Analysis of iterative deepening minimax tree search using hashing based memoization for Abalone

Amol Deshpande, Akshay Peshave, Bharat Prakash

(amold1, peshave1, bhp1)@umbc.edu

Abstract

In the paper Exploring Optimization Strategies in Board Game Abalone for Alpha-beta Search the authors have devised a game state memorization strategy using Zobrist hashing and transposition tables for Abalone. Further they have stated speed-up in game tree search using this strategy.

In this paper we extend their approach by applying chaining for each transposition table entry. By applying various heuristics we dynamically rearrange the game states in each individual chain and also adapt the chain length. We provide evidence that quantifies the increased availability of memoized game-states having a comparatively higher probability of re-occuring in the near future ensured by our optimization. The time of access a game state hashed to a particular transposition table location has an upper bound of O(θmax) while the game-states memoized are at-most (θmax * 2^{22}) as opposed to 2^{22} in the original paper. This is the time-space tradeoff of our approach.

1. Introduction

This project includes the analysis of iterative deepening minimax tree search using hashing and memoization for the board game Abalone. When there are game trees where frequent repetition and overlap of sub-trees are expected, memoization helps us to improve performance. We have observed that the data structures used for memoization and hashing is a major factor in the analysis of the time and space bounds for this particular problem. We also explore several other techniques to come up with mathematical analysis and modelling to show time and space bounds for iterative deepening minimax tree search and game state memoization.

2. Keywords

Memoization, Hashing, Transposition Table, Iterative Deepening Search, Abalone, Board Games.

3. Previous Work

In their research paper “Exploring Optimization Strategies in Board Game Abalone for Alpha-Beta Search”, Papadopoulos A., Toumpas K. et. al. have used memoization coupled with iterative deepening search to achieve upto 25% speedup. The main aim was to to reduce the redundant searching of nodes. They have studied how an appropriate combination of different techniques is useful in achieving the same. To ensure viability of this approach,though, only a fraction of all the game states are memoized. Consequently, transposition table hash-collisions occur at a large rate.

4. Study of Abalone game-play, transposition tables, hashing techniques

4.1 Abalone game-play

Abalone is two player board game where players are represented by marbles of opposing colors. It is a perfect information zero sum game. The players have 14 marbles each. The board consists of 61 circular spaces which are arranged in a hexagon. There are 5 spaces on each side. Each player has 14 marbles of different colors. There can be different types of initial positions. Usually an initial position is

used the opposite marbles are placed on two opposite sides of the board.

Starting from the initial configuration, each player takes a turn. Players take turn pushing the marbles around the board in a column (that is, two or three marbles of the same color that are adjacent and arranged in a straight line). A move can be either in-line, where a marbles are moved parallel to the line of marbles. Or it can be moved broadside, where the move is not parallel to the line of marbles. On each turn, a player can move either a single marble or a column of marbles one space. The goal is to push 6 of the opposing player's marbles off the board.

The basic idea is that a line of marbles has the weight of the number of marbles in that line. The opposing player will need to push the column with a heavier line of marbles along that line. All marbles in a Column must move in the same direction. When a player's column faces a lesser number of opponent's marbles, the player has an advantage. As soon as the player has had six marbles pushed off the board, the game is over and the opponent wins.

Rules are very simple, but numerous strategic moving, pushing and defending possibilities make the game complex. Generally the strategy is to keep the marbles close to each other, and force the opponent's marble towards the edges. Creating a hexagon of marbles among the opponents' marbles, allows one's army to push or provide defense in all directions. Make defense as tight as possible which might provide attack power in the long run, if the opponent gets careless about its defense. Pushing the opponent's marbles has to be done carefully. One needs to take care that the geometry of his own marbles is not weakened in the process of pushing the other marbles.

Abalone has a state-space complexity of 10^{25}. The state-space complexity of a game is the number of legal game positions reachable from the initial position of the game. It has a game-tree complexity of 10^{154}. It is the total number of games that can be played or the total number of leaf nodes. The average game-length is 87 plies considering a human player. The average branching factor is 60. The average game tree size is 60^{87} states. The average repetition of a game state is 5×10^{127}.

4.2 Zobrist Hashing in Abalone:

Zobrist hashing is used in many simulated computer programs for board games, to generat keys representing all the states of the game uniquely. Abalone consists of a board that contains 61 squares, 18 black marbles and 18 white marbles. A 64 bit random number is generated for every combination of color and place that a marble can have on the board and stored into a 2D array of size 2X61.

```
constant indices
       white_marble := 1
       black_marble := 2

function init_zobrist():
#fill     a     table     of     random
numbers/bitstrings
   table := a 2-d array of size 2X61
      for i from 1 to 61:
         for j from 1 to 2:
                  table[j][i]=
              generateRandom64BitKey()
```

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]

4.3 Generating the State of the board:

64 bit random numbers from this table representing the color and square of each marble which is present on the board are used for generating the state of the game. XOR operation is performed on these 64 bit random numbers and the result is used for representing the state of the board uniquely.

```
function hash(board):
  h := 0
  for i from 1 to 61:
    if board[i] != empty:
      if colorOfMarble(board[i])== "BLACK"
            j:= 2
       else
            j:= 1
            h := h XOR table[j][i]
      return h
```

However, Abalone makes use of the property of XOR operation of being self-inversive for calculating the state of the board quickly. Due to this property, the new state of the game resulting after a single move of a game can be generated by performing just 2 XOR operations. If S1 and S2 represent the states of the game respectively

before and after the move M1 is made, which moves a white marble from square 43 to 42 then the relation between S2 can be generated from S1 in a following manner:

S2 = S1 [XOR] table[2][43] [XOR] table[2][42]

4.4 Transposition Table and Two Level Hashing in Abalone:

One of the appraoches that have been mentioned in the paper makes use of a transposition table for memoizing the calculated evaluation function values of the previously occured states for the future reference. This is done to avoid the recalculation of the evaluation function value of the same states.

The transposition table generally memoizes the evaluation function value V associated with a state S by storing the key-value pair containing the value V and key K representing the state S. However, even though the state of the board in Abalone can be uniquly represented by using a 64 bit random number with very less probability of a collision, transposition table needs to have 2^64 entries to store the values associated with each state. To tackle this hard memory requirement, two level hashing is done to reduce the size of the transposition table. This two level hasing techinque does a [mod 2^22] operation on each 64 bit key representing the state and makes use of only 22 least significant bits for representing any state. Transposition table only needs to have 4MegaEntries with this second level of hasing in place.

Collisions due to second level of hashing and the existing approach to tackle it:

Due to the fact that the second level hashing makes use of only 22 least significant bits of a key representing a particular state, there is a possibility of multiple states getting mapped to a particualar entry of the transposition table. The approach described in the paper allows transposition table to hold evaluation function value of at most one state. It also maintains a bit (0/1) along with this evaluation function to indiacate whether or not this value has been referred at least once after it was memoized. Whenever the collision occurs, depending on the value that this bit holds, it is decided whether or not to replace the existing state stored within that particular entry with the new one. Hash miss occurs whenever, while searching a particular state in transposition table, it is not found at the entry at which it has been mapped to after the second level of hashing.

5. Proposing a new approach

5.1 Reducing the number of hash miss counts:

To reduce the number of hash miss counts we propose a new approach of storing the evaluation function values of the states once the second level hashing is done. According to this approach instead of storing the evaluation function value of only one state at each entry of the transposition table, data of multiple states can be memoized. This is achieved by associating a doubly linked list with each entry of the transposition table. Each node of the dobly linked list represents data related to one of the 2^22 states that can be possibly mapped to the respective entry of transposition table. However, beacuase of the fact that not all 2^22 possible states can be stored in a linked list due to hard memory requirements, there has to be an upper limit on the number of states that can be stored in a doubly linked list. This upper limit is defined by the Threshold (**θ**) which may hold different value for every row of transposition table based on several factors such as number of hash hits received by that row. Moreover, as only some fixed number of states can be memoized at each row, there should be some basis for deciding which state should be deleted if the new state is hashed to the row of transposition table which has already reached its threshold.

5.2 Policy for controlling the value of Threshold(θ):

Initially all the rows of transposition table will have the same value of 3 for the threshold(**θ**). We propose a policy which based on the number of times the current threshold is hit and the timestamp of the last access controls the value of the threshold. Whenver a current threshold value is hit for the three times, the current threshold value is incremented by one.

5.3 Policy for deleting already memoized states:

We define two important heuristics for the policy to be used for deleting the memoized states when the threshod has been reached and the new node is to be inserted in a DLL.

5.4 Number of White and Black marbles:

The fact, that the number of marbles on the board of Abalone always goes on decreasing with the progress in time, forms the basis fot this heuristic. For example, if the number of black marbles on the board falls below 8 due to some move, then all memoized states having

more thatn 8 white marbles become obsolate and will no longer be required. Keys generated by Zobrist hashing do not contain any information about the number of mables which are present on the board. Hence, we need to associate the number of white and black marbles with each state.

5.5 Number of hash hits for a state:

A particular memoized state can be accessed mutiple times. We maintain a field which is associated with each memoized state and represents the number of times the state has been accessed after it was memoized. Everytime a state gets mapped to some row of a transposition table, this field associated with each state of that row gets changed. Hit counts of all the states belonging to such a row are decremented by one except for the state which is same as that of the state being searched; for this state the count is incremented.

5.6 Data Sructure: Transposition table with Doubly Linked List (DLL) at every row:

Structure of Head node

θ	Chain Length	chainHeadPtr →
$\theta_{hitCount}$	$PLY_{lastAccess}$	chainTailPtr →

Figure1: Structure of Head Node

θ : This field represent the upper limit of the length of the double linked list which starts at this node.

Chain Length: This represents the number of nodes currently present in the double linked list

$\theta_{hitCount}$: Represents the number of times this location(head node) is hit.

$PLY_{lastAccess}$: It represents the timestamp of the last time this key was accessed.

chainHeadPtr: Points to the first node of the linked list

chainTailPtr: Points to the last node of the linked list

Structure Chain node

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

Figure2: Structure of Chain Node

64-bit Zobrist Key: This represents the actual 64 bit key generated by Zobrist hashing initially.

Utility Value: This is the unitility value of the state calcuated by thr AI algorithm.

$MarbleCount_{black}$ & $MarbleCount_{white}$: These values are the number of black and white marbles present in that particular state.

Hash Hit Count: This field represents the number of times this key is hit after the second level hashing has happened.

Successor & Predecessor: These store the previous and next nodes in thedouble linked list.

6. Results

The search time for a memoized entry hashed to a transposition table location is bounded as below:

Best case: O(1) Worst case: $O(\theta_{max})$

The transposition table entries can increase to at most (θ_{max}* 2^{22}), which quantifies the space tradeoff of our optimization.

The probability of a hash collision in the existing scheme has been stated to be 0.6. Thus there is a uniform probability of 0.4 that a memoized state will be retained every time a transposition table hash collision occurs.

Given the high rate of game state repetition in a Abalone game tree this probability needed to be increased for game states which have a comparatively higher probability of re-occurring in the near future.

Our optimization exhibits this behavior. Heuristics such as the the Hash Hit Count, $Plies_{lastAccess}$ and marble counts enable this.

Let us consider uniform probability for all nodes present in the chain of a transposition table entry to reside at the tail of the chain. Also let $\theta_{avg\ be}$ the average threshold on the chain length across the transposition table. Then in the worst case, the probability of retaining any memoized game state on a hash collision is

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

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 when

our heuristics are used to rearrange the game states in a chain. 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.

θ	Probability	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

The above table shows that our bound converges to 1 as the chain length increases. The highlighted thresholds on the chain length are the optimal tradeoffs for this optimization.

7. Future Works

Presently the value of θ does not depend on the present state of the game. A good idea will be to vary this value based on some heuristic which is based on the present state of the game. This will ensure that memoized states are not deleted unnecessarily and in turn help in achieveing better access time. One more approach would be to introduce a second level of hashing at the head node. Here, the hash function can make use of the number of black and white marbles to distrubute the states in a better way in order to improve the access time.

Apart from these improvements, there is measuring average case retrieveal times and I/O hits based on practical expretiments. Another task is to compute tighter bounds on the probability of memoized state retention.

8. Conslusion

We have attempted to suggest a better model to decide on the future availability of memoized states by making use of some heuristics. Original paper work relied on a single bit to indicate whether or not a memoized state is frequently used. Instead of storing only one state in an entry of the transposition table, we modified the strucutre of the transposition table entry to accomodate a chain of states along with the metadata required for the heuristics. We have formulated heuristics, based on metadata about a memoized state, to indicate whether a memoized state is more likely to be used in the future than other states in the chain. Heuristics defined based on the number of white marbles and black marbles and the frequency of access of already memoized states, helped us to increase the availabiity of memoized states. $\boldsymbol{\theta_{max}}$ controls the maximum value of the threshold on the number of states that an entry of the transposition table can hold. Moreover, the total space requirement and the availability of states memoized can be controlled by adjusting the value of $\boldsymbol{\theta}_{max}$. This approach has increased availability of memoized states at the expense of increased space utilization which has a upper bound of (2^{22} * θ_{max})with the best case and worst case complexity of O(1) and O (θ_{max}) for searching the required state in a transposition table.

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

Pseudo code:

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


DELETE-NODE(node, headNode)
     if(node = headNode → chainHeadPtr)
          (headNode → chainHeaderPtr) ← (node → successor)
           DEALLOCATE(node)
     else
          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, θmax, PLYcurrent, Δthreshold)
               case 0: Add newNode at tail of chain
               case 1: Replace tail of chain with newNode
     headNode.PLYlastAccess = PLYcurrent
```

RECALIBRATE-θ (headNode, θ_{max}, $PLY_{current}$, $\Delta_{threshold}$)

```
     if (headNode.θ) = θmax then
          return 1 //means replace chain tail
     if (headNode.θhitCount < 3) then do
          headNode.θhitCount++
          return 1 //means replace chain tail
     else if (headNode.PLYlastAccess - PLYcurrent < Δthreshold) then do
          headNode.θ++
          headNode.θhitCount ← 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
```