Computing Science (CMPUT) 657

Algorithms for Combinatorial Games

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Winter 2022

More on Alpha-beta

Time and Memory Requirements of Alphabeta

- We Looked at best case and worst case number of nodes searched
- Runtime is dominated by number of nodes searched
- What are the costs per node, and overall?
- Let's look at both time and memory costs

Time Cost Per Node of Alphabeta Search

Simplest model: assume time per node (approximately) constant

Real costs:

- execute/undo moves during tree traversal
- generate (and usually sort) moves in each interior state
- check if game over (terminal node)
- Often: evaluate a heuristic
  - can be expensive, e.g. big neural net
- Bookkeeping for alphabeta logic - low overhead

Memory Cost of Alphabeta Search

Basic algorithm:

- depth-first search
- needs only path from root to current node
- a few numbers per recursion level
- Kept in function call stack
- very low overhead

Memory Cost of Alphabeta Search (2)

Alphabeta enhancements may use much more memory:

- Alphabeta can take advantage of extra memory
- Main example: use for transposition table
- Can control the size freely, very flexible
- Other enhancement using memory:
  - endgame tablebases
  - opening books
  - table-based evaluation heuristics

Dealing with Repetitions and Cycles

- Result of repetition depends on rules
- Examples: draw (chess), illegal (Go)
- Exact solution (inefficient): use full state including history
- Practical “Solution”: ignore history - leads to **graph history interaction** (GHI) problem
- Efficient exact GHI solution (Kishimoto and Müller)
  - Reuse proofs for similar positions
  - Prove during search that history does not matter in many cases
  - Used in the checkers proof and for Go Life and Death solver

Other Alpha-beta Based Algorithms

Other Alpha-beta Based Algorithms

- Idea: smaller windows cause more cutoffs
- *Null window* - equivalent to boolean search
- With good move ordering, value of first move will allow to cut all other moves
- 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

Reducing the Search Window

- Classical alphabeta starts with window (-INFINITY, +INFINITY)
- 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 windows” in chess: take score from last move, $\pm$ a pawn or so
- Gamble: can reduce search effort, but can fail
- Can optimize it using runtime statistics

Reducing the Search Window (2)

- Call alphabeta with initial values alpha > -INFINITY, beta < +INFINITY
- Three results:
- result *v* within window (alpha, beta): reliable, true minimax value
- $v \leq$ *alpha*: *fail low* - re-search with window (-INFINITY,v)
- $v \geq$ *beta*: *fail high*- re-search with window (v,+INFINITY)
- Make it work well: failsoft, use transposition table

Null Window Search

- Also called Minimal window search
- Alphabeta Search with beta = alpha + 1
- equivalent to boolean search with test ($v < beta$)

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.

PVS / NegaScout

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

NegaScout Code

Adapted from

https://www.chessprogramming.org/NegaScout

```
int NegaScout(GameState state, int alpha, int beta)
if (state.IsTerminal())
    return state.StaticallyEvaluate()
    \\ evaluate from root player's view
b = beta
foreach legal move m_i, i=1,2,... from state
    state.Execute(m_i)
    int value = -NegaScout(state, -b, -alpha)
    if (value > alpha && value < beta && i > 1) // re-sea
       value = -NegaScout (state, -beta, -alpha)
    if (value > alpha)
        alpha = value
    state.Undo()
    if (alpha >= beta)
        return alpha
    b = alpha + 1 // set up null window
return alpha
```

Alpha-beta Enhancements

Search Enhancements

- Basic alphabeta search is simple but limited
- To create high performance games program, need many enhancements
- General (game-independent, algorithm-independent) and specific
- Even many “general” ones work well only in some games
- Depends on many things: size, structure of search tree, availability of domain knowledge, speed vs quality tradeoff, parallel vs sequential,...
- Look at some of the most important ones in practice

Types of Enhancements

- Exact (guarantee minimax value) vs inexact (speculative, heuristic)
- Improve move ordering (reduces tree size)
- Improve search behavior
- Improve search space (pruning)

Iterative Deepening (ID)

- Series of depth-limited searches, $d = (0), 1, 2, 3, ...$
- Advantages:
  1. anytime algorithm - first iterations are very fast
  2. If b is big, small overhead - last search dominates
  3. With transposition table, store previous best move to improve move ordering
  4. In practice, often searches less than without ID
- some games/programs increase d in steps of 2 (e.g. odd/even fluctuations in evaluation, small branching factor)

ID 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 Tables

- Idea: store and reuse information about search
- Avoid searching same subtree twice
- Get best move information from earlier, shallower searches
- essential in DAG's where many paths to same node exist
- Help even in trees e.g. with iterative deepening

Transposition Table Content

- Hash code of state (usually not one-on-one)
- Evaluation
- Flags - exact value, upper bound, lower bound
- Search depth, possibly other engine settings
- Best move

Use of Transposition Tables in Search


```
int AlphaBeta(GameState state, int alpha, int beta, int depth)
bool foundHashEntry = tpTable.Lookup(state, hashEntry)
    if (foundHashEntry AND hashEntry.depth >= depth
        AND Valid(currentNodeType, hashEntry))
     return hashEntry.value
    if (state.IsTerminal() OR depth == 0)
        return state.StaticallyEvaluate()
    moves = LegalMoves()
    if (foundHashEntry AND HasLegalMove(hashEntry))
    MoveToFront(moves, hashEntry.bestMove)
    foreach m in moves
... rest of code as before...
    tpTable.store(...) just before returning value or bound
```

- Design choice: store/retrieve static evaluation or not?
- Depends on speed of evaluation, size of table

Example: Implementation of Hash Data in Fuego


```
/** Hash data used in class SgSearch. */
class SgSearchHashData
...
    unsigned m_depth : 12;
    unsigned m_isUpperBound : 1;
    unsigned m_isLowerBound : 1;
    unsigned m_isValid : 1;
    unsigned m_isExactValue : 1;
    signed m_value : 16;
    SgMove m_bestMove;
```

- 32 bit design (could do `m_bestMove : 32` as well on 64 bit)
- Use C bitfields to pack multiple data into one word
- Usually not a great idea, but justified here - save space, can make table larger

Store and Lookup in Table

- Size of table typically power of 2, 2^n, as big as memory permits
- Use first n bits of hash code to index into table
- Lookup: verify that whole code is the same first
- Store: need overwrite strategy - what if entry already filled?
- Popular: two-level table with 1. recent entries, 2. valuable entries
- Persistence: keep table between different searches, reuse between moves in a game. Effective with iterative deepening

Hash Collisions

- Same index in table, different full code: discard one
- Same hash code, different positions: **error**
- What is probability of error?
- Are astronomically small errors acceptable for your application?

Zobrist Hashing

- Fast, simple technique for computing hash codes
- Idea:
  1. Unique random hash code for each (square, value) pair - typically use 0 for one value (e.g. empty square)
  2. Code of overall position is XOR of all square's codes
  3. Update: single XOR for each changed square - use transition table CodeChange[square][oldValue][newValue]
  4. If code of oldValue is 0 (e.g. for empty), need only 2-d table.
  5. for Undo moves, XOR again -> restore previous code
  6. Typically, at least 64 bits are used (why?)

Hashing: Exact vs Heuristic

- Tradeoff: exact code for whole state may be too big, especially if history is needed
- Example: n cells, 3 states each, $\lceil n \log_2 3 \rceil \approx 1.58n$ bits
- Typical: 64 bit code, heuristic
- in Solver, can use non-exact hash code, but must then *verify* the proof tree (see df-pn and GHI discussions later)
- For Assignment, correctness is key. More important than speed.

Space for Square Board with 3 States

n	squares(n)	$squares(n) \times \log_2 3$	bits(n)
1	1	1.584963	2
2	4	6.339850	7
3	9	14.264663	15
4	16	25.359400	26
5	25	39.624063	40
6	36	57.058650	58
7	49	77.663163	78
8	64	101.437600	102
9	81	128.381963	129
10	100	158.496250	159
...			
13	169	267.858663	268
...			
19	361	572.171463	573

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

- Improve move ordering without needing game-specific knowledge (Schaeffer 1983, 1989)
- Give bonus for moves that lead to cutoff
- Prefer those moves at other places in the search
- Will see similar ideas later in MCTS - all-moves-as-first heuristic, RAVE

Search Extensions and Reductions, Selective Search

- Idea: search promising moves deeper, unpromising ones less deeply
- Shape the search tree
- Both exact and heuristic methods

Examples of Search Extensions and Reductions

- Null move
- ProbCut
- Fractional search extensions and reductions
- Quiescence search
- Late Move Reductions

Null Move

- Observation: almost all searched paths contain one or more terrible moves
- Idea: cut off those subtrees quicker
- Null move: if we pass and can still get a search cut, then prune

ProbCut

- Developed by Michael Buro 1994, 1995, 1997
- Observation: in many games, with good evaluation, search results are highly correlated between different depths
- Reduce search depth for moves that are “probably” bad
- Yields more time to search promising moves deeper
- More info: `https://www.chessprogramming.org/ProbCut`

Fractional Search Extensions and Reductions

- Idea: Extend search after *forcing, urgent* moves
- Reduce search depth by less than 1
- Example in Go: capture threats and responses count only $1/8$ towards depth
- Effect: lines with many such moves searched much deeper - e.g. A nominal depth 5 search can reach 40 ply.
- Use with care - size of search can explode
- Search Reduction: reduce depth by more than 1 for unpromising lines

Realization Probability Search and Fractional Search Depth Algorithms

- One way of systematically setting fractional search depth reductions
- Idea: define move categories, assign a fractional depth to each category (Levy et al 1989)
- Realization Probability Search (Tsuruoka et al 2002), Enhanced Realization Probability Search (Winands and Bjornsson 2007)
- Estimate probability that next move is in specific category by learning from master game records

Quiescence Search

- Hard to evaluate chaotic, unstable positions. Examples in chess: king in check, hanging pieces.
- Idea: evaluate only “stable” positions
- Replace static evaluation by a small *quiescence search*
- Highly restricted move generation - just resolve unstability

Late Move Reductions

- Source: `https://www.chessprogramming.org/Late_Move_Reductions`
- Late Move Reductions (LMR), similar: History Pruning, History Reductions
- Idea: in likely fail low nodes, reduce search depth of low-ranked moves
- Popular, used in many chess programs

Summary

- Discussed a large number of search enhancements
- Many are general and work for other search algorithms, not just games
- Many were first developed for games