An Update on Game Tree Research
Akihiro Kishimoto and Martin Mueller

Tutorial 3: Alpha-Beta Search and Enhancements

Presenter:
Akihiro Kishimoto, IBM Research - Ireland
Outline of this Talk

- Techniques to play games with alpha-beta algorithm
  - Alpha-beta search and its variants
  - Search enhancements
  - Search extension and reduction
  - Evaluation and machine learning
  - Parallelism
Alpha-Beta Algorithm

• Unnecessary to visit every node to compute the true minimax score
  • E.g. \( \max(20, \min(5, X)) = 20 \), because \( \min(5, X) \leq 5 \) always holds
  • Idea: Omit calculating \( X \)

• Idea: keep upper and lower bounds \((\alpha, \beta)\) on the true minimax score

• Prune a position if its score \( v \) falls outside the window
  • If \( v < \alpha \) we will avoid it, we have a better-or-equal alternative
  • If \( v \geq \beta \) opponent will avoid it, they have a better alternative
How Does Alpha-Beta Work? (1 / 2)

• Let \( v \) be score of node, \( v_1, v_2, \ldots, v_k \) scores of children
• By definition: in MAX node, \( v = \max(v_1, v_2, \ldots, v_k) \)
• By definition: in MIN node, \( v = \min(v_1, v_2, \ldots, v_k) \)
• Fully evaluated moves establish lower bound
  • E.g., if \( v_1 = 5 \), \( \max(5, v_2, \ldots, v_k) \geq 5 \)
• Other moves of score \( \leq 5 \) do not help us, can be pruned
How Does Alpha-Beta Work? (2 / 2)

- Similar reasoning at MIN node – move establishes upper bound
  - E.g., $v=2$, $v=\min(2, v_2, ..., v_k) \leq 2$
- If a move leads to position that is too bad for one of the players, then cut.
Alpha-Beta Algorithm – Pseudo Code

```c
int AlphaBeta(GameState state, int alpha, int beta, int depth) {
    if (state.IsTerminal() or depth == 0)
        return state.StaticallyEvaluate();
    score = -INF;
    foreach legal move m from state
        state.Execute(m)
        score = max(score, -AlphaBeta(state, -beta, -alpha, depth-1))
    alpha = max(score, alpha)
    state.Undo()
    if (alpha >= beta) // Cut-off
        return alpha // Cut-off
    return score
}
```

This is a negamax formulation.
Initial call: AlphaBeta(root, -INF, INF, depth_to_search)
Example of Alpha-Beta Algorithm

principal variation

Cutoff
Principal Variation (PV)

- Sequence where both sides play a strongest move
- All nodes along PV have the same value as the root
- Neither player can improve upon PV moves
- There may be many different PV if players have equally good move choices
- The term PV is typically used for the *first* sequence discovered. Others are cut off by pruning
Properties of Alpha-Beta

• Number of nodes examined
  • Best case: \( b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1 \) (see minimal tree, next slide)
  • Basic minimax: \( O(b^d) \)
    \( b \): branching factor, \( d \): depth

• Assuming score \( v \) is obtained after alpha-beta searches with window \((\alpha, \beta)\) at node \( n \), real score \( sc \) is:
  • If \( v \leq \alpha \): fail low, \( sc \leq v \),
  • if \( \alpha < v < \beta \): exact, \( sc = v \), and
  • if \( \beta \leq v \): fail high, \( sc \geq v \)

\( \textbf{We will keep using this property in this lecture} \)
Minimal Tree

Tree generated by alpha-beta with perfect ordering
- 3 types of nodes (PV, CUT, and ALL)
Reducing the Search Window

• Classical alpha-beta starts with window (-INF,INF)
• Cutoffs happen only after first move has been searched
• What if we have a “good guess” where the minimax value will be?
  • E.g., “Aspiration window” in chess: take score from last move, (-one-pawn, +one-pawn) or so
• Gamble: can reduce search effort, but can fail
Other Alpha-Beta Based Algorithms

• Idea: smaller windows cause more cutoffs

• Null window \((\alpha, \alpha+1)\) – equivalent to Boolean search
  • Answer question whether \(v \leq \alpha\) or \(v > \alpha\)

• With good move ordering, score of first move will allow to cut all other branches

• Change search strategy. Speculative, but remain exact by re-search if needed

• Scout by Judea Pearl, NegaScout by Reinefeld: use null window searches to try to cut all moves but the first

• PVS – principal variation search, equivalent to NegaScout
PVS/NegaScout
[Marsland & Campbell, 1982] [Reinefeld, 1983]

• Idea: search first move fully to establish a lower bound $v$
• Null window search to try to prove that other moves have score $\leq v$
• If fail high, re-search to establish exact score of new, better move
• With good move ordering, re-search rarely needed. Savings from using null window outweigh cost of re-search
NegaScout Pseudo-Code

```c
int NegaScout(GameState state, int alpha, int beta, int depth) {
    if (state.IsTerminal() || depth == 0)
        return state.Evaluate()
    b = beta
    bestScore = -INF
    foreach legal move mi i=1,2,.. from state
        State.Execute(mi)
        int score = -NegaScout(state, -b, -alpha, depth – 1)
        if (score > alpha && score < beta && i > 1) // re-search
            score = -NegaScout(state, -beta, -score, depth – 1)
        bestScore = max(bestScore, score)
        alpha = max(alpha, score)
    state.Undo()
    if (alpha >= beta)
        return alpha
    b = alpha + 1
    return bestScore
}
```

Note for experts: A condition to reduce re-search overhead is removed here. See [Reinefeld, 1983][Plaat,1996] for details
Search Enhancements

• Basic alpha-beta is simple but limited

• Need many enhancements to create high-performance game-playing programs

• General (game-independent, algorithm-independent) and specific

• Depends on many things: size, structure of search tree, availability of domain knowledge, speed versus quality tradeoff, parallel versus sequential

• Look at some of the most important ones in practice
Enhancements to Alpha-Beta

There are several types of enhancements

- Exact (guarantee minimax value) versus inexact
- Improve move ordering (reduce tree size)
- Improve search behavior
- Improve search space (pruning)
Iterative Deepening

• Series of depth-limited searches \( d = (0), 1, 2, 3, \ldots \)

• Advantages
  • Anytime algorithm – first iterations are very fast
  • If branching factor is big, small overhead – last search dominates
  • With transposition table (explain later), store best move from previous iteration to improve move ordering
  • In practice, usually searches less than without iterative deepening

• Some game programs increase \( d \) in steps of 2
  • E.g. odd/even fluctuations in evaluation, small branching factor
Iterative Deepening and Time Control

- With fixed time limit, last iteration must usually be aborted
- Always store best move from recent completed iteration
- Try to predict if another iteration can be completed
- Can use incomplete last iteration if at least one move searched (however, the first move is by far the slowest)
Transposition Table (1 / 3)

- Idea: Cache and reuse information about previous search by using hash table
- Avoid searching the same subtree twice
- Get best move information from earlier, shallower searches
- Essential in DAGs where many paths to same node exist
  - Discuss issues in solving games/game positions
- Help significantly even in trees e.g. with iterative deepening
- Replace existing results with new ones if TT is filled up
Transposition Table (2 / 3)

• Typical TT Content
  • Hash code of state (usually not one-on-one, but astronomically small error of different states with identical hash code)
  
  See [http://chessprogramming.wikispaces.com/Zobrist+Hashing](http://chessprogramming.wikispaces.com/Zobrist+Hashing)

• Evaluation
• Flags – exact value, upper bound, lower bound
• Search depth
• Best move in previous iteration
Transposition Table (3 / 3)

- When $n$ is examined with $(\alpha, \beta)$, retrieve information TT
- Do not examine $n$ further if TT information indicates:
  - Node $n$ is examined *deep enough and*
  - TT contains exact value for $n$, or
  - Upperbound in TT $\leq \alpha$, or
  - Lowerbound in TT $\geq \beta$
- Try best move in TT first if $n$ needs to be examined:
  - Best move is often stored in previous iterations
  - Usually causes more cutoffs than without iterative deepening even if search space is tree
- Save evaluation value, search depth, best move etc in TT after $n$ is examined
Move Ordering

• Good move ordering is essential for efficient search
• Iterative deepening is effective
• Often use game-specific ordering heuristics e.g. mate threats
• More general: use game-specific evaluation function
History Heuristic
[Schaeffer 1983, 1989]

- Improve move ordering without game-specific knowledge
- Give bonus for moves that lead to cutoff such as
  
  - history_table[color][move] += $d^2$
  
  - history_table[color][move] += $2^d$ ($d$: remaining depth)
- Prefer those moves at other places in the search
- Will see later in MCTS – all-moves-as-first heuristic, RAVE
- History heuristic might not be as effective as it used to be but is effectively combined with late move reduction (later)
  
  - E.g. Chess program Stockfish gives a penalty for “quiet moves” that do not cause cut-offs
Performance Comparison of Alpha-Beta Enhancements

C.f. Figure 8 in [Marsland, 1986]
MTD(f) [Plaat et al, 1996]

- PVS, NegaScout: full window search for move 1, null window searches for moves 2, 3, ...

- Idea: Only null window searches (γ,γ+1) that can check either score <=γ or >γ. Compute minimal value by series of null window searches.

- Start with score in a previous iteration, then go up or down

- Perform better than PVS/NegaScout by a factor of 10%

- PVS/NegaScout are still used in practice because of instability of MTD(f)'s behavior
Search Extensions, Reductions, and Selective Search

• Ideas: Search promising moves deeper, unpromising ones less deep

• Avoid “horizon effect”
  • E.g. extend search for check, piece capture in chess

• Shape the search tree

• Both exact and heuristic methods

• Try to perform safe form of pruning in recent approaches

• Look at some of most important approaches
Example of Search Extensions and Reductions

- Quiescence search
- Null move pruning
- Futility pruning
- Late move reduction
- ProbCut
- Realization probability search
- Singular extension
Quiescence Search

• Hard to evaluate chaotic, unstable positions at leaf nodes
  • E.g., King in check, hanging pieces
• Idea: evaluate only “stable” positions
• Replace static evaluation by a small “quiescence search”
• Evaluate leaf nodes (stable positions) generated by quiescence search
• Highly restricted move generation – just resolve instability
  • E.g., generate check, piece exchange, and pass in chess/shogi
Null Move Pruning (1 / 2)
[Beal, 1990][Donninger, 1993]

- Almost all searched paths contain at least one terrible move
- Idea: cut-off those subtrees quicker
- Null move: if we pass and can still get a search cut, then prune
Null Move Pruning (2 / 2)

• Assume $n$ is examined with window $(\alpha, \beta)$ with depth $d$
  • Pass and reduce depth to $d-R$ where $R$ is a tuned value (large when remaining depth is large)
  • Perform null window search to check if returned score $\geq \beta$ or not (from current player's viewpoint)
  • If score $\geq \beta$, perform cutoff – indication that opponent may have made a terrible move and $n$ is unlikely to be in PV line
  • Otherwise, perform normal search

• Scenarios where null move pruning shouldn't be applied
  • E.g., positions in check, chess endgames (avoid Zugzwang)
Futility Pruning and its Extension
[Schaeffer, 1986][Heinz, 1998]

- Idea: discard moves that are unlikely to become best
- Performed at nodes close to leaf nodes e.g. remaining depth = 1 or 2
- Assume n is examined with window \((\alpha, \beta)\) with depth \(d\)
  - Prepare evaluation function \(\text{eval0}(m)\) that roughly calculates the score for move \(m\) and margin \(F\) – use larger \(F\) for deeper search
  - If \(\text{eval0}(m) + F \leq \alpha\), prune \(m\) because \(m\) has almost no chance to be a good move
  - Otherwise, perform normal search
- Do not apply futility-pruning for tactical moves because they usually have high errors in \(\text{eval0}\)
Late Move Reduction (LMR)

- See http://chessprogramming.wikispaces.com/Late+Move+Reductions
- Similar to history pruning, history reductions, null window search for realization probability search
- Idea: in likely fail low nodes, reduce search depth of low-ranked moves
- Popular in some strong chess/shogi programs
- Assume n is examined with window (α, β)
  - Perform null window search with reduced depth to check if score <= α for move m ranked low in move ordering
  - If score <= α, cutoff, otherwise perform normal search
ProbCut [Buro 1995, 2000]

- Observation: in many games, with good evaluation, search outcomes are highly correlated between different depths
- Reduce search depth for moves that are probably bad
- Yields more time to search more promising moves deeper
- Assume $n$ is about to be examined with window $(\alpha, \beta)$
  - Perform shallower search for move $m$ and obtain score $sc$
  - Check if $a \times sc + b - \beta \geq \Phi^{-1}(p) \times \sigma$, which indicates the real score for move $m$ is $\geq \beta$ with probability $p$
  - Check analogously if real score for $m$ is $\leq \alpha$ with probability $p$
  - Up to two null window searches are performed
Search Performance of Pruning Techniques

C.f. Figure 5 in [Hoki et al, 2012]

**Fig. 5.** The search-depth dependency of the number of nodes searched in chess. Crafty is used as a base program of this experiment.
Realization Probability Search [Tsuruoka et al, 2002]

- One example of fractional search depth extensions and reductions
- Define move categories, assign a fractional depth to each category
- Set fractional depth by estimating probability that next move is in specific category from master game records
- Need to avoid horizon effect caused by moves with large fractional depth
  - Perform null window search to check if score $sc >$ current best score
  - Perform full window search with small fractional depth (i.e. deeper search) if $sc >$ current best score
Singular Extension
[Anantharaman et al, 1990]

• Observation: One move (singular move) that is much better than the others may have some pitfalls

• Idea: Extend the search for a singular move at (expected) PV and CUT nodes

• Idea can be extended to binary, trinary [Campbell et al, 2002]

• Whether a move is singular or not cannot be known beforehand

• Perform null window searches for non-singular moves with reduced search depths + lowered window values
Evaluation Functions

- Returns heuristic value that indicates probability of winning
- A lot of domain knowledge is added
  - E.g. piece values, material balance, mobility etc in chess
- Trade-off between knowledge and speed
- Most features are linear combination
  - \( \text{eval}(n) = W_1 \times F_1(n) + W_2 \times F_2(n) + \ldots + W_k \times F_k(n) \)
    - \( W_1, \ldots, W_k \) are parameters and \( F_1, \ldots, F_k \) are features
- Parameter tuning – by hand or machine learning
- This tutorial deals with one recent successful approach to tune parameters in shogi
- See references for other approaches e.g., [Buro, 1998]
Minimax Tree Optimization (MMTO) [Hoki and Kaneko, 2014]

- Earlier version known as “Bonanza method” [Hoki, 2006]
- Successful for tuning evaluation function with 40 million parameters in shogi
- All of strong computer shogi programs incorporate machine learning approaches influenced by this approach
- Assumption: grandmasters play good moves
- Idea: Prepare many game records of grandmasters and learn to increase the number of moves that match between alpha-beta and grandmasters
MMTO (Cont'd)

1. Find best $w$ to maximize $J_{MMTO}^P(w) = J(P,w) + J_C(w) + J_R(w)$

where $J(P,w) = \sum_{p \in P} \sum_{m \in M_p} T(s(p,d_p,w) - s(p,m,w))$

$T(x)$ : Sigmoid function

$s(p,m,w)$ : minimax value for move $m$ at position $p$ identified by alpha-beta (use score at PV leaf in practice)

$d_p$ : move played by grandmaster at position $p$

$M_p$ : set of legal moves except $d_p$ at position $p$

$J_C(w)$ : constraint term

$J_R(w)$ : $l_1$-regularization term

$P$ : Set of positions

2. Use grid-adjacent update $w_i(t+1) = w_i(t) - h \cdot \text{sgn}\left(\frac{\partial J_{MMTO}^P(w(t))}{\partial w_i}\right)$
Other Issues on Alpha-Beta in Practice

• In some games, specialized search is invoked by main alpha-beta (previous lecture)

• E.g., in shogi, main alpha-beta cannot often find long sequence to mate player even with search extensions

• Specialized search called tsume-shogi solver with limited time/node expansions is used to avoid loss that results from main alpha-beta failing to find mating sequence

• Tsume-shogi solver cannot always be invoked because of its high overhead

• Typical computer shogi programs invoke tsume-shogi solver only at important lines
  • E.g., PV line, move that improves $\alpha$ value of window ($\alpha, \beta$)
Parallel Alpha-Beta

• Known to be notoriously difficult to achieve reasonable parallel performance

• Parallel alpha-beta suffers from performance degradation caused by several types of overhead
  • Search overhead: extra nodes examined only by parallel alpha-beta
  • Synchronization overhead: idle time for other processors to finish work
  • Communication overhead: communication latency in the network
  • Load balance: metric on how evenly work is distributed
Young Brothers Wait Concept (YBWC) [Feldmann, 1993]

- Generalization to PVSplit [Marsland & Popowich, 1985] and many variants exist
- Observation: High-performance alpha-beta achieves good move ordering
  - First move to try has a high probability of causing cutoffs/narrowing windows at PV nodes
- Idea: recursively apply the rule that the “left-most” branch at a node must be examined before the others are examined
- Achieves reasonable parallelism with small search overhead
- Global synchronization point at each iteration – work starvation in the beginning and end of iterations
Issues in Distributed Memory Environments

- High-performance alpha-beta uses transposition tables
- Search space of many games are DAG or DCG
- Identical states can be reached via different paths
- Sequential alpha-beta effectively uses information saved in transposition table
- Shared-memory parallel alpha-beta can still share TT among threads
- How to effectively share TT in distributed memory environments?
- See approaches e.g. [Brockington & Schaeffer, 2000][Feldmann, 1993][Romein, 2001][Kishimoto & Schaeffer, 2002]
Partitioned Transposition Table [Feldmann, 1993]

- Each processor preserves part of TT disjointly
- Distribute work and use *work stealing* for load balance
- Ask corresponding processor for TT information
- Incur communication & synchronization overhead for TT accesses, and additional search overhead for DAG

![Diagram of Partitioned TT and DAG](image)

Duplicate search
TDSAB
[Kishimoto & Schaeffer, 2002]

- Apply Transposition-table driven scheduling (TDS) [Romein et al, 1999] to alpha-beta
- Can remove synchronization overhead to access TT and some search overhead for DAG
- See MCTS part as successful example of TDS
Massively Parallel Alpha-Beta in GPSShogi [Kaneko & Tanaka 2012,2013]

- Very recent method that might be less efficient but is much simpler than previous approaches
- Won against Miura (professional 8-dan player) with 679 computers (> 2700 cores, mostly iMac 2.5GHz)
- Uses one master and many slaves
  - Master manages a tree from root and generates work assigned to slaves
  - Slave independently examines states assigned by master
  - Master updates its tree when slave reports new scores
Master's Algorithm in GPSShogi

- Assign more slaves to promising subtrees
- Perform quick alpha-beta search to select k promising children (e.g., 1 sec)
- Repeat recursively until all slaves have work
- Effectively reuse master's tree when opponent's move matches predicted move [Himstedit 2012]
Comments on Alpha-Beta (1 / 2)

- Time: node evaluation, execute/undo moves, alpha-beta logic – low overhead
- Memory: depth-first search, need only path from root to current node – very low overhead
- Memory(2): can take advantage of extra use of transposition table
- Very good overall
Comments on Alpha-Beta (2 / 2)

• Evaluation function: must be reasonably accurate, trade-off between speed and accuracy

• Solving games/game positions
  • Fixed-depth search nature is a problem even with search extensions + fractional depth
  • Rules of repetition depends on rules, e.g. draw in chess, illegal in Go
  • Repetitions must be handled correctly
  • Practical “solutions” ignore history – leads to graph history interaction problem
  • Issues about repetitions are handled in the lectures in the afternoon
Conclusions

- Gave an overview of alpha-beta algorithms and enhancements
  - Alpha-beta variants
  - Search enhancements
  - Search extension and reductions
  - Evaluation function and machine learning
  - Parallel alpha-beta