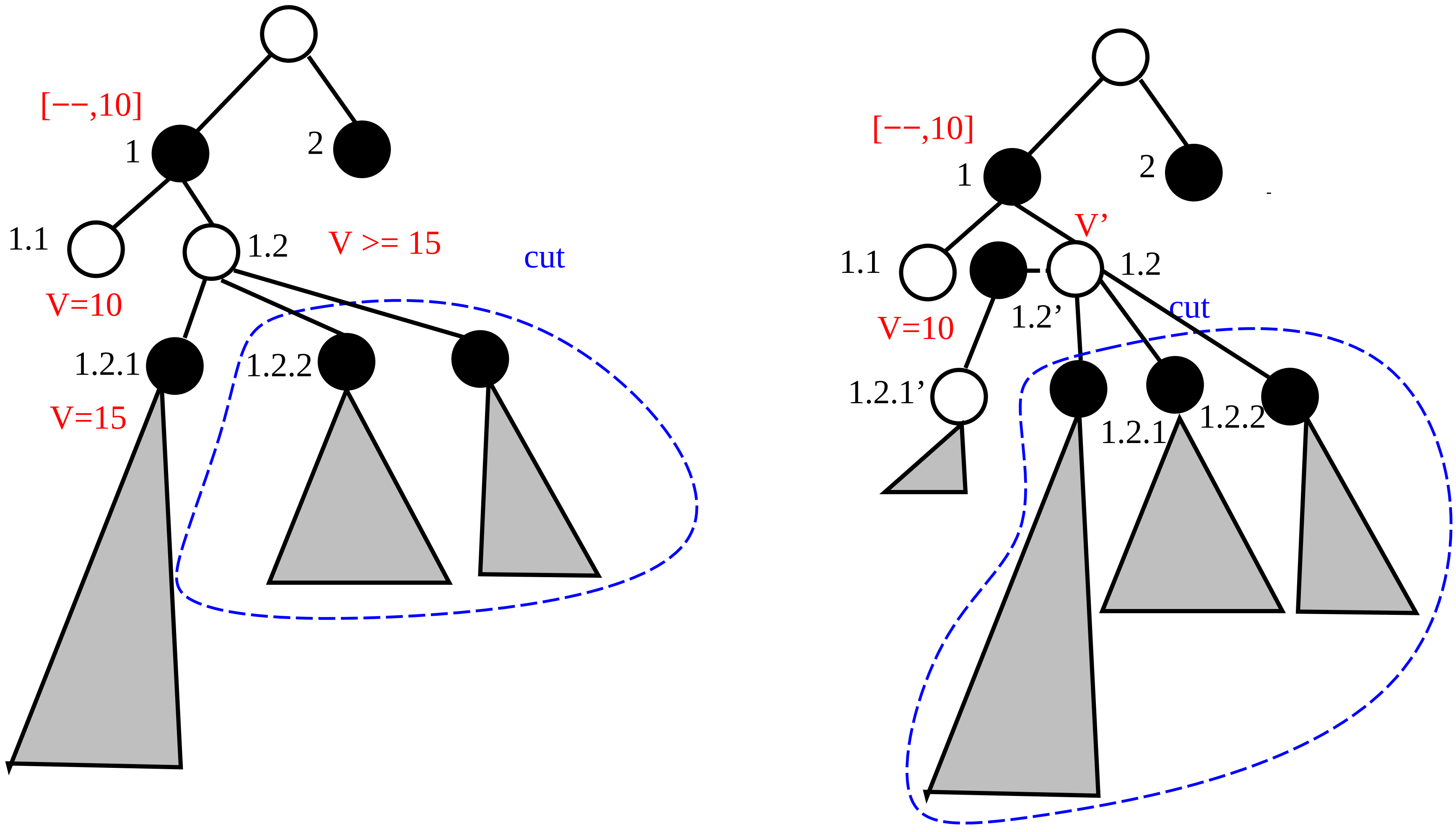
- **Assumptions:**
  - **The depth reduced, R, is usually 2 or 3.**
  - **The disadvantage of doing a null move can offset the errors produced from doing a shallow search.**
  - **Usually do not apply null move when**
    - ▷ ***your king is in danger, e.g., in check;***
    - ▷ ***when the number of pieces are small;***
    - ▷ ***when there is chance of Zugzwang;***
    - ▷ ***when you are already in null move search;***
    - ▷ ***when the number of remaining depth is small.***
- **Performance is usually good with about 10 to 30 % improvement, but needs to set the parameters right in order not to prune moves that need deeper search to find out their true values.**

Null move pruning — Algorithm

- **Algorithm** $F4.4'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_d$
  - **if** $d = 0$ **// a terminal node**
    $\cdots$
  - **then return** $f(p)$ **else**
    begin
    - ▷ ***// null move pruning***
    - ▷ $null\_score := F4.4'(p', beta, beta+1, depth - R - 1)$***, where*** p' ***is the position obtained by switching the player in*** p***, and*** R ***is usually*** 2
    - ▷ ***if*** $null\_score \geq beta$ ***return*** $null\_score$ ***// null move pruning***
    - ▷ ***// normal NegaScout search***
    - ▷ $m := -\infty$ ***//*** m ***is the current best lower bound; fail soft***
      $m := \max\{m, G4.4'(p_1, alpha, beta, depth - 1)\}$ ***// the first branch***
      if $m \geq beta$ ***then return(***m***) // beta cut off***
    - ▷ ***for*** $i := 2$ ***to*** d ***do***
    - ▷ $\cdots$

    end
  - **return** m

Null move pruning — Example



Late move reduction (LMR)

- **Assumption:**
  - **The move ordering is relatively good.**
- **Observation:**
  - **During search, the best move rarely comes from moves that are ordered very late in the move queue.**
- **How to make use of the observation:**
  - **If the first few, say 3 or 4, moves considered do not produce a value that is better the current best value, then**
    - ▷ ***consider the rest of the moves with a reduced depth.***
  - **If some moves considered with a reduced depth returns a value that is better than the current best, then**
    - ▷ ***re-search the game tree at a full depth.***

LMR: analysis

- **Performance:**
  - **Reduce the effective branching factor to about 2.**
- **Usually do not apply this scheme when**
  - **your king is in danger, e.g., in check;**
  - **you or the opponent is making an attack;**
  - **the remaining searching depth is too small, say less than 3;**
  - **it is a node in the PV path.**

LMR — Algorithm

- **Algorithm** $F4.5'$(**position** p, **value** $alpha$, **value** $beta$, **integer** $depth$)
  - **determine the successor positions** $p_1, \ldots, p_d$
  - **if** $d = 0$ **// a terminal node**

    $\cdots$
  - **then return** $f(p)$ **else**

    begin

    ▷ $m := -\infty$ ***// m is the current best lower bound; fail soft***

    $\cdots$

    ▷ ***for*** $i := 2$ ***to*** d ***do***

    ▷   ***if*** $i \geq 4$ ***and*** $depth > 3$ ***and*** p_i ***is not dangerous, then***

            $depth' := depth - 3$ ***// late moves are searched with reduced depth***

            else $depth' := depth$

    ▷ $t := G4.5'(p_i, m, m+1, depth' - 1)$ ***// null window search***

    ▷   ***if*** $t > m$ ***then // failed-high***

            if ($depth' < 3$ ***or*** $t \geq beta$)

            then $m := t$

            else $m := G4.5'(p_i, t, beta, depth - 1)$ ***// re-search***

    ▷   ***if*** $m \geq beta$ ***then return***(m) ***// beta cut off***

    end
  - **return** m

Dynamic search extension

- **Search extensions**
  - **Some nodes need to be explored deeper than the others to avoid the horizontal effect.**
    - *Horizontal effect is the situation that a stable value cannot be found because a fixed searching depth is set.*
  - **Needs to be very careful to avoid non-terminating search.**
  - **Examples of conditions that need to extend the search depth.**
    - *Extremely low mobility.*
    - *In-check.*
    - *Last move is capturing.*
    - *The current best score is much lower than the value of your last ply.*
- **Quiescent search: to check forceful variations.**
  - **Invoke your search engine, e.g., alpha-beta search, to only consider moves that are in-check or capturing.**
    - *May also consider checking moves.*
    - *May also consider allowing upto a fixed number, say 1, of non-capturing moves in a search path.*

Dynamic depth extension — Algorithm

- **Algorithm** $F4.6'$(**position** p, **value** $alpha$, **value** $beta$, **integer** $depth$)
  - **determine the successor positions** $p_1, \ldots, p_d$
  - **if** $d = 0$ **// a terminal node**
    $\cdots$
  - **then return** $f(p)$ **else**
    begin
    - ▷ ***if*** p_1 ***is capturing, ..., then*** $depth' := depth + 1$ ***else*** $depth' := depth$
    - ▷ $m := -\infty$ ***//*** m ***is the current best lower bound; fail soft***
      $m := \max\{m, G4.6'(p_1, alpha, beta, depth' - 1)\}$ ***// the first branch***
      if $m \geq beta$ ***then return***(m) ***// beta cut off***
    - ▷ ***for*** $i := 2$ ***to*** d ***do***
    - ▷ ***if*** p_i ***is capturing, ..., then*** $depth' := depth + 1$ ***else*** $depth' := depth$
    - ▷ ***9:*** $t := G4.6'(p_i, m, m + 1, depth' - 1)$ ***// null window search***
    - ▷ ***10:*** ***if*** $t > m$ ***then // failed-high***
      11: ***if*** ($depth < 3$ ***or*** $t \geq beta$)
      12: ***then*** $m := t$
      13: ***else*** $m := G4.6'(p_i, t, beta, depth' - 1)$ ***// re-search***
    - ▷ ***14:*** ***if*** $m \geq beta$ ***then return***(m) ***// beta cut off***

    end
  - **return** m

Comments

- **There are very more such search enhancements.**
  - **Mainly designed for alpha-beta based searching.**
  - **It is worthy while to think whether techniques designed for one search method can be adopted to be used in the other search method.**
- **Finding the right coefficients, or parameters, for these techniques can only now be done by experiments.**
  - **Is there any general theory for finding these coefficients faster?**
  - **The coefficients need to be re-tuned once the searching behaviors change.**
    - ▷ ***Changing evaluating functions.***
    - ▷ ***faster hardware so that the searching depth is increased.***
    - ▷ $\cdots$
- **Need to tradeoff between the time spent and the technique used.**

References and further readings

* J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.
* A. L. Zobrist. A new hashing method with applications for game playing. Technical Report 88, Department of Computer Science, University of Wisconsin, Madison, USA, 1970. Also in *ICCA journal*, vol. 13, No. 2, pp. 69–73, 1990.
* Selim G. Akl and Monroe M. Newborn. The principal continuation and the killer heuristic. In *ACM '77: Proceedings of the 1977 annual conference*, pages 466–473, New York, NY, USA, 1977. ACM Press.
- S.C. Hsu. Searching Techniques of Computer Game Playing. *Bulletin of the College of Engineering, National Taiwan University*, 51:17–31, 1991.