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 to improve the efficiency of alpha-beta based searching algorithms.**
  - **Re-using information: Transposition table.**
    - ***Can also be used in MCTS based searching.***
  - **Adaptive searching window size.**
  - **Better move ordering.**
  - **Dynamically adjusting the searching depth.**
    - ***Decreasing***
    - ***Increasing***
- **Study the effect of combining multiple heuristics.**
  - **Each enhancement should not be taken in isolation.**
  - **Try to find a combination that provides the greatest reduction.**
- **Be careful on the game trees to study.**
  - **Artificial game trees.**
  - **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 the the same game tree can be evaluated much deeper under the same timing by using faster hardware, e.g., CPU.**

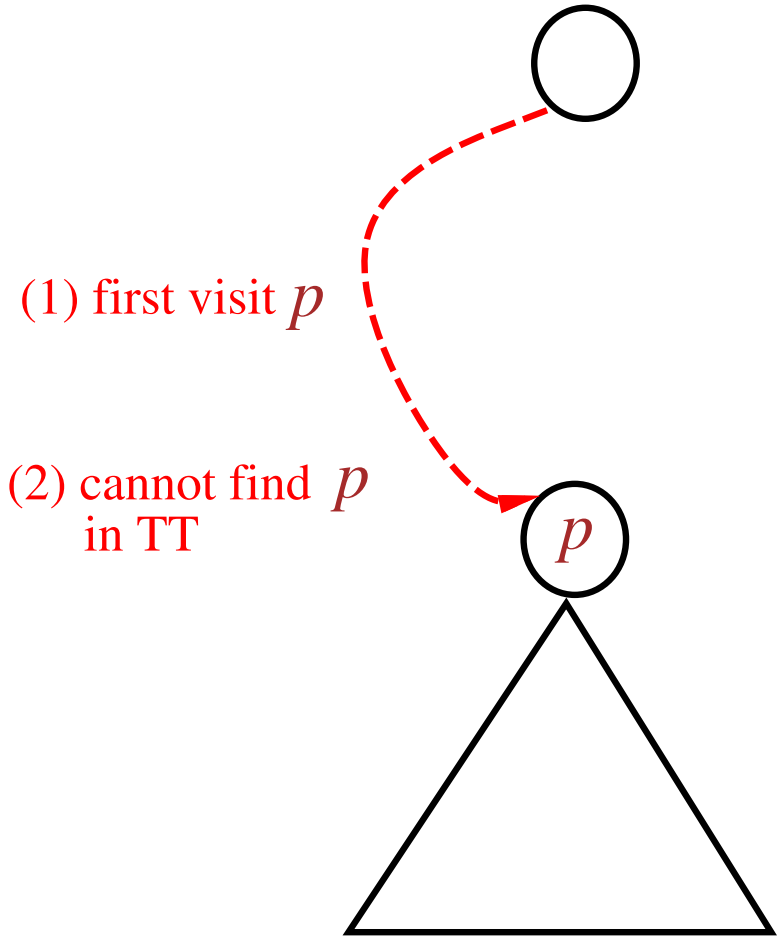
Enhancements and heuristics

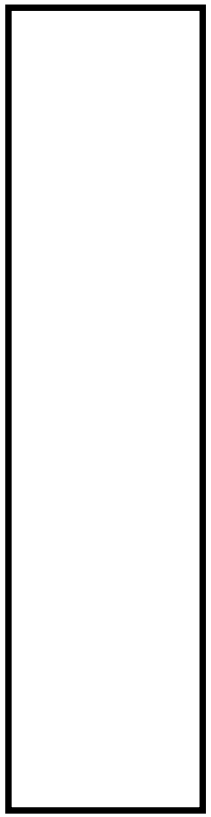
- **Always used enhancements**
  - Alpha-beta, NegaScout or Monte-Carlo search based algorithms
  - Iterative deepening
  - Transposition table
  - Knowledge heuristic: using domain knowledge to enhance the design of evaluation functions or to make the move ordering better.
- **Frequently used heuristics**
  - 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 (TT)

- **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.**
    - ▷ ***Save information obtained from searching into a transposition table.***
    - ▷ ***When being to search a position, first check whether it has been searched before or not.***
    - ▷ ***If yes, reuse the information wisely.***
- **Several search engines, such as NegaScout, need to re-search the same node more than once.**
  - **Anything with interactive deepening also do.**

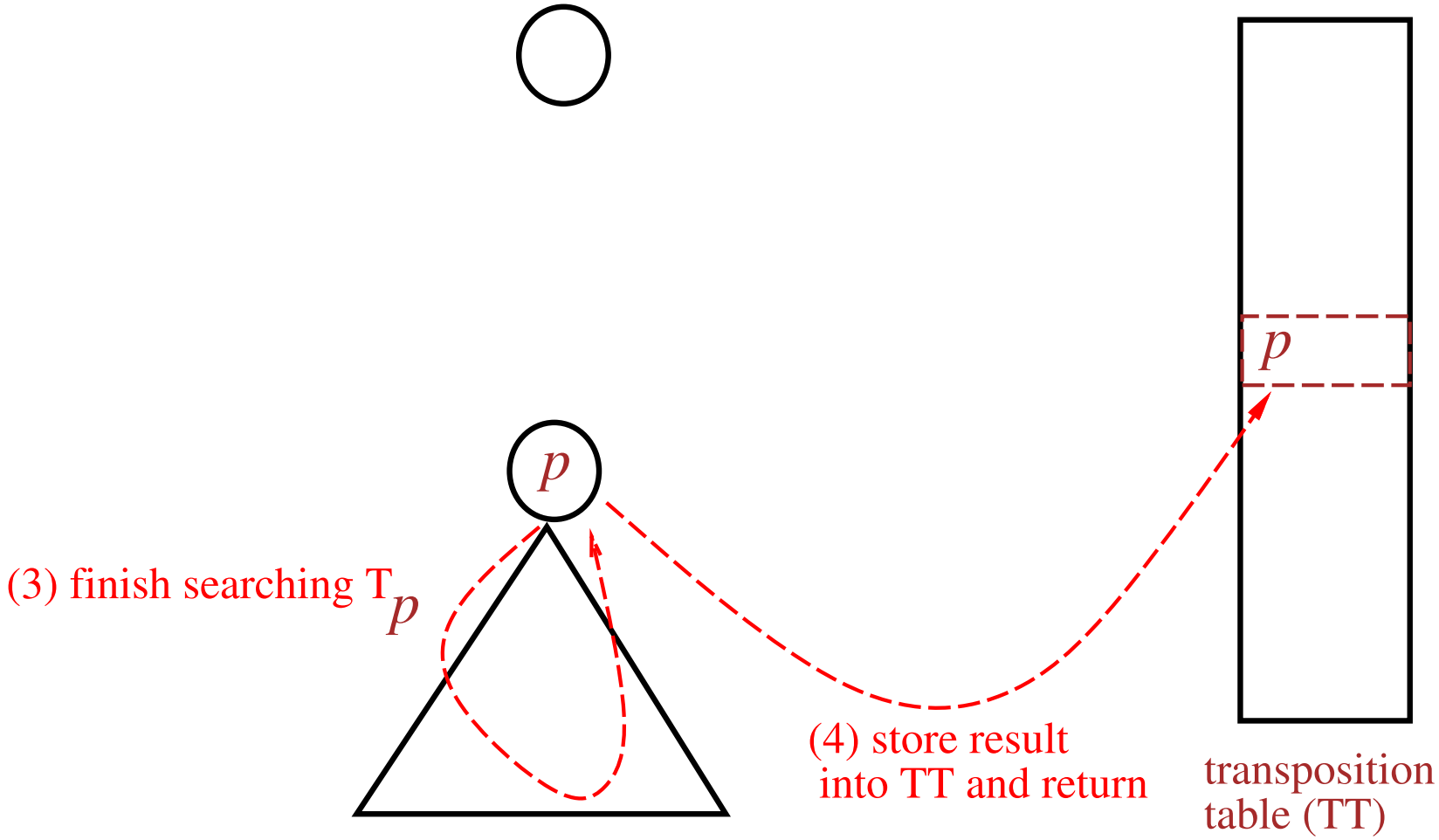
Usage of TT: Illustration



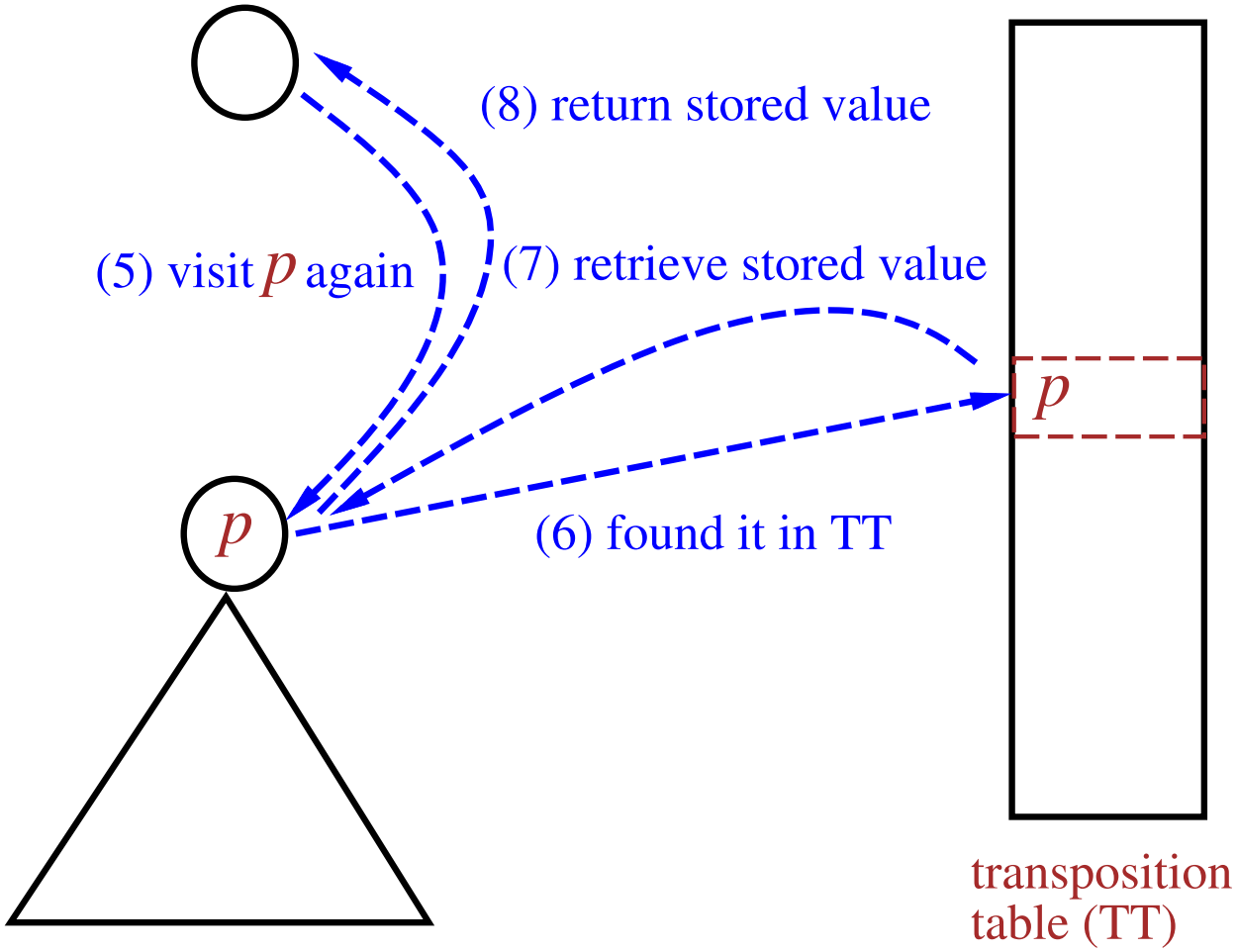


transposition table (TT)

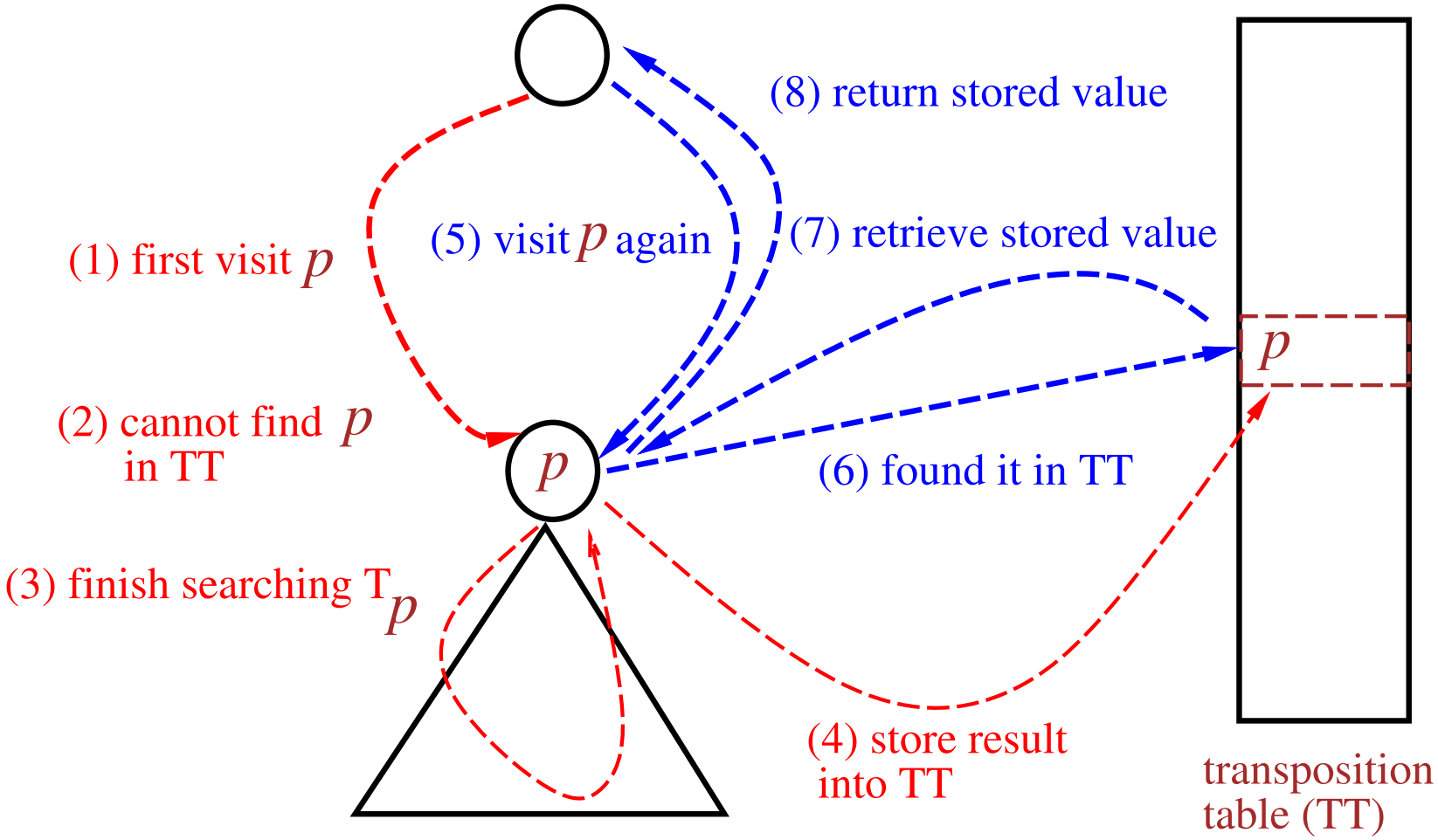
Usage of TT: Illustration



Usage of TT: Illustration



Usage of TT: Illustration



Transposition tables: contents

- **What are recorded in an entry of a transposition table?**
  - **The position p.**
    - ▷ ***Note: the position also tells who the next player is.***
  - **Searched depth d.**
  - **Best value in this subtree of depth d.**
    - ▷ ***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 and $p \neq q$, i.e., **hash clash**.*
    - ▷ *In most cases of hash crashes, 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 compare the depth, and use the one with a deeper searching depth when the value is exact.*
      $\longrightarrow$ *When the searching depths are the same, one normally favors one with the latest information.*
    - ▷ *In the case of bounds being stored,*
      $\longrightarrow$ *When the searching depths are the same, one normally use better bounds*

Alpha-beta (Mini-Max) with memory

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

    begin
    - $m := \max\{m, G4.1'(p_1, alpha, beta, depth - 1)\}$ // ***the first branch***
    - ***if*** $m \geq beta$ ***then return***(m) // ***beta cut off***
    - ***for*** $i := 2$ ***to*** b ***do***
    - $\cdots$ ***recursive call***
    - ***14: if*** $m \geq beta$ ***then*** { ***record into the TT as a lower bound*** m***; return*** m } // ***beta cut off***

    end
  - **if** $m > alpha$ **then record into the TT as an exact value** m
    else record into the TT as an upper bound m**;**
  - **return** m

TT hit: data structure

- **Be careful to check whether the position is exactly the same.**
  - **The turn, or who the current player is, is crucial in deciding whether the position is exactly the same.**
  - **To make it easy, usually positions to be played by different players are stored in different tables.**
- **The recorded entry consists of five parts:**
  - **the position p hashed which can be its hash key;**
  - **the value m';**
  - **the depth $depth'$ where it was recorded;**
  - **a 3-way flag $exact$ indicating whether it is**
    - ▷ *an exact value;*
    - ▷ *a lower bound value causing a beta cut; or*
    - ▷ *an upper bound value causing an alpha cut;*
  - **the child p' where m' comes from or causing a cut to happen.**

TT hit: code

- **May consider using p' in the move ordering.**
- **If $depth' < depth$, namely, we have searched the tree shallower before, then normally**
  - ▷ *cannot use it in the current searching;*
  - ▷ *may use this information in making the move ordering of p's parent better.*
- **If $depth' \geq depth$, namely, we have searched the tree not shallower before.**
  - **It is an exact value.**
    - ▷ *Immediately return m' as the search result.*
  - **It is a lower bound.**
    - ▷ *Raise the $alpha$ value by*
      $$alpha = max\{alpha, m'\}$$
    - ▷ *Check whether this causes a beta cut!*
  - **It is an upper bound.**
    - ▷ *Lower the $beta$ value by*
      $$beta = min\{beta, m'\}$$
    - ▷ *Check whether this causes an alpha cut!*

TT: comments (1/2)

- **The above code $F4.1'$ is the code for the MAX node.**
  - **Need to write similarly for the MIN node $G4.1'$.**
  - **Need to take care of the NegaMAX version $F4.1$.**
- **Reasons you need to make "turn" into the TT design.**
  - **Sometimes, it is possible a legal arrangement of pieces on the board can be reached by both players.**
    - ▷ ***In Go, a player can pass.***
    - ▷ ***In Chinese dark chess, a cannon can capture an opponent piece at a location whose Manhattan distance is an even number in one ply.***
  - **When you do null move pruning (see later slides for details).**

TT: Comments (2/2)

- **Fundamental assumptions:**
  - **Values for positions are history independent.**
  - **The deeper you search, the better result you get.**
    - ▷ *Better in the sense of shorter in the "distance" to the real value of the position.*
- **Need to be able to locate a position p in the transposition table, which is large, efficiently.**
  - **Using a very large transposition table may not be the best.**
    - ▷ *Only some nodes are re-searched frequently.*
    - ▷ *Searching in a very large database is time consuming.*
  - **Some kinds of hash is needed for locating p efficiently.**
    - ▷ *Binary search is normally not fast enough for our purpose.*
- **Need to consider a transposition table aging mechanism.**
  - **Q: Do we really need to reuse information obtained from search a long time or many plys ago?**
  - **Clear a large transposition table takes time.**
  - **Need to weight between the time used in cleaning the transposition table and the mis-information obtained from out of dated information.**

Zobrist's hash function

- Find a hash function $hash(p)$ so that with a very high probability that two distinct positions do not have the same hash value.
- Using bit-wise XOR to realize fast computation.
- Properties of XOR, which is an operator of a commutative group in abstract algebra on the domain of binary strings:
  - associativity: x XOR (y XOR z) = (x XOR y) XOR z
  - commutativity: x XOR $y = y$ XOR x
  - identity: x XOR 0 = 0 XOR $x = x$
  - self inverse: x XOR $x = 0$
  - undo: (x XOR y) XOR $y = x$ XOR (y XOR y) = x XOR 0 = x
- x XOR y is uniformly random if x and y are also uniformly random
  - A binary string is uniformly random if each bit has an equal chance of being 0 and 1.
  - Not all operators, such as OR and AND, can preserve uniform randomness.

Hash function: design

- Assume there are k different pieces and each piece can be placed into r different locations in a 2-player game with red and black players.
  - Obtain $k \cdot r$ random numbers in the form of $s[piece][location]$
  - Obtain another 2 random numbers called $color[red]$ and $color[blk]$.
- Given a position p with $next$ being the color of the next player that has x pieces where q_i is the ith piece and l_i is the location of q_i.
  - $hash(p) = color[next]$ XOR $s[q_1][l_1]$ XOR $\cdots$ XOR $s[q_x][l_x]$
- Comment: can be extended to games with arbitrary number of players, kinds of pieces, and number of pieces.
- We can also remove $color[next]$ from the hash design and maintain 2 hash tables, one for each player.
  - This version will be used in this lecture from now on.

Hash function: update (1/2)

- $hash(p)$ **can be computed incrementally in** $O(1)$ **time.**
  - **Note that computing** $hash(p')$ **from scratch takes time that is linear in the size of** p **which is the number of pieces in** p**.**
  - **Assume** $p' = p + m$ **where** m **is a ply.**
  - **Assume we have computed and store** $hash(p)$**.**
  - **How to obtain** $hash(p')$ **efficiently?**
- **Basic operations:**
  - **If** m **is to place a new piece** q_{x+1} **is placed at location** l_{x+1}**, then**
    - $hash(p') = hash(p)$ ***XOR*** $s[q_{x+1}][l_{x+1}]$**.**
  - **If** m **is to remove a piece** q_y **from location** l_y**, then**
    - $hash(p') = hash(p)$ ***XOR*** $s[q_y][l_y]$**.**
  - **If** m **is to change the next player from** $next$ **to** $\neg next$**, namely,** $pass$**, then**
    - ***Version with color code:***
      $hash(p') = hash(p)$ ***XOR*** $color[next]$ ***XOR*** $color[\neg next]$
      first remove the effect of "XOR $color[next]$***" from*** $hash(p)$***, then add the effect of "XOR*** $color[\neg next]$***";***
    - ***Version without the color code: switch to the other hash table.***

Hash function: update (2/2)

- **Advanced operations:**
  - **A piece q_y is moved from location l_y to location l'_y then**
    - ▷ ***first remove q_y from location l_y, then place it at location l'_y;***
    - ▷ $hash(p') = hash(p)$ **XOR** $s[q_y][l_y]$ **XOR** $s[q_y][l'_y]$.
  - **A piece q_y is moved from location l_y to location l'_y and captures the piece q'_y at l'_y then**
    - ▷ ***first remove q_y from location l_y, then remove q'_y from location l'_y, and finally place q_y at location l'_y;***
    - ▷ $hash(p') = hash(p)$ **XOR** $s[q_y][l_y]$ **XOR** $s[q_y][l'_y]$ **XOR** $s[q'_y][l'_y]$.
  - ...
- **Can use the above primitives to assembly almost, if not all, game playing plys.**
- **It is also easy to undo a ply.**
  - **Perform the XOR operations for the ply again to undo it.**
  - **It is needed in a DFS-like search to undo a move when backtracking is performed.**

Practical issues (1/2)

- **Normally, design a hash table H of 2^n entries, but with a longer key length of $n+m$ bits.**
  - **That is, $color[next]$ and each $s[piece][location]$ are random values each of $n+m$ bits.**
  - **Hash key $= hash(p)$ is $n+m$ bits long.**
  - **Hash index = $hash(p) \mod 2^n$.**
  - **Store the hash key to compare when there is a hash hit.**
    - ▷ ***Longer hash keys ensure better the chance of finding false positive entries.***
    - ▷ ***Usually ≥ 64 bits, and better to be ≥ 128 bits.***
- **How to store/update a hash entry?**
  - **Store it when the entry is empty.**
  - **Use a good updating rule to replace an old entry.**

Practical issues (2/2)

- **How to match an entry?**
  - **First compute hash index** $i = hash(p) \mod 2^n$
  - **Compare** $hash(p)$ **with the stored key in the** i**th entry** $H[i].key$ **to decide whether we have a hit.**
  - **Since the error rate is very small, if** m **is large enough, there is no need to store the exact position in making a comparison.**
  - **Sanity check:**
    - ▷ ***Check whether the stored best move is a legal move in the current position or not.***
    - ▷ ***This will avoid the extreme case of happening of a rare error further by a minimal cost.***

Clustering of errors

- **Errors**
  - **Hash collision**
    - ▷ *Two distinct positions store in the same hash entry, namely have the same hash index.*
  - **Hash clash**
    - ▷ *Two distinct positions have the same hash key.*
- **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.*

Error rates

- **Estimation of the error rate:**
  - **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 (since 2015):**
    - ***$n \leq 32$, and is hardware dependent***
    - ***$w \leq 34$, and is related to resources used in searching.***
    - ***$n + m = 128$ or at least 64 which may not be enough for a large w, and is also hardware related***

Comments

- Zobrist's hash function is now the standard technique used for implementing the transposition table.
- A very good technique that is used in many applications including most game playing codes even the ones that use Monte-Carlo search engines.
- Quality of the random keys is crucial in this design. It is important to spend some efforts to test the quality of the set of random keys picked.
- A must have when you want to efficiently find patterns that change incrementally.
- Can be used in many other applications.
  - pattern based with a finite set of basic components
  - an upper bound on the area is known in advance
  - example: VLSI layout, text documents, line drawings

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, namely move ordering.**
  - **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;**
  - **approximated heuristics:**
    - ▷ *forward pruning;*
    - ▷ *dynamic search extension;*
    - ▷ . . .

Knowledge heuristic

- **Use game domain specified knowledge to obtain a good**
  - **move ordering;**
  - **evaluation function.**
- **Moves that are normally considered good for chess like games:**
  - **Winning moves**
  - **Moves avoiding losing (unchecking)**
  - **Checking moves**
  - **Capturing/avoid-capturing moves**
    - ▷ ***Favor capturing pieces of important.***
    - ▷ ***Favor capturing pieces using pieces as little as possible for capturing symmetry games such as chess because any piece moved can be attacked.***
    - ▷ ***Favor capturing pieces using pieces as large as possible for capturing un-symmetry games such as Chinese dark chess because a large piece moved is less easy to be attacked.***
  - **Moving of pieces with larger material values.**
- **Search good moves first can find the best move faster in any search engine based in alpha-beta or MCTS.**
  - **This is also a must have technique.**

Aspiration search

- **It is seldom the case that you can greatly increase or reduce your chance of winning by playing only one or two plys.**
- **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 when a good heuristic is used [Schaeffer '89].**

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*
      $m := F4(p,m,\infty,current_depth_limit)$
    - ▷ ***endif***
    - ▷ ***endif***
    - ▷ $best := m$ *// found*
    - ▷ ***if another deeper search cannot be done in the remaining time then return*** $best$
    - ▷ $current_depth_limit := current_depth_limit + 1$
  - **return** $best$

IDAS: comments

- **May want to try incrementally reshaping of window sizes.**
  - **For example, use 2 levels t_1 and t_2 where $t1 < t2$.**
  - **try $[best - t_1, best + t_1]$ first.**
  - **If the returned value $best_1$ is exact, then use it.**
  - **If failed low, try $[best_1 - t_2, best_1]$.**
  - **If failed high, try $[best - 1, best + t_2]$.**
  - **$\ldots$**
  - **Need to decide the values of t_i's 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 a better progressive pruning policy for Monte-Carlo based search.**
- **Take a tiny effort to implement.**

Iterative deepening: comments

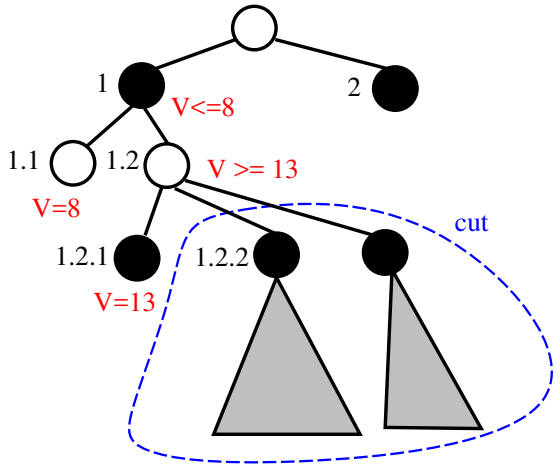
- It is better to increase the search depth each time by 2 because stopping at an odd level is usual too optimistic for the root player.
- Before you start the next round of deeper search, try to estimate whether you have a chance to finish it in time or not.
  - If not, do not waste time for the new round.
  - Save the time for future usage.
- You may want to use the optimal ply obtained from last searching as an important reference for move ordering in this new and deeper search.
- Assume you have obtained a value v_ℓ and a best ply m_ℓ from completely searching of depth=ℓ.
  - In searching deeper depth of $\ell + 2$, time is running up.
  - If none of the branches rooted at the root is completely searched, then return v_ℓ and m_ℓ.
  - If you have completely searched some branches of the root, then use the best value so far obtained from these branches.
  - Do not use any value that is obtained from incompletely searching a branch of the root.

Better move ordering

- **Intuition: the game evolves continuously.**
  - **What are considered good or bad in previous few plys cannot be off too much or too often 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 or too often at this iteration.**
- **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**
  - **the one yielding the best minimax score i.e., in PV, or**
    - ▷ ***Example: move from 1 to 1.1.***
  - **the one that is a sibling of the chosen one that yields the best minimax score and has the same best score, or**
  - **one causes a cutoff.**
    - ▷ ***Remark: this move is potentially good for its parent u, though a cutoff happens may depend on the values of u's older siblings.***
    - ▷ ***Example: move from 1.2 to 1.2.1.***



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 principal variation (PV) or principal continuation.
- Q: What moves are considered good in the context of Monte-Carlo simulation?
  - Can information in Monte-Carlo search accumulated in the previous plys be used in searching this ply?

Refutation tables

- **Assume using iterative deepening with a current search depth limit of $current_depth_limit$ which is bounded by the overall depth limit $limit$.**
  - **Store the current best principal variation at $P_{current_depth_limit,i}$ for each depth i at the current depth limit $current_depth_limit$.**
  - **$PV[i][j]$ is the jth node on the PV when search limit is i.**
- **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 $P_{current_depth_limit-2,i}$;***
    - ▷ $\ldots$
  - **Remark: need to make sure it is a legal move for this node at this moment.**

How to store the PV path

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

    end
  - **return** m

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** $p_1, \ldots, p_b$
  - **// 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 the*** k***th move to be considered***
    - ▷ $k := k + 1$
  - $\ldots$

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 best moves at this depth.
- The next time the same depth in the tree is reached, the killer moves are retrieved and used first in searching, if it is valid in the current position.
- Good codes can be written to efficiently maintain the latest two best moves at depth i in KILLER$[i, 0]$ and KILLER$[i, 1]$.
- Comment:
  - It is plausible to record more than two killer moves. However, the time to maintain them may be too much.
  - Most search engines now record 2 killer moves.

Killer heuristic — sample code

- **The next time the same depth in the tree is reached, the killer moves are retrieved and used first in searching, if it is valid in the current position.**
- **Good codes can be written to efficiently maintain the latest two best moves at depth i in KILLER$[i,0]$ and KILLER$[i,1]$.**
  - **Always make sure KILLER$[i,0]$ occurs before KILLER$[i,1]$ so that it can reused effectively.**
  - **Let b be the newly found good move.**
  - **if KILLER$[i,0]$ is empty then // killers are empty KILLER$[i,0] = b$ // found a killer b**
  - **else if $b ==$ KILLER$[i,0]$ then // refound b**
    - ▷ ***if KILLER$[i,1]$ is not empty // killer 1 is now older***
    - ▷ ***then swap KILLER$[i,0]$ and KILLER$[i,1]$***
  - **else if $b \neq$ KILLER$[i,1]$ then // found a new killer**
    - ▷ ***// flush out killer 0***
    - ▷ ***KILLER$[i,0] =$ KILLER$[i,1]$***
    - ▷ ***KILLER$[i,1] = b$***

History heuristic

- **Intuition:**
  - **A move m is shown to be the best in one position p.**
  - **Later on in the search tree a similar position p' 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 so that a ply m is best in p, but is very bad in p'.**
- **Recall: In alpha-beta search, a sufficient, or good, move at an interior node is defined as**
  - **the one yielding the best minimax score, or**
  - **a move that is "equivalent" to the best move in terms of score, or**
  - **one causes a cutoff**

Implementation (1/2)

- **Keep track of the history on what moves were good before.**
  - **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 that is moved, does 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.**
    - ▷ *May use different weights for cutoff moves or truly best moves.*
  - **Be careful: 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 is, except in pathological trees which are rarely seen in practice.**
  - **The longer the tree searched, and hence larger, the greater the differences between two arbitrary positions in the tree are, 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** $p_1, \ldots, p_b$
  - **if** $b = 0$ **then return** $f(p)$ **else// a terminal node**
  - **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*
      $where := 1$; *// $where$ is the child best comes from*
      if m ***is max or*** $m \geq beta$ ***then*** $\{$ $HT[p_1]$ $+=$ $weight1$; ***return***$(m)\}$ *// beta cut off*
    - ***for*** $i := 2$ ***to*** b ***do***
    - ***9:*** $\{t := G4.3'(p_i, m, m + 1, depth - 1)$; *// null window search*
    - ***10:*** ***if*** $t > m$ ***then*** *// failed-high*
      $\{where := i$; *// $where$ is the child best comes from*
      11: ***if*** $(depth < 3$ ***or*** $t \geq beta)$
      12: ***then*** $m := t$
      13: ***else*** $m := G4.3'(p_i, t, beta, depth - 1)\}$ *// re-search*
    - ***14:*** ***if*** m ***is max or*** $m \geq beta$ ***then*** $\{HT[p_i]$ $+=$ $weight1$; ***return***$(m)\}\}$
      // beta cut off

    end
  - $HT[p_{where}]$ **+=** $weight2$**; return** m

History heuristic: usage of the counter

- **Algorithm** $F4.3.1'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **order the legal moves** $p_1, \ldots, p_b$ **according to their weights in HT[ ]**
  - **// Good sorting codes are needed here to sort a short list of values.**
  - $\ldots$

Comments: better move ordering

- **Need a good sorting routine in $F4.3.1'$ to order the legal moves according to their history values.**
  - **The number of possible moves is small.**
    - ▷ *Better sorting methods are known for very small number of objects.*
- **Need to take care of the case for counter overflowing.**
  - **Need to perform counter aging periodically.**
    - ▷ *That is, discount the value of the current counter as the game goes.*
    - ▷ *For example after a second has past, $HT[i] >>= 1$*
    - ▷ *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 a 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 [Schaeffer '89].**

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.
  - This conclusion is implementation and performance dependent.
- 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.
- With the ever increasing CPU speed, it may be profitable to use more than 2 techniques now.
- Better tools and techniques, such as hyper parameter optimization and gradient descent, are known now from deep learning studies to fine tune parameters.

Dynamically adjusting searching depth

- **Aggressive forward pruning: do not search too deep on branches that seem to have little chance of being the best.**
  - **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.**
- **Comments:**
  - **Similar ideas are shared by MCTS search by spending less time in hopeless branches, and more time in hopeful branches.**
  - **Spend at least some time in seems hopeless branches.**
  - **Maybe possible to come out with a hybrid technique.**

Null move pruning — MAX node

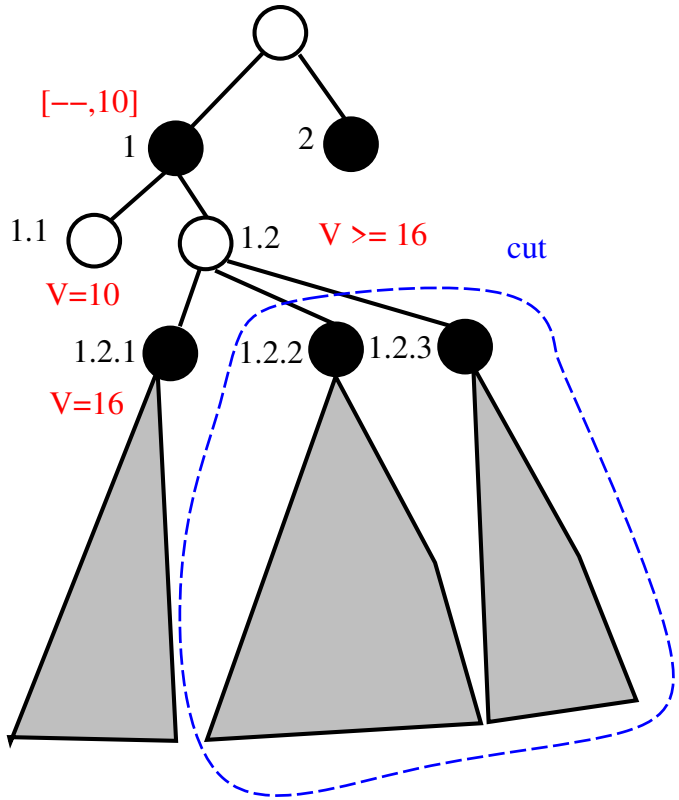
- **In general, if you forfeit the right to move and can still maintain the current advantage in a small number of plys played later, then it is usually true you can maintain the advantage in a larger number of plys later if you do not forfeit the right to move.**
- **Algorithm:**
  - **It's your turn to move (MAX node); the searching depth for this node is $depth$ with a search window of (α, β).**
  - **Make a null move, i.e., assume you do not move and let the opponent move again.**
    - ***Perform an null-window $[beta - 1, beta]$ alpha-beta search with a reduced depth $(depth - R)$, 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.***
      $\Rightarrow$ Allowing your opponent to move twice does not produce a better, lower valued score — lower than β, position for him.
    - ***If $v < beta$, namely does not produce a cutoff, then do the normal alpha-beta search.***
- **Similar ideas work for the case of your opponent's turn to move.**

Null move pruning — MAX node

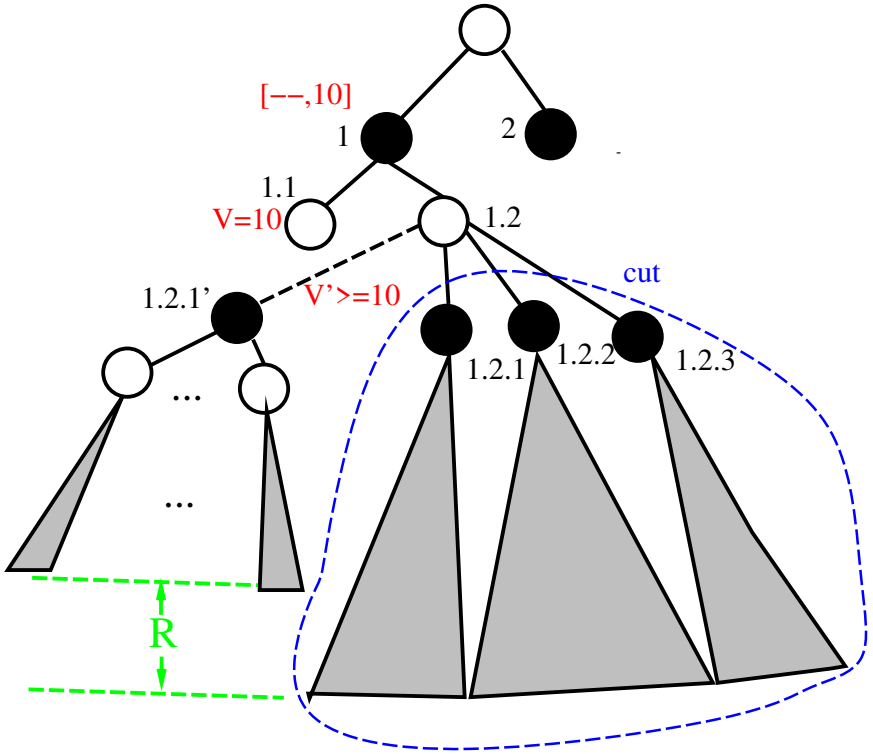
- **Algorithm** $F4.4'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**, Boolean** in_null**)**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **// a terminal node**
    $\ldots$
  - **then return** $f(p)$ **else begin**
    - ▷ ***If*** $depth \leq R + 3$ ***or*** in_null ***or dangerous, then goto Skip;***
    - ▷ ***// null move pruning***
    - ▷ $null_score := G4.4'(p', beta - 1, beta, depth - R - 1, TRUE)$ ***// p' is the position obtained by switching the player in p, and R is usually 2***
    - ▷ ***if*** $null_score$ ***is max or*** $null_score \geq beta$ ***return*** $null_score$ ***// null move pruning***
    - ▷ ***Skip: // normal alpha-beta search***
    - ▷ $m := -\infty$ ***// m is the current best lower bound; fail soft***
    - ▷ $m := \max\{m, G4.4'(p_1, alpha, beta, depth - 1, in_null)\}$
    - ▷ ***if*** m ***is max or*** $m \geq beta$ ***then return***(m) ***// beta cut off***
    - ▷ ***for*** $i := 2$ ***to*** b ***do***
    - ▷ $\ldots$

    end
  - **return** m

Null move pruning — MAX node illustration



alpha−beta prune



null move prune

Null move pruning — MIN node

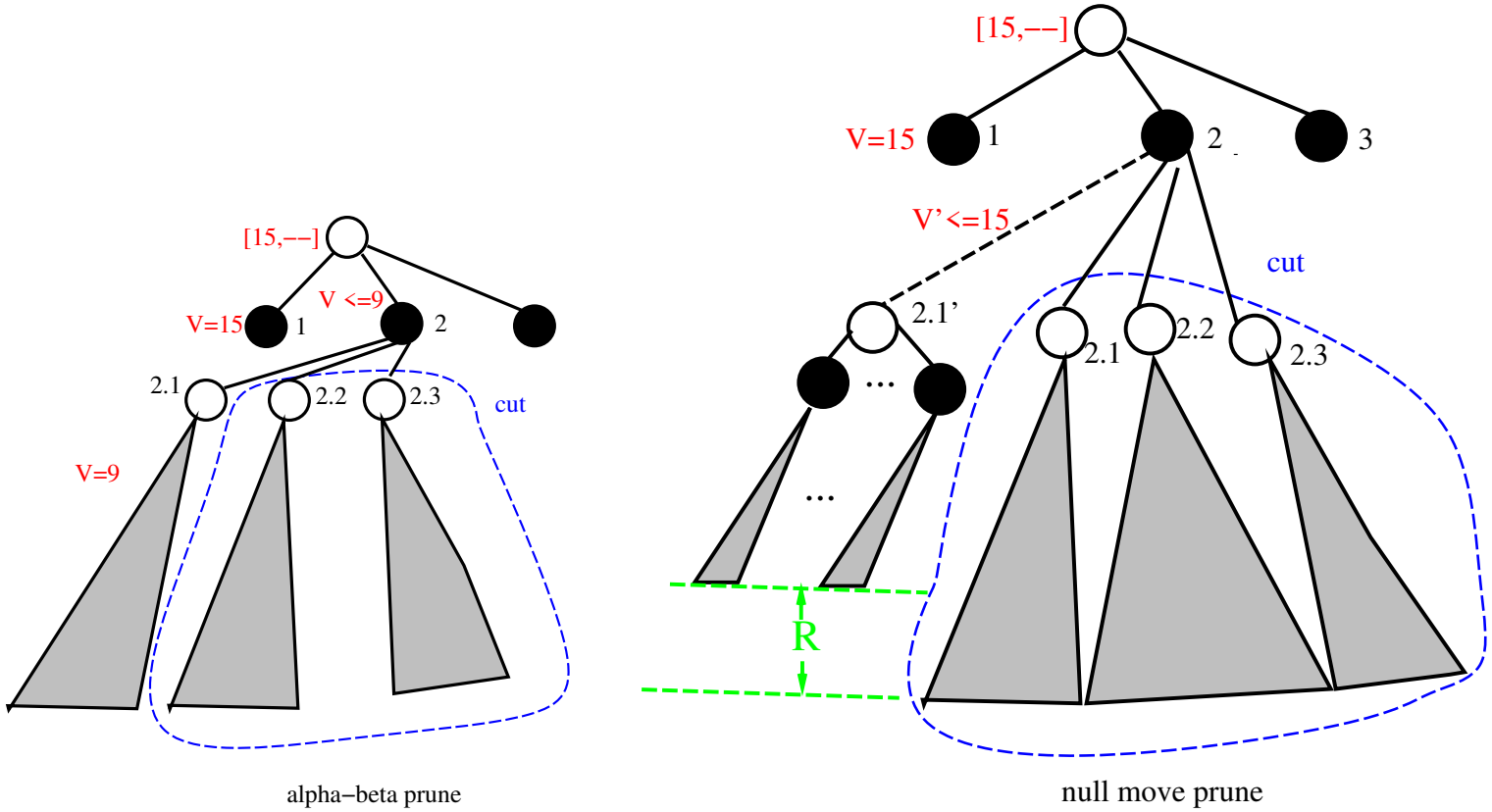
- **If your opponent forfeit the right to move and can still maintain the current advantage in a small number of plys played later, then it is usually true your opponent can maintain the advantage in a larger number of plys later if he do not forfeit the right to move.**
  - **Note that a score that is good to your opponent is bad to you.**
- **Algorithm:**
  - **It's your opponent's turn to move; the searching depth for this node is $depth$ with a search window (α, β).**
  - **Make a null move, i.e., assume your opponent do not move and let you move again.**
    - ***Perform an null-window $[alpha, ahpha + 1]$ alpha-beta search with a reduced depth $(depth - R)$, where R is a constant decided by experiments.***
    - ***If the returned value v is at most $alpha$, then apply an $alpha$ cutoff and return v as the value.***
      $\Rightarrow$ Allowing you to move twice and you cannot find a better position, score more than α, then it is no needed to consider this branch.
    - ***If $v > alpha$, namely does not produce a cutoff, then do the normal alpha-beta search.***

Null move pruning — MIN node algorithm

- **Algorithm** $G4.4'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**, Boolean** in_null**)**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **// a terminal node**
    $\ldots$
  - **then return** $f(p)$ **else begin**
    - ▷ ***If*** $depth \leq R + 3$ ***or*** in_null ***or dangerous, then goto Skip;***
    - ▷ ***// null move pruning***
    - ▷ $null_score := F4.4'(p', alpha, alpha + 1, depth - R - 1, TRUE)$***// p' is the position obtained by switching the player in p, and R is usually 2***
    - ▷ ***if*** $null_score$ ***is min or*** $null_score \leq alpha$ ***return*** $null_score$ ***// null move pruning***
    - ▷ ***Skip: // normal alpha-beta search***
    - ▷ $m := \infty$ ***// m is the current best upper bound; fail soft***
    - ▷ $m := \min\{m, F4.4'(p_1, alpha, beta, depth - 1, in_null)\}$
    - ▷ ***if*** m ***is min or*** $m \leq alpha$ ***then return***(m) ***// alpha cut off***
    - ▷ ***for*** $i := 2$ ***to*** b ***do***
    - ▷ $\ldots$

    end
  - **return** m

Null move pruning — MIN node illustration



Null move pruning: analysis

- **Assumptions:**
  - **The depth reduced, R, is usually 2 or 3.**
  - **The advantage of doing a null move can offset the errors produced from doing a shallow search.**
  - **Usually do not apply null move pruning when**
    - ▷ *your king is in danger, e.g., in check;*
    - ▷ *when the number of remaining pieces is small;*
    - ▷ *when there is a chance of Zugzwang;*
    - ▷ *when the number of remaining depth is small, for example* $\leq R + 3$.
  - **It is usually not a good idea to do this recursively, i.e., when in_null flag is true.**
- **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 [Heinz '00].**
- **Do not store the null move cut off into the transposition table since it cuts off a node that does not need to be searched by its parent due to the goodness of an older sibling.**

Late move reduction (LMR)

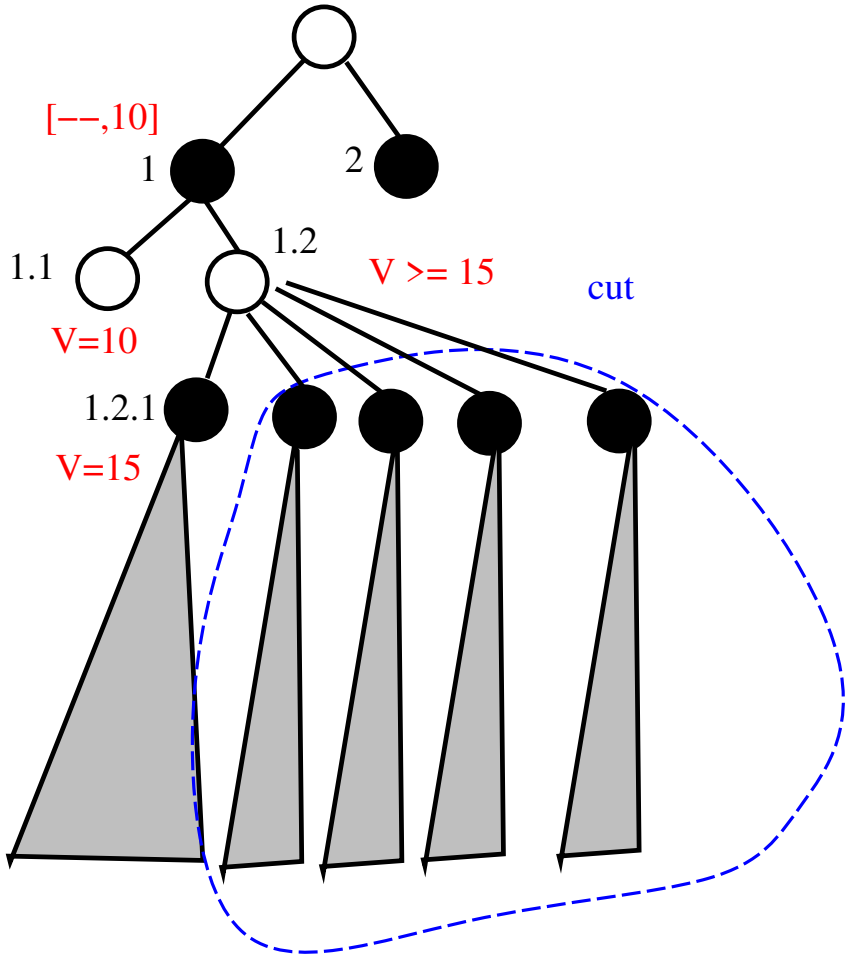
- **Assumption:**
  - **Move ordering is relatively good.**
    - ▷ *Verify this experimentally.*
- **Observation:**
  - **During search, the best move rarely comes from moves that are ordered very late in the move queue.**
- **How to make use of the observation:**
  - **If the first K, say $K = 3$ or 4, moves considered do not produce a value that is better than the current best value, then**
    - ▷ *consider reducing the depth of the rest of the moves with H, say $H = 3$.*
  - **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 — Algorithm

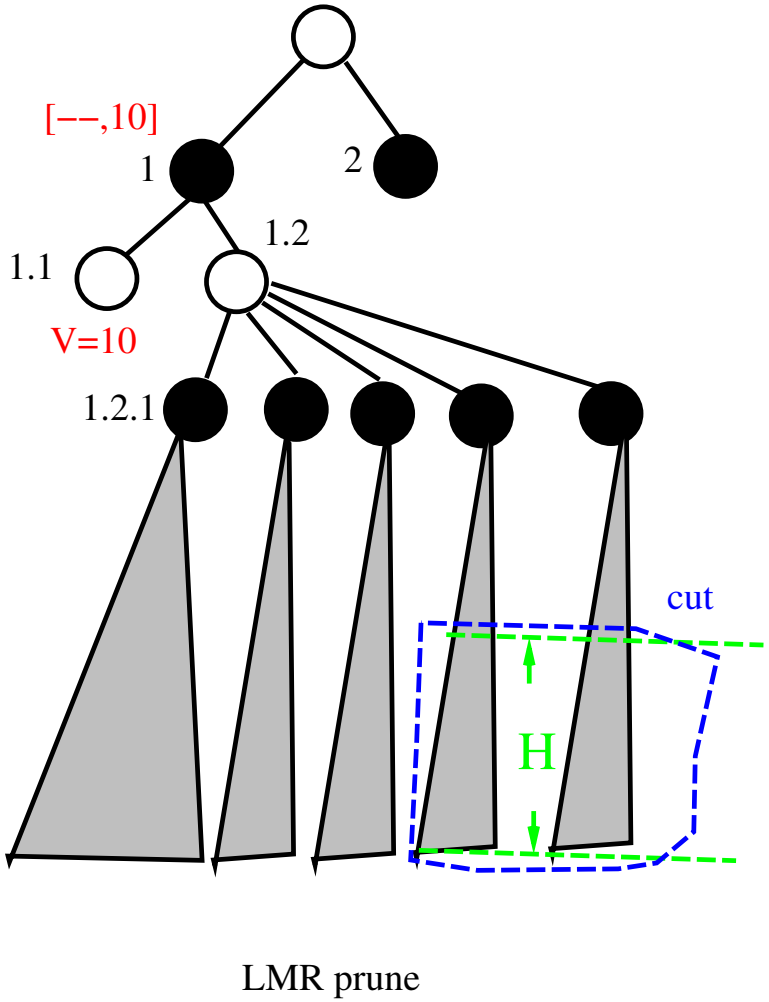
- **Algorithm** $F4.5'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**, Boolean** in_lmr**)**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **// a terminal node**
  - **then return** $f(p)$ **else**
    begin
    - $m := -\infty$ *// m is the current best lower bound; fail soft*
      $\ldots$
    - ***for*** $i := 2$ ***to*** b ***do***
    - ***if*** in_lmr ***or*** $i \leq K$ ***or*** $depth \leq H + 3$ ***or*** p_i ***is dangerous,***
      then $\{depth' = depth;\ flag := in_lmr;\}$ *//no LMR*
      else $\{depth' := depth - H;\ flag := true\}$*; //depth reduced*
    - ***9:*** $t := G4.5'(p_i, m, m + 1, depth' - 1, flag)$ *// null window search*
    - ***10:*** ***if*** $t > m$ ***then*** *// failed-high*
      11: ***if*** $(depth' < 3$ ***or*** $t \geq beta)$
      12: ***then*** $m := t$
      13: ***else*** $m := G4.5'(p_i, t, beta, depth - 1, in_lmr)$ *// re-search*
    - ***14:*** ***if*** m ***is max or*** $m \geq beta$ ***then return***(m) *// beta cut off*

    end
  - **return** m

LMR — Example



alpha−beta prune



LMR prune

LMR: analysis

- **Performance:**
  - Reduce the **effective branching factor** to about K.
    - ▷ ***Effective branching factor** is the average number of children considered in full details.*
  - Experience: very effective, namely can search 2 levels deeper in the same time constraint, when move ordering is good.
- **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.
- **It is usually not a good idea to do this recursively, i.e., when** in_lmr **flag is true.**

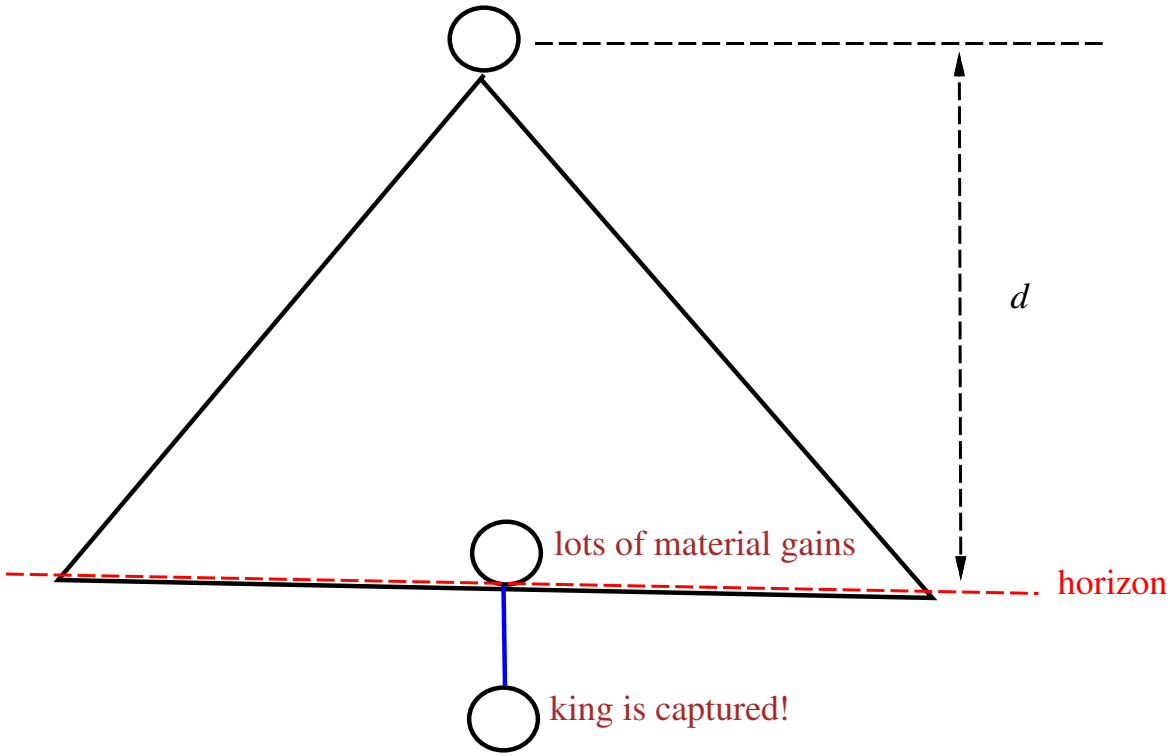
Comments

- **Null move pruning and LMR are implemented together with the usual alpha-beta pruning.**
  - **You first try null move pruning and if it does not work, you re-search with the usual alpha-beta part with LMR.**
  - **For LMR, you use the normal alpha-beta part for the first few branches. Then you do extra cutoff on later branches if those branches are not cut previously.**

Dynamic search extension

- **Search extensions**
  - **Some nodes need to be explored deeper than the others to avoid the horizon effect.**
    - ▷ ***Horizon 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.***

Horizon effect

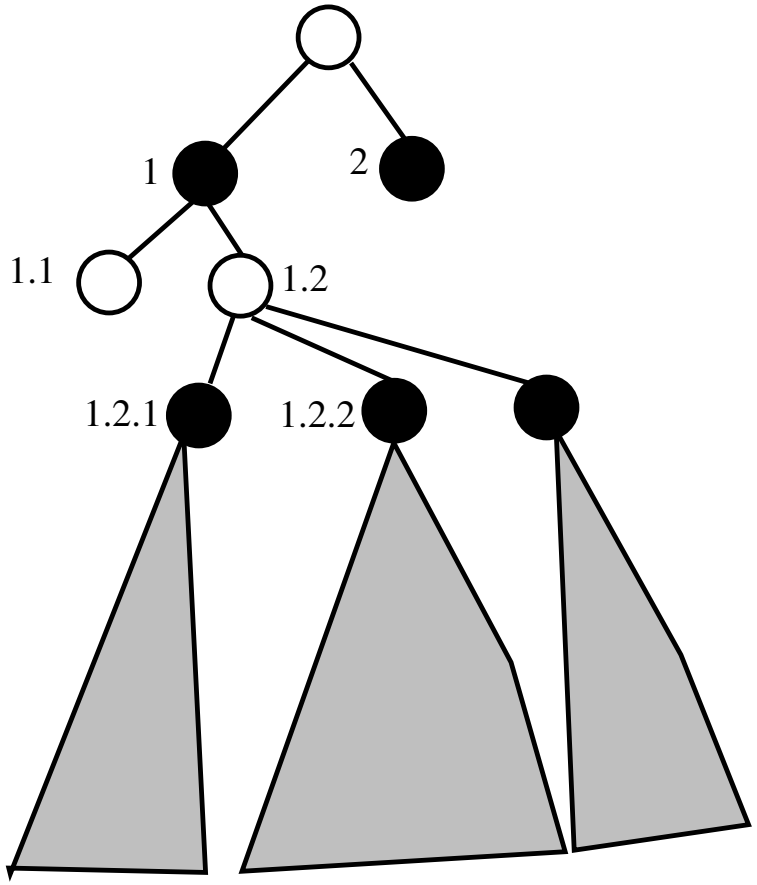


Dynamic depth extension — Algorithm

- **Algorithm** $F4.6'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **or** $depth = 0 \cdots$ // **a terminal node**
    $\ldots$
  - **then return** $f(p)$ // **a terminal node**
    else begin
    - ▷ ***if*** p_1 ***is unstable, 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 ***is max or*** $m \geq beta$ ***then return***(m) // ***beta cut off***
    - ▷ ***for*** $i := 2$ ***to*** b ***do***
    - ▷ ***if*** p_i ***is unstable, 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 ***is max or*** $m \geq beta$ ***then return***(m)**}** // ***beta cut off***

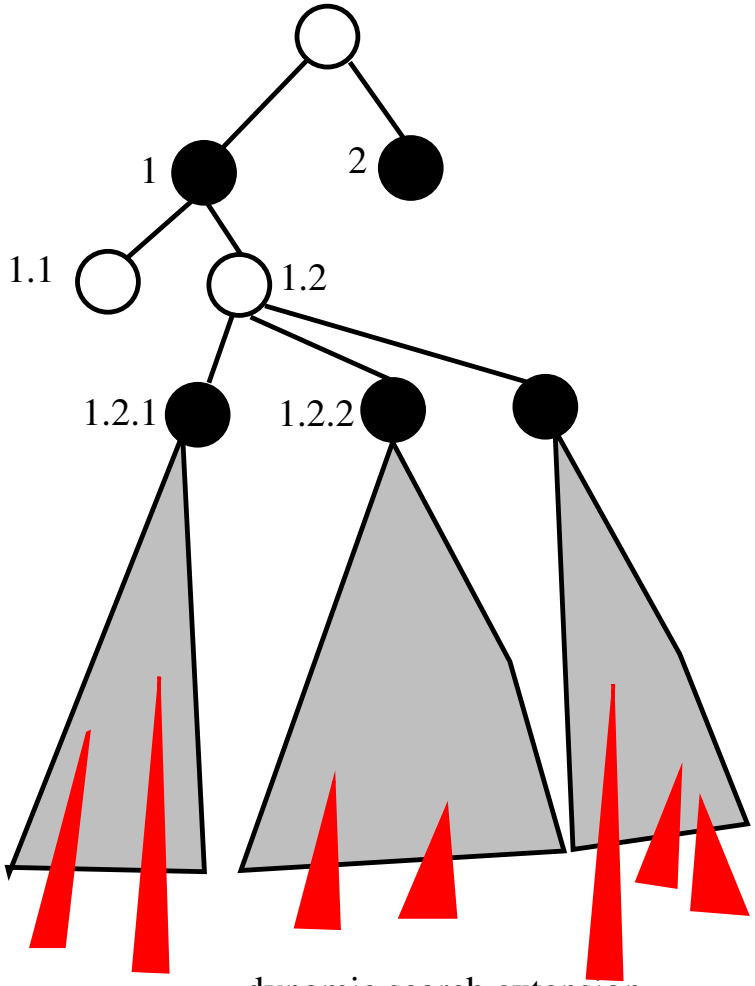
    end
  - **return** m

DSE — Illustration



normal search

search extension



dynamic search extension

Quiescent search (1/2)

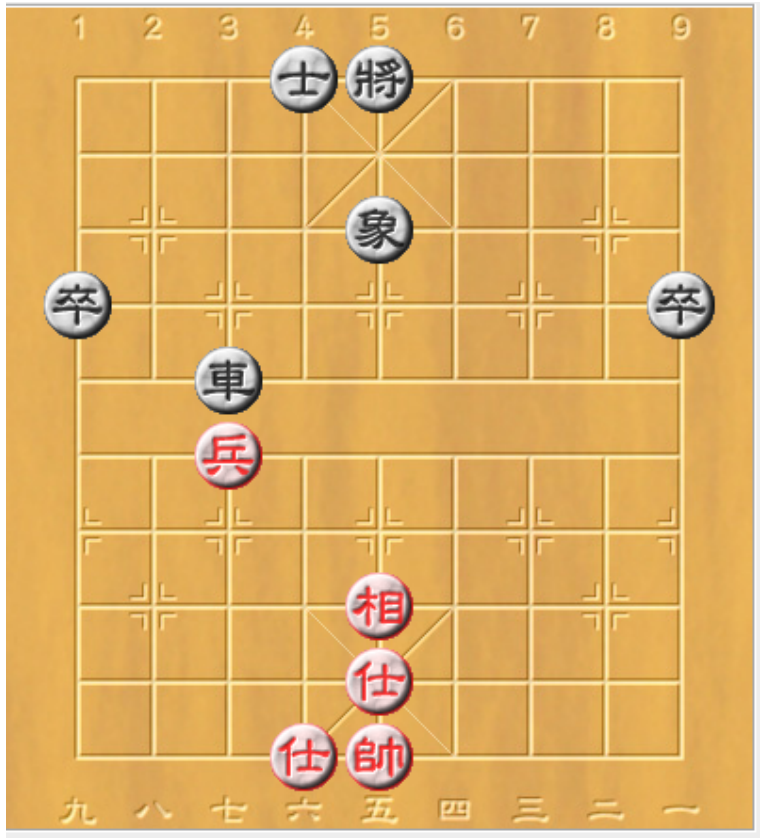
- **Quiescent search: to check further on only forceful variations.**
  - **Invoke your search engine, e.g., alpha-beta search, to only consider moves that are in-check or capturing.**
    - ▷ *May also consider checking moves.*
    - ▷ *May also consider allowing upto a fixed number, say 1, of non-capturing moves in a search path.*
  - **Watch out of unneeded piece exchanges by checking the Static Exchange Evaluation (SEE) value first.**

Quiescent search (2/2)

- **We invoke a quiescent search so that searching is not stopped in the middle of a sequence of forced actions and counter-actions due to a fixed searching depth limit.**
  - **A sequence of checking and unchecking and finally leads to checkmate.**
  - **A sequence of moves with very limited number of choices.**
  - **A sequence of piece exchanges.**
    - ▷ ***It is p's turns to move, p will carry on the rest of exchanges only if he will be profitable.***

Illustrations

- **Example: red pawn will capture black rook if it's red's turn, but black rook will not capture red pawn if it's black's turn.**



Dynamic depth extension — Algorithm

- **Algorithm** $F4.7'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **// a terminal node**
    $\ldots$
  - **then return** $Quiescent_F'(p, alpha, beta)$
    else
    begin
    ▷ ***continue to search***
    ▷ $\ldots$
    end
  - **return** m

Quiescent search algorithm

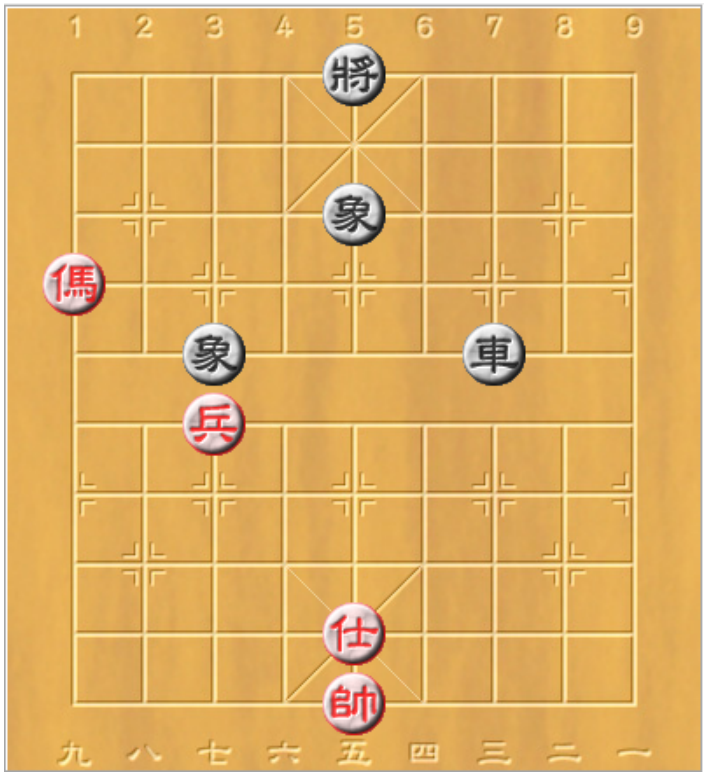
- **Algorithm** $Quiescent_F'$**(position** p**, value** $alpha$**, value** $beta$**)**
  - **generate the successor positions** $p_1, \ldots, p_{b'}$ **such that each** $p \rightarrow p_i$ **is**
    - ***capturing,***
    - ***unchecking, or***
    - ***checking // may add other types of non-quiescent moves***
  - **if** $b' = 0$ **then return** $f(p)$ **// a quiescent position**
  - **else** $m := -\infty$
  - $quies := 0$; **// count the number of quiescent capturing moves**
  - **for** $i := 1$ **to** b' **do**
    - ***if*** $p \rightarrow p_i$ ***is not a capturing move OR*** $SEE(destination(p_i)) > 0$ ***then // not a quiescent position, search deeper***
      {$m := max\{m, Quiescent_G'(p_i, max\{m, alpha\}, beta)\}$
      if m ***is max or*** $m \geq beta$ ***then return*** (m)***}*** *// beta cut off*
      else $quies := quies + 1$
  - **if** $quies = b'$ **then return** $f(p)$ **// a quiescent position**
    else return m**;**
- **Can also use NegaScout as the main search engine.**

Algorithm SEE($location$)

- **Assume w.l.o.g. it is red's turn and there is a black piece bp at $location$. Compute the net gain of materials for a fight at $location$.**
- **Algorithm SEE($location$)**
  - $R :=$ **the list of red pieces that can capture a black piece at** $location$**.**
  - **if** $R = \emptyset$**, then return 0;**
  - **Sort** R **according to their material values in non-decreasing order.**
  - $B :=$ **the list of black pieces that can capture a red piece at** $location$**.**
  - **Sort** B **according to their material values in non-decreasing order.**
  - $gain := 0$; $piece := bp$;
  - **While** $R \neq \emptyset$ **do**
    - ▷ ***capture*** $piece$ ***at*** $location$ ***using the first element*** w ***in*** R***;***
    - ▷ ***remove*** w ***from*** R***;***
    - ▷ $gain := gain + value(piece)$***;***
    - ▷ $piece := w$***;***
    - ▷ ***if*** $B \neq \emptyset$
      then { ***capture*** $piece$ ***at*** $location$ ***using the first element*** h ***in*** B***;***
      remove h ***from*** B***;*** $gain := gain - value(piece)$***;*** $piece := h$***;*** }
      else break;
  - **return** $gain$ **//the net gain of material values during the exchange**

Example

- **Net gain in red's turn.**
  - **Captured: two black elephants**
  - **Be captured: a red pawn and a red horse.**
  - **Usually, a pawn and a horse are more valuable than two elephants.**
    - ▷ ***Hence this is a Quiescent position for the red side.***

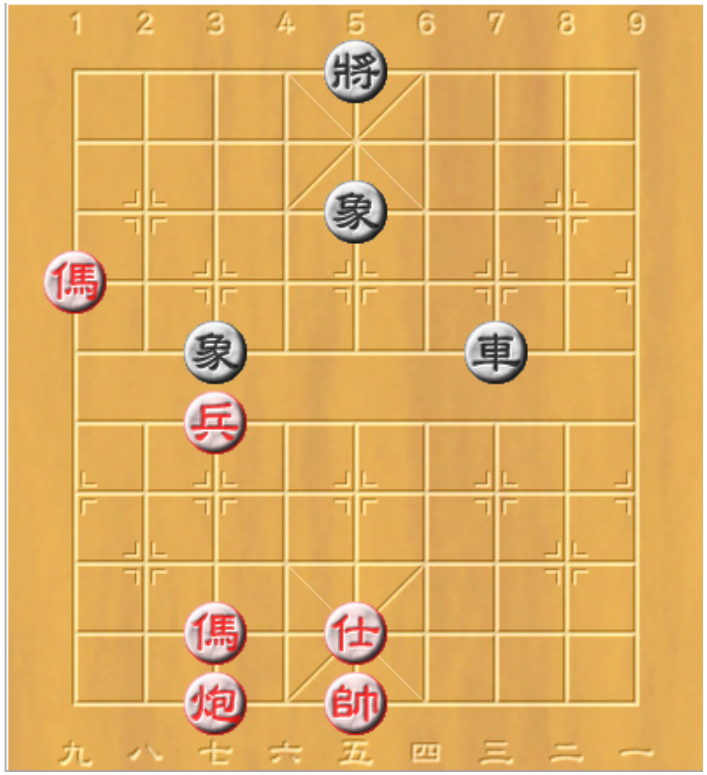


SEE: Comments

- We carry out a capturing move in Quiescent search only if the net gain is positive.
- Always use a lower valued piece to capture if there are two choices for getting the best gain.
- SEE is static and imprecise for performance issues.
  - Some pieces may capture or not able to capture a piece at a location because of the exchanges carried out before.
  - If SEE considers more dynamic situations, then it costs more time.

Counter example of SEE

- **Red cannon attack the location where the black elephant was at the river after red pawn captures this black elephant, and then the black elephant captures the red pawn.**
  - **SEE advises RED not to initiate piece exchanges.**
  - **In this case, RED actually needs to do so.**



Comments for SEE implementation

- **Can store the searched results from applying Quiescent search or even SEE into a transposition table.**
- **Usually, use separate transposition tables for main search, Quiescent search and SEE.**

SEE for capturing un-symmetry games and otl

- The above SEE is mainly for capturing symmetry games such as Chinese chess where a piece can capture any opponent's piece.
- For Chinese dark chess that is capturing un-symmetry where the pieces are ranked, a different estimation is needed.
  - Assume a black piece at a location can be attacked by a list of red pieces R and defended by a list of black pieces B.
  - If R and B do not contain cannon, then R will start an exchange only if R's largest ranked piece cannot be captured by a piece in B.
  - If R has a cannon, then consider whether the value of the black piece captured is more valuable than cannon.
  - If B also has a cannon, then re-consider whether it gains to exchange.
  - Since the cardinalities of R and B are small, one may want to skip SEE before invoking quiescent search.
- For stochastic games, it takes more efforts to do quiescent searching.
  - CDC: consider among only non-flipping moves.
  - EWN: may need to check whether there are possible capturing/checkmating moves for all dice outcomes.

Concluding comments

- **There are many 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?**
    - ▷ *May want to look into hyper-parameter optimization used in machine learning [Bergstra and Yoshua'12].*
  - **The coefficients need to be re-tuned once the searching behaviors change.**
    - ▷ *Changing evaluation functions.*
    - ▷ *Faster hardware so that the searching depth is increased.*
    - ▷ . . .
- **Need to consider tradeoff between the time spent and the amount of improvements obtained.**

References and further readings (1/2)

* 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.
- E. A. Heinz. *Scalable Search in Computer Chess*. Vieweg, 2000. ISBN: 3-528-05732-7.

References and further readings (2/2)

- **S.C. Hsu. Searching Techniques of Computer Game Playing.** *Bulletin of the College of Engineering, National Taiwan University*, **51:17–31, 1991.**
- **Bergstra, James and Bengio, Yoshua. Random Search for Hyper-Parameter Optimization** *Journal of Machine Learning Research. 13: 281–305, 2012*