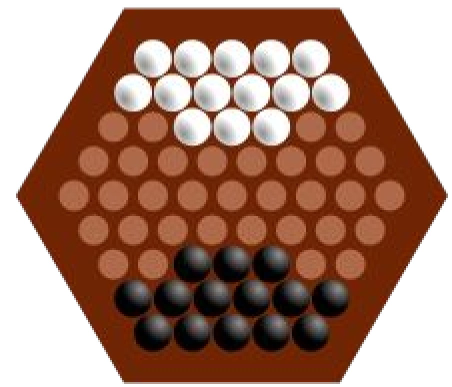
Optimized memoization for Abalone game tree search

CMSC 641 - Research Project

Deshpande, Amol
Peshave, Akshay
Prakash, Bharat

Abalone



- Perfect information, two player, zero sum game
- 61 spaces, 14 marbles each
- 2 kinds of moves, inline and broadside
- Goal is to push 6 opponent marbles off the board
- Pushing opponent's marbles

Abalone : Complexity

- State-space complexity: 10^{25}

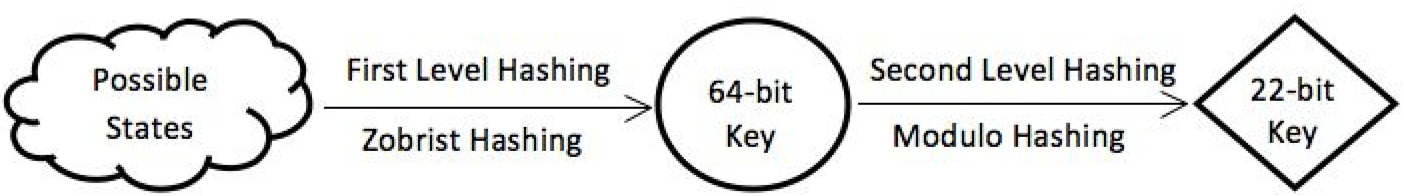
  This is similar to checkers which suggests that it is low.
- Game tree complexity: 10^{154}
- Average Game Length: 87 plies(human players)
- Average Branching Factor = 60

  This is substantially more than that of Chess.
- Average game tree size = 60^{87} states

Average repetition of a game state = 5×10^{127} (approx)

Current Approach

- Memoization - overlapping subproblems
- Perfect Information games
- Transposition tables
- Zobrist Hashing
- Second level hashing
- Flag to indicate that the value has been used



Zobrist Hashing

- 2D Table containing 64-bit random numbers:

Marble Color/Square Number	1	2	.....	61
(Black) 1	table[1][1]	table[1][2]	......	table[1][61]
(White) 2	table[2][1]	table[2][2]	.....	table[2][61]

- State representation using this table
  - XOR operation
- Easy calculation of new state after a transition
  - self-inversive property of XOR

Optimized Memoization Heuristics

- Goal: to increase the availability of states by storing more number of states in a transposition table
- Identifying the obsolete stored states
- Heuristics:
  - Number of Marbles
  - Frequency of hits of a memoized state

Augmented Transposition Table

Structure of transposition table row :

Structure of Head node

θ	Chain Length	chainHeadPtr $\rightarrow$
$\theta_{hitCount}$	$TS_{lastAccess}$	chainTailPtr $\rightarrow$

Structure Chain node

64-bit Zobrist Key + Utility Value	$MarbleCount_{black}$ $MarbleCount_{white}$	Hash Hit Count	Successor $\rightarrow$ Predecessor $\rightarrow$

Algorithm Key Features

- Increase in chain length ($0 \leq chainLength \leq \theta$) is governed by the chain length threshold ($3 \leq \theta \leq \theta max$).
- The chain is sorted in descending order of state hit counts in the transposition table chain.
- A new game-state is appended to the chain if either
  - $chainLength < \theta$ by incrementing chainLength or
  - $chainLength=\theta$ and $\theta_{hitCount}=3$ by incrementing both θ and chainLength
- A new game-state replaces the tail of the chain if either
  - $chainLength=\theta$ and $\theta_{hitCount} \leq 3$ or
  - $\theta = \theta_{max}$

Algorithm Key Features

- All previously stated logic is overridden and the decision to not increment θ is made if the most recent reference (read/edit) to any game-state in the chain is more than Δ_{plies} old.

- During search all game states in a chain disagreeing with the number of marbles (back and white) currently on the board are invalidated and deleted from the chain.

Time and Space

Search Time:

Best case - O(1)

Worst case - $O(\theta_{\mathtt{max}})$

Transposition table entries:

Increased by at most $(\theta_{max} * 2^{22})$

Probability of memoized state retention

$$\leq 0.4 + \boxed{0.6 * \left[\left(\frac{(\theta avg-1)}{\theta avg} * \frac{3}{4}\right) + \left(\frac{1}{4}\right)\right]}$$

where θ_{avg} is

- governed by iterative depth of the search and game-play in general
- constrained by alpha-beta pruning of the game tree

This bound is tight for frequently occurring states and very weak for infrequent states. This is ensured by dynamic variation in chain length and chain sorting based on frequency of occurrence of states in every run of iterative-deepening search.

Probability of memoized state retention (Weak Bound)

θ	Retention Probability	Transposition Table Size
3	0.85	12582912
4	0.8875	16777216
5	0.91	20971520
6	0.925	25165824
7	0.9357142857	29360128
8	0.94375	33554432
9	0.95	37748736
10	0.955	41943040
11	0.9590909091	46137344
12	0.9625	50331648
13	0.9653846154	54525952

Future Works

- Measuring average case retrieval times and I/O hits based on practical experiments.
- Heuristics to optimize memoization decisions based on pruning on the go.
- Computing tighter bounds on the probability of memoized state retention

References

- Papadopoulos, A.;Toumpas, K.; Chrysopoulos, A.; Mitkas, P.A.; , "Exploring optimization strategies in board game Abalone for Alpha-Beta search," Computational Intelligence and Games (CIG), 2012 IEEE Conference on , vol., no., pp.63-70, 11-14 Sept. 2012
- Albert Lindsey Zobrist, A New Hashing Method with Application for Game Playing, Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, (1969)
- Cuckoo Hashing for Undergraduates, 2006, R. Pagh, 2006
- Game Complexity [ONLINE] Available *http://en.wikipedia.org/wiki/Game_complexity*
- Implementing a Computer Player for Abalone using Alpha-Beta and Monte-Carlo Search, Pascal Chorus [ONLINE] Available *http://www.personeel.unimaas.nl/uiterwijk/Theses/MSc/Chorus_thesis.pdf*

Appendix : Algorithmic Sketch

SEARCH-HASH (searchHashKey, searchZobristKey, $ply_{current}$, numOfBlackMarbles, numOfWhiteMarbles)

```
headNode ← hashTable[searchHashKey]
node ← (headNode → chainHeadPtr)
while ( node not null ) do
    if node.zobristKey = searchZobristKey then do
        node.hashHitCount++
        headNode.ply_lastAccess = ply_current
        REPOSITION(node)
        return node.utilityValue
    else if (head->MarbleCount_black > numOfBlackMarbles OR
            head->MarbleCount_white > numOfWhiteMarbles) do
        node ← node.successor
        DELETE-NODE(node->predecessor, headNode)
    else do
        node.hashHitCount--
        node ← node.successor
return -1
```

```
DELETE-NODE(node, headNode)
    if(node = headNode → chainHeadPtr) then do
        (headNode → chainHeaderPtr) ← (node → successor)
        DEALLOCATE(node)
    else do
        node → predecessor → successor ← node → successor
        node → successor → predecessor ← node → predecessor
        DEALLOCATE(node)
```

```
REPOSITION(node)
    while ( true ) do
        if ( node → predecessor is null) then do
            return
        else if ((node → predecessor).hashHitCount) > node.hashHitCount then do
            return
        else
            (node → predecessor) → successor ←(node → successor)
            (node → successor) → predecessor ←(node → predecessor)
            (node → successor) ←  node → predecessor
            (node → predecessor) ← node → predecessor → predecessor
            (node → successor) → predecessor ← node

            return
```

INSERT-NODE (newNode, $ply_{current}$)

```
headNode ← hashTable[searchHashKey]
if (headNode.θ < headNode.chainLength) then do
    Add newNode at tail of chain
else if (headNode.θ = headNode.chainLength) then do
    switch RECALIBRATE-θ (headNode, θ_max, ply_current, Δ_plies)
        case 0: Add newNode at tail of chain
        case 1: Replace tail of chain with newNode
headNode.ply_lastAccess = ply_current
```

RECALIBRATE-θ (headNode, θ_{max}, $ply_{current}$, Δ_{plies})

```
if (headNode.θ) = θ_max then
    return 1 //means replace chain tail
if (headNode.θ_hitCount < 3) then do
    headNode.θ_hitCount++
    return 1 //means replace chain tail
else if (headNode.ply_lastAccess - ply_current < Δ_plies) then do
    headNode.θ++
    headNode.θ_hitCount ← 0
    return 0 //add new node at chain tail
else
    return 1 //means replace chain tail
```

```
DELETE-HEAD_NODE (headNode)
    predecessorNode ← ((headNode → tailPtr) → predecessor)
    DEALLOCATE(headNode → tailPtr)
    return predecessorNode
```