Pruning Techniques of Minimax Algorithm on Adversarial 2048 Game

Yingyue Qiu
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024
qiuyingyue@cs.ucla.edu

Abstract

Adversarial 2048 game is a two-player sliding tile game which is adapted from the single-player 2048 puzzle. The Adversarial 2048 is different in that the newly generated tile in every turn will be regarded as the operation of the opponent player and the opponent player also try to play optimally instead of using random mechanism. To solve Adversarial 2048 game, alpha-beta minimax search algorithm with transposition table and symmetric detection is applied, and we propose Lower Bound Pruning and lower bound heuristic static value specially effective for pruning in this game. The game of 2×2 and 2×3 board can be solved for all initial states within half a minute per instance and this algorithm enables large amount of pruning on 3×3 board with average effective branching factor of no larger than 2.

Introduction

The 2048 puzzle, a single-player sliding block puzzle game which is played on a 4×4 grid board, has been a very popular game among people. Adversarial 2048 game is adapted from original 2048 puzzle to a two-player sliding-tile game. The game rules are as follows: Initially, the board contains two numbered tiles with value of 2 or 4. Every turn, the player (called tile-sliding player) slide the tiles to one of the four directions (up, right, left, down), so that the numbered tiles will slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid board. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. The resulting tile cannot merge with another tile again in the same move. After every move, a new tile will be placed in an empty cell on the board with a value of either 2 or 4, and we regard this as the operation of tile-placing player.

The score of the board starts at zero, and is incremented whenever two tiles merge, by the value of the new tile generated. The tile-sliding player is aimed to maximize the score of the board and the tile-placing player is aimed to minimize the score. This scoring system is interesting because in many games, their states are evaluated by the score obtained by the player during the game and the pruning techniques introduced here may be able to generalized to other games.

Our work is focused on solving smaller size of board as 4×4 board is unsolvable. Since it's a two player game, it's natural to apply alpha-beta minimax search algorithms. However, the size of the game tree even with alpha-beta pruning is too large to be searched thoroughly. For example, a 3×3 board can generate tile 1024 in the best case, but to generate a tile of 1024, at least 256 tiles of 4 needs to be added to the board by the tile-placing player and thus the depth of the search is at least 512. The branching factor of the game can be as large as 4 for tile-sliding player and the number of empty cells times 2 for the tile-placing player. The number of nodes generated in 3×2 board is already huge. The proposed pruning techniques can help prune a large amount of nodes. My algorithm with all pruning techniques combined can successfully solve 3×2 in less than a minute and it's hopeful to solve 3×3 board with faster implementation.

Related Work

Currently, most people have been working on achieve higher score on the single-player 2048 puzzle on 4×4 board. They typically apply minimax search with alpha beta pruning and search to the depth of only 3 to 8. A good heuristic with such depth is able to generate tiles of 2048 and even 4096 until the board is ended. In the single-player 2048 puzzle, the tile 2 is generated with 90% and a tile 4 is generated with only 10% and thus the Expectimax algorithm which accounts for this probability can achieve much higher score and may generate tile of 16384. However, no previous research has been done on solving the game.

As for general two player games, there are other complementary pruning techniques apart from alpha beta to accelerate the search. (Breuker 1998)propose the transposition table algorithm that is compatible with alpha beta pruning in a way that a lower bound or upper bound value will be recorded in the table to help pruning if the returned value is not exact. This is extremely helpful for pruning in this game.

(Sturtevant and Korf 2000) proposes a branch-and-bound pruning on multi-player game which require monotonic heuristic to produce a lower bound for heuristic and enables early pruning. In our two player game, we can also apply this pruning techniques. Although we only have lower

bound heuristic available, the lower bound pruning is effective combined with alpha beta pruning.

Problem Formulation

Scoring System

The score of the board starts at zero, and is incremented whenever two tiles combine, by the value of the new tile. This score g, has a property that it's monotonically non-decreasing.

State

A board is composed of the configuration of the grid as well as the indication of player's turn (tile-sliding player or tile-placing player). That is to say, two boards with the same configuration (same numbered tiles on the same position) but in different player's turn, should be considered different. A child board of the board is the resulting board after one operation is executed on the board.

A state consists of the board and a score g based on the scoring system. Since a tile 4 may be generated by tile-placing player with 0 point added or generated in a merge with 4 points added, two same boards may have different scores g. Let f_d be the backed-up value of the state after searching to depth d. $f_d = g + h_d$. h_d is the scored obtained during future slides. If all the leaf nodes examined when searching to depth d is an end state, $f_d = f^*$. $f^* = g + h^*$ Two same boards will have the same h_d and h^* regardless of its historical information g, since the score it can obtain in the future search will always be the same if both players play optimally.

Operation

An operation of tile-sliding player (MAX node), is to slide the tiles to one of the four directions(up, left, down, right). A valid operation will incur the merges or slides of the tiles and will increase the score if any merge happens. If the child board is the same as its parent board, it's an invalid operation. (The pseudo code below has already eliminated invalid operation.)

An operation of tile-placing player (MIN node) is to place a new tile of 2 or 4 on the empty cell, which will not change the score. The number of valid moves is the number of empty cells on the board times 2.

Initial State

The initial state contains a board with two numbered tiles of 2. The score of the initial state is 0 and it's tile-sliding player's turn. $f^*(initial\ state) = h(initial\ state)$.

End State

When it's tile-sliding player's turn and there is not any valid operations, that is, sliding to any direction won't change the board configuration, it's an end state. An end state always occurs on tile-sliding player's turn.$f^*(end\ state) = g(end\ state)$.

tile number X	2	4	8	16	32	64	128	256
lower bound $N_{lo}(X)$	0	0	8	32	96	256	640	1536
upper bound $N_{up}(X)$	0	4	16	48	128	320	768	1792

Table 1: Lower bound and upper bound of N(X)

Theoretical Analysis

Optimal End State Let N(X) be the accumulated points obtained from all previous steps for generating a tile X. Score obtained for generating a tile 4 can be 0 or 4 and thus score obtained for generating a tile X can vary. table 1 shows the lower bound and upper bound of N(X). Consider board of size $m \times n$. Theoretically, the end state with highest score we can achieve when both player is maximizing the score, is the board filled with all tiles from 4 to 2^{m+n+1}. The left board in Figure 1 is one optimal end state of 3×3 board. Only for analysis, we can also consider a modified setting that the tile-placing player can only place tile 2. The score of theoretical optimal end state is : $\sum_{i=1}^{m+n} N_{up}(2^i)$. The right one in Figure 1 is an optimal end state in this setting.

Optimal End State

4	8	16
128	64	32
256	512	1024

Optimal End State only 2

2	4	8
64	32	16
128	256	512

Figure 1: Optimal End State of 3×3 Board

Higher Score and Larger Tile The goal of the original game is not only to obtain a higher score but also to generate larger tiles. the points obtained to generate a tile of number 2X is strictly larger than two tiles of number X, and thus one tile of 2X plus any tile is preferred rather than two tiles of X. However, it's not always true that higher score indicates larger tiles For example, three tiles of number X value more than one tile of 2X plus two tiles of 2.

Pruning Techniques on Minimax Algorithm

Our goal is to calculate the minimax value for an initial state. Minimax algorithm is applied to compute this value and I have proposed the following pruning techniques. The pseudo code is showed in Algorithm 1.

Transposition Table with Alpha Beta Pruning

The transposition table is similar to (Breuker 1998). The main difference is that the key of the table is the board , and the value is the h of the state, because the same boards may have different g given different history but they will all have the same h, which is what we are concerned about.

Transposition table can help cache the minimax value of the board that is examined before, which is a way of duplicate detection. There are a lot of duplicate boards in 2048

Algorithm 1 Alpha beta minimax search algorithm with transposition table and lower bound pruning

procedure MINIMAXSEARCH($board, \alpha, \beta, Ttable, d$)
  if $board$ in *Ttable* **then**
    $Tvalue, Tflag, Tdepth \leftarrow Ttable[board]$
    if $Tdepth \geq d$ **then**
      if $Tflag = Exact$ **then**
        return $Tvalue + board.score$
      if $Tflag = Lowerbound$ **then**
        $\alpha \leftarrow max(\alpha, Tvalue + board.score)$
      if $Tflag = Upperbound$ **then**
        $\beta \leftarrow min(\beta, Tvalue + board.score)$
    else
      if $Tflag = Exact$ **then**
        $\alpha \leftarrow max(\alpha, Tvalue)$
    if $\beta \leq \alpha$ **then return** $Tvalue$

  $Hvalue \leftarrow LBHSV(board)$
  $\alpha \leftarrow max(\alpha, Hvalue)$
  if $\beta \leq \alpha$ **then return** Hvalue
  if $d \leq 0$ or $board$ is in end state **then return** $Hvalue$

  $alphaOrig \leftarrow \alpha$
  $betaOrig \leftarrow \beta$
  if $board.player\_turn =$ "$tile_sliding$" **then**
    $bestval \leftarrow +\infty$
    for childboard of board **do**
      $bestval \leftarrow max(bestval, MinimaxSearch(childboard, \alpha, \beta, Ttable, d-1))$
      $\alpha \leftarrow max(\alpha, bestval)$
  else
    $bestval \leftarrow -\infty$
    for childboard of board **do**
      $bestval \leftarrow min(bestval, MinimaxSearch(childboard, \alpha, \beta, Ttable, d-1))$
      $\beta \leftarrow min(\beta, bestval)$

  $storeval \leftarrow bestval - board.score$
  if $bestval \leq alphaOrig$ **then**
    $Ttable[bound] \leftarrow (storeval, Upperbound, d)$
  else if $bestval \geq betaOrig$ **then**
    $Ttable[bound] \leftarrow (storeval, Lowerbound, d)$
  else
    $Ttable[bound] \leftarrow (storeval, Exact, d)$
  return $bestval$

game and it's worth implementing. For example, sliding upward in all boards in Figure 2 will both lead to same child board.

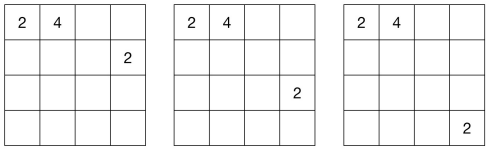

Figure 2: Example of Duplicate Board

For an entry in a transposition table, the key is the board including the indication of tile-sliding player (MAX node) or tile-placing player (MIN node), and the content of the key is a tuple consisting of:

1. flag: indicates whether the value is an exact value, a lower bound or a upper bound
2. value: the backed up value from the previous search minus the score of the current board g (it's $board.score$ in pseudo code) .
   When $bestval$ obtained from the child nodes is equal or smaller than the lower bound passed to the child nodes (we store it as $alphaOrig$ because α may be updated to $bestval$ after the child nodes are examined if current state is MAX node), it means that the child nodes may have returned a larger value than its exact value because the decedent nodes may have been pruned early because of the lower bound provided. Thus this value should be marked 'Upperbound'.
   Similarly, when $bestval$ obtained from the child board is equal or larger than the upper bound passed to the child (we store it as $betaOrig$ because β may be updated), it means that the child board may have returned a smaller value than its exact value. Thus this value should be marked 'Lowerbound'.
   Only if $bestval$ is between $betaOrig$ and $alphaOrig$, the value is 'Exact' which is f_{depth}, meaning that it has completed the search to depth d.
3. depth: contains the relative depth of the subtree searched. The depth indicates how deep a previously encountered board has been searched
4. move*: the best move returned from the previous search, which might be helpful for node ordering but not implemented.

When evaluating the board of a node, we should first check whether the board is in the table. If it's in the table and the search depth stored is equal or larger than the current search depth[1], it means that the stored board has been searched even deeper than the one we are going to evaluate and the stored value of the board should be more accurate and we are able to use that values. In this case, if the flag is 'Exact', we can directly return the value; if the the flag is 'Lowerbound'

[1] In fact, if the searching depth is smaller than d and the flag is 'Exact', it's also a lower bound, since searching deeper will obtain more scores in this game.

or 'Upperbound', we shall update our current lower/uppder bound α and β accordingly.

Symmetric Board Detection Since the board is square or rectangular, there are many symmetric boards. Combining rotations and mirrors can produce 8 symmetric boards for a square board and 4 for a rectangular board. There are two way of implementations. One is to store the values after examining the board for all its symmetric boards, which costs more time on writing table but constant time on reading. Another is to generate its symmetric boards one by one and check if it's in the table, and replace the board as the symmetric one if it's in the table, which will cost more time on reading table than writing.

Since the values of the table may be updated (if the value is only an upper bound/lower bound and doesn't cause pruning), there is no huge difference between the number of reads and writes. Thus, the latter implementation is preferred since it costs 1/8 space of the former one for square board. In this way, the child board of the board is also highly likely to be in the transposition table too.

There is a trade-off between the symetric detection because although it helps generate fewer nodes but spend more time in every node expansion. As the experiment result shows in the 'Experiment Result' section 3, it's worth implementing because the total running time is smaller.

For simplicity, the implementation of symmetric detection is not showed in the pseudo code.

Lower Bound Pruning

It's a branch-and-bound pruning on two player games mentioned in (Sturtevant and Korf 2000). Since we can only obtain the lower bound of the state in this game, I call it Lower Bound Pruning. Combined with alpha beta pruning, Lower Bound Pruning is very effective.

The minimax value of a state is f^*. Let f_{lo} be the lower bound of the backed-up value of the state. If we obtain f_{lo}, we can update the current lower bound $\alpha = max(\alpha, f_{lo})$. There are also three kinds of pruning, immediate pruning, shallow pruning and deep pruning corresponding to those in alpha beta pruning.

Notation for the example figure provided below: the square represents the MAX nodes and the circle represents MIN nodes; the number in the node is the backed-up value and the subtree searched is omitted for simplicity. The character in the node is its identification only.

1. Immediate Pruning
   For example in Figure 3, we first obtain the lower bound of MIN node A $f_{lo}(A) = 10$. After getting the backed up value of the first child, 10, and we know that other child nodes cannot provide a smaller value than 10, we can directly prune the rest of the child nodes. Such case is rare because it only occurs when the lower bound provides the exact minimax value, that is, $f_{lo} = f*$.
2. Shallow Pruning
   For example in Figure 4, after getting the backed-up value of the first child node 10, we update the upper bound to 10. $f_{lo}(A) = 8$ we cannot prune any node yet. When evaluating node B, we calculate its lower bound $f_{lo} = 15$ before expanding its child nodes and it has an upper bound 10 from its parent smaller than 15. Then, we can prune all child nodes of B now and return 15 for node B.

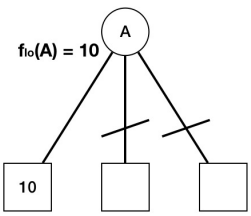

Figure 3: Example of Immediate Pruning

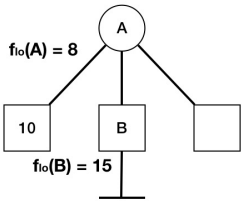

Figure 4: Example of Shallow Pruning

3. Deep Pruning
   The deep lower bound pruning is also valid in two player games. For example in 5 , the upper bound 10 of node A is passed to its child node B and its grand child node C. Since $f_{lo} = 8 < 10$, we cannot prune its child nodes yet before evaluating them. Then we find the lower bound of C is $15 > 10$ and we are able to prune its child nodes return value 15 for C. The rest of child nodes of node B is also pruned because its lower bound is updated to $15 > 10$.

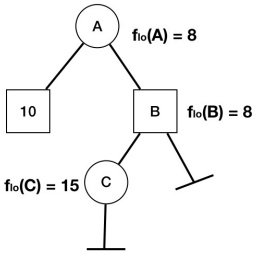

Figure 5: Example of Deep Pruning

Minimax Value f^* The exact minimax value of a state f^* (the backed up value after searching the entire minimax tree rooted at that node) is the sum of the current score g and the score it can force to obtain until the end state h^*. $f^* = g + h^*$. The lower bound f_{lo} should be no larger but as close to f^* as possible.

We can use the score g to be the lower bound of the minimax value, since $f_{lo} = g < f^*$. However, the current score g is a loose lower bound because it doesn't take into account

any potential score it may obtain in the future. We can provide a more accurate estimate of the minimax value. This value is a much tighter lower bound of the minimax value, called Lower Bound Heuristic Static Value (LBHSV). Below is the algorithm to obtain LBHSV.

Lower Bound Heuristic Static Value (LBHSV) Lower Bound Heuristic Static Value(LBHSV) is my designed algorithm to compute f_{lo}. Intuitively, it's the score that the board will definitely obtain without adding any more tiles to the current board. We will do a depth first search to merge all the tiles that can be definitely merged without newly added tiles. The algorithm to find such values is showed in the Algorithm 2

If the board is in tile-sliding player's turn, the score obtained and its child boards after one slide should be the same as the original game. And then the board is passed to the HSearch function to evaluate. When the board is in tile-placing player's turn, it's directly passed to HSearch function.

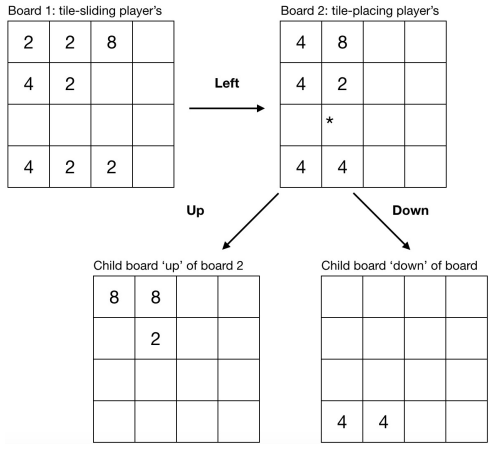

Figure 6: Example of heuristic search

HSearch function is a depth first search function that only simulate the slide operation and calculate the largest minimum score which can be obtained in the future slides. It will eliminate the effect of newly added tiles, that is, we won't execute the merge if any added tile on any position may cause the merge to fail.

In HSearch function, we only slide and merge tiles that is adjacent on the sliding direction because this is the score it can obtain definitely. For example in Figure 6, we know that moving upwards and downwards in the board 2 will both get 8 points. No matter what tile is placed in the empty position, this is the least amount of the score tile-sliding player can obtain on this board.

Then what is the resulting child board that can be used for subsequent searching after this move? If the move is sliding down, we will find the we have no idea where the tiles will be placed that has empty space below them. And if the move is sliding up, the positions of tiles with empty cells above them cannot be decided. Therefore, a natural way is to removes these tiles in the resulting child board (that is what $slides\&remove$ function does in the Algorithm 2). Therefore, sliding upward is more hopeful to obtain higher score because two 8s can be surely adjacent to be merged again. If the slide doesn't cause any merge, it's not a valid operation.

Will the added tiles that is not considered here cause the board to end earlier in the real game? No, because any merge will generate at least an empty position.

Note that the Search algorithm in the heuristic function has no effect on the search in real game tree. In the real game, it may be the case that tile 4 in generated on the empty position in the second column (where * is marked), and sliding downward may produce the same or more scores than upwards and 'up' should be selected as the best move. But, the value in real game tree is surely larger than the one we estimate in HSearch.

Algorithm 2 Algorithm for Computing LBHSV

```
1: procedure LBHSV(board)
2:     if board.player_turn =' tile_sliding' then
3:         bestval ← board.score
4:         for childboard of board do
5:             bestval ← max(bestval, HSearch(childboard))
           return bestval
       return HSearch(board)
6: procedure HSEARCH(board)
7:     bestval ← board.score
8:     for move in ["U","L","D","R"] do
9:         childboard ← board.slide&remove(move)
10:        if no merge occur in the childboard then
11:            continue
12:        bestval ← max(bestval, HSearch(childboard))
       return bestval
```

Heuristic Hash Table Since the computation complexity of LBHSV is not so trivial, we can compute and store it in the hash table at the beginning and thus the computation time becomes constant.

Heuristic Static Value

The Heuristic static value of a state is to best estimate the merit of the state if we stop searching its subtree . In this game, it's easy to know that the best static value should be the exact minimax value of the state f^*. If the state of leaf node is already an end state, the static value is just its score $f^* = g$. If the state of leaf node is not an end state, a heuristic static value f is given to estimate f^*.

There are many heuristic static value that people have designed to obtain better score in real game. For example, the state will have larger heuristic if tiles of larger number are kept at the corner, and tiles of the same or similar number are kept adjacent. But such heuristic is not guaranteed to produce the lower bound and purely depend on intuition. In my work, I directly use LBHSV as my heuristic static value, that is, $f = f_{lo}$.

Node Ordering

Fixed Ordering

From our intuition, we know that the tile-placing player usually have more benefit if it places 4 instead of 2 (it's not always true though). Therefore, we can generate its operation with 4 first instead of 2. This ordering is completed when generating valid moves and doesn't cost any time on sorting but can in effect bring a large amount of pruning. Since there is no intuition for the ordering of position and child node ordering for tile-sliding player, only a fixed default order is applied.

Ordering based on Heuristic Static Value

The node ordering can be based on the heuristic static value . If our heuristic static value is using LBHSV, this value is a helpful ordering metrics. It can help prune more nodes but there is a trade-off between the running time and the number of generated nodes, since the time for sorting is $O(nlogn)$ where n is the number of child nodes. Figure 7 shows the result of different ordering. Heuristic Ordering is the one with all nodes ordered based on LBHSV. Mixed Ordering is that MIN nodes apply fixed ordering on its child nodes but MAX nodes apply Heuristic Ordering. As can be seen from the result, Heuristic Ordering might generate more nodes than Mixed Ordering which means the fixed ordering on MIN node is more effective. Mixed Ordering generates the fewest nodes but not fastest.

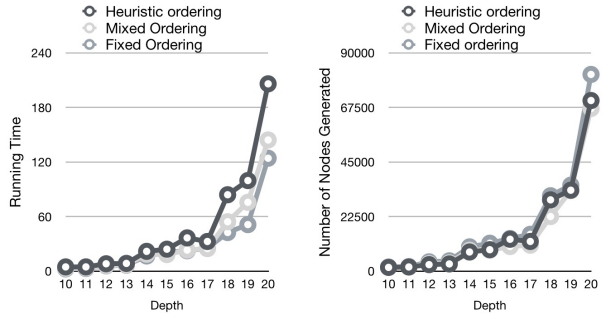

Figure 7: Different node ordering on 3×3 Board

Other Possible Ordering

These are possible node ordering that isn't experimented.

Ordering based on Iterative Deepening If we are using iterative deepening, the backed up value of last iteration is a good indication of ordering. However, iterative deepening is not applied here because the effective branching factor of small board is usually smaller than 2 for board size smaller than 4×4, thus we cannot use the value obtained from previous iteration.

Ordering based on Transposition Table In my implementation, the table only records the best value but not the best move, because the cost of computing moves for symmetric board is large. If the board is in the table and its next best move is recorded, the child node of executing the best move should be evaluated first.

Ordering based on Heuristic Hash Table Since our LBHSV is computed by simulating the moves of tile-sliding player, thus the child node ordering of tile-sliding player can reference the best move in LBHSV computation.

Experiments

Experiment Environment

- Language: Python
- Operating System: macOS 10.13.4
- Processor: 2.3 GHz Intel Core i5
- Memory: 16 GB 2133 MHz LPDDR3

Since Python is a little bit slow, and the experiment machine is only a laptop, the running time is much slower.

Experiment Result

The experiment result is showed in the following tables and figures which may contain these entries.
Time(s): the total running time of the algorithm (seconds)
# Nodes: the number of nodes generated during the search
End state: the selected end state associated with the backed-up value
Score: the minimax(backed-up) value of the initial state, also the score of selected end state

1. Result of 2×2 Board (compared with Bill)
   There are a total of six initial states after eliminating the symmetric boards, which is showed in Table 2 Since many end states have the same scores so our end states might be different with different node orderings. The largest tile generated is 8 for all initial states.
2. Result of 3×3 Board (compared with Bill)
   It's not solved so we only show the running time and the number of nodes generated when searching to depth d, see Table 3.

   My algorithm is able to search to depth 20 as showed in Table 4 with branching factor listed. It's surprising that the average branching factor is no larger than 2.
3. Comparison of Different Algorithms on 3×3 Board
   On the board of size 3×3, I have experimented and compared the running time and the number of nodes generated on algorithms that don't apply all pruning techniques. The result proves that all pruning techniques introduced above is useful for saving running time, see Figure 8.
   According to the analysis in section 'Node Ordering', the saving nodes generated by node ordering based on LBHSV cannot compensate for the time cost on sorting on 3×3 board, we all use fixed ordering in the following experiments.

   Our algorithm that use all pruning techniques, including alpha-beta, lower bound pruning, transposition table with symmetric detection, and LBHSV as heuristic is called AB&LB with Sym-Ttable with LBHSV. It generates the fewest nodes and costs least time among others.

Initial State	End State	End State (Bill)	Score	Score(Bill)	Time(s)	Time(s) (Bill)	# Nodes	# Nodes (Bill)
$\begin{bmatrix}2,2\\0,0\end{bmatrix}$	$\begin{bmatrix}2,4\\8,2\end{bmatrix}$	$\begin{bmatrix}4,2\\8,4\end{bmatrix}$	12	12	0.13030	0.27483	32	1261
$\begin{bmatrix}2,4\\0,0\end{bmatrix}$	$\begin{bmatrix}8,2\\2,4\end{bmatrix}$	$\begin{bmatrix}4,2\\2,8\end{bmatrix}$	8	8	0.07774	0.12017	15	518
$\begin{bmatrix}4,4\\0,0\end{bmatrix}$	$\begin{bmatrix}4,2\\8,4\end{bmatrix}$	$\begin{bmatrix}2,4\\8,2\end{bmatrix}$	8	8	0.05481	0.12661	16	592
$\begin{bmatrix}2,0\\0,2\end{bmatrix}$	$\begin{bmatrix}8,4\\4,2\end{bmatrix}$	$\begin{bmatrix}4,2\\8,4\end{bmatrix}$	12	12	0.09271	0.22837	34	1061
$\begin{bmatrix}2,0\\0,4\end{bmatrix}$	$\begin{bmatrix}2,4\\8,2\end{bmatrix}$	$\begin{bmatrix}4,2\\2,8\end{bmatrix}$	8	8	0.08518	0.11526	35	514
$\begin{bmatrix}4,0\\0,4\end{bmatrix}$	$\begin{bmatrix}8,4\\4,2\end{bmatrix}$	$\begin{bmatrix}2,4\\8,2\end{bmatrix}$	8	8	0.04706	0.09086	15	435

Table 2: Result of 2×2 board

depth	6	7	8	9	10
Time(s)	0.288	0.348	0.638	0.717	2.535
Time(s)(Bill)	0.496	2.190	7.445	30.809	98.063
#Nodes	181	233	385	489	1722
#Nodes(Bill)	3838	8287	53601	114576	681833

Table 3: Result of 3×3 board from depth 6-10

depth	11	12	13	14	15
Time(s)	2.930	5.856	6.525	16.745	20.596
#Nodes	1910	3858	4276	10031	11534
Branching	2.02	1.11	2.34	1.15	1.17
depth	16	17	18	19	20
Time(s)	22.454	25.118	42.514	51.217	124.22
#Nodes	13465	15205	31121	35465	81136
Branching	1.13	1.11	2.05	1.14	2.29

Table 4: Result of 3×3 board from depth 11-20

AB&LB with Sym-Ttable use only the score of the board as its heuristic static value and lower bound, that is, $f = f_{lo} = g$, The number of nodes generated is approximately two times as the one with LBHSV, but the running time is only a faction longer mainly because the constant time to retrieve value from a large heuristic hash table is still slower than getting the score g directly.

AB with Sym-Ttable with LBHSV doesn't apply lower bound pruning. Since the lower bound pruning is nearly cost-free, the running time and nodes generated without this pruning are both larger.

AB&LB with Ttable with LBHSV eliminates the symmetric detection only. The number of nodes generated is around as large as 4 times of the best but the time saved without symmetric detection in every node is huge and thus its running time ranked third.

AB&LB with Ttable is the worst since it not only eliminates symmetric detection and also LBHSV.

The result of brute force search or alpha beta pruning only without transposition table is surely too time-consuming to search to depth 20 so we omit the test result.

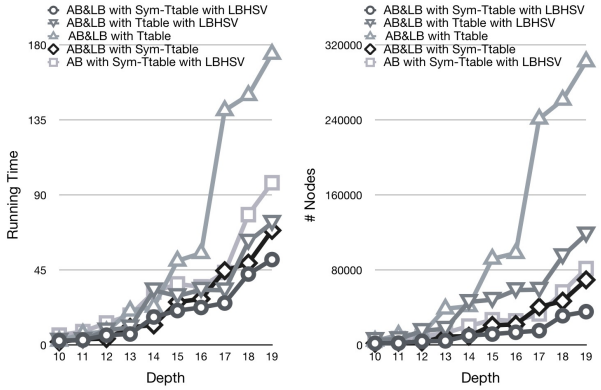

Figure 8: Running Time on 3×3 Board

Another benefit of using LBHSV is that the backed up value will be larger than the one without, which means that the tile-sliding player can force to obtain a larger score.

There is also an interesting phenomenon that if using LBHSV, the branching factor of nodes on MIN nodes is much larger than MAX nodes, but it's the contrary on the results without LBHSV. That might be because the Lower Bound Pruning is more effective on MAX nodes than on MIN nodes if using LBHSV, which means that the LBHSV provides better estimate on MAX nodes. Without LBHSV, all MAX nodes has the same g which is a poor estimate.

4. Result of 3×2 Board
These pruning techniques are impressive in that combined together the algorithm can solve the board of size 3×2 in half a minute. The result can be seen in Figure 9. Note that Heuristic Ordering is applied here which both generates much fewer nodes and cost less time than fixed ordering in this board size.

Only for analysis, I also list the result of a modified version of game with tile-placing player only able to place 2 instead of 2 or 4. It's interesting to see the result in this setting, because the tile-sliding player can always force

Initial states	Original (tile 2 or 4 can be placed)				Modified (only tile 2 can be placed)				Initial states	Original (tile 2 or 4 can be placed)				Modified (only tile 2 can be placed)			
	End state	Score	Time(s)	# nodes	End state	Score	Time(s)	# nodes		End state	Score	Time(s)	# nodes	End state	Score	Time(s)	# nodes
2 2	4 2 8 4 16 2	44	16.90089	8780	4 2 32 8 64 16	516	33.409404	17161	 2 4	2 4 8 2 2 16	44	18.4332910	10714	4 2 32 8 64 16	512	29.775097	17508
2 4	2 4 8 2 2 16	40	17.87148	9455	4 2 8 16 32 64	512	26.691735	14592	 2 2	4 16 2 8 4 2	44	21.7721230	11557	4 2 32 8 64 16	516	29.336394	17611
4 4	16 4 8 2 2 4	40	14.42942	7473	4 2 32 8 64 16	508	28.534458	16888	 2 4	2 4 8 2 2 16	40	17.5890381	9456	4 2 8 16 32 64	512	25.493635	14593
2 2	2 4 8 2 2 16	44	30.44145	16330	4 2 8 16 32 64	516	23.304803	14961	 4 4	2 4 8 2 2 16	40	17.3023509	9323	4 2 32 8 64 16	508	28.691652	16672
2 4	16 2 8 4 4 2	44	18.97470	10880	4 2 32 8 64 16	512	29.981361	17092	2 2	16 2 8 4 4 2	44	20.4333829	11763	4 2 32 8 64 16	516	28.335108	16983
 2 4	2 4 8 2 2 16	40	21.32651	9818	4 2 32 8 64 16	512	31.748425	17381	2 4	2 4 8 2 16 4	44	14.4117817	7925	4 2 32 8 64 16	512	32.612082	17307
4 4	4 2 2 8 16 2	40	19.37840	8898	4 2 32 8 64 16	508	26.732203	16765	4 4	8 2 16 4 4 2	40	13.6843466	7849	4 2 8 16 32 64	508	22.659274	14206
2 2	2 4 8 2 2 16	44	28.84475	12441	4 2 32 8 64 16	516	30.932729	17648	2 2	2 4 8 2 2 16	44	24.8633980	14435	4 2 32 8 64 16	516	32.070047	17658
2 4	2 4 8 2 2 16	44	20.83113	10696	4 2 32 8 64 16	512	28.218039	17375	2 4	2 4 8 2 16 4	44	20.3201949	11443	4 2 32 8 64 16	512	33.203101	17396
4 4	2 4 8 2 2 16	40	15.68412	8853	4 2 32 8 64 16	508	24.913190	16280	4 4	2 4 8 2 2 16	40	19.7372021	9335	4 2 32 8 64 16	508	27.727495	16248

Figure 9: Result of board of size 3×2

to achieve the theoretical optimal end state with highest score (while in the original setting the end state only has the largest tile of 16)[2]. It's also true on 2×2 board. Although it's not likely the case for larger board size but it's obvious that the game will become easier in this setting for tile-sliding player. This should be one of the reasons why the original game rules should have tile of 2 or 4 generated instead of only 2.

Conclusion

Multiple pruning techniques for minimax algorithm on Adversarial 2048 game are introduced in the paper, including alpha beta pruning and transposition table with symmetric detection, lower bound pruning and better heuristic value. The result is promising on solving board of size 2×2 and 3×2. The 3×2 board can be solved within twenty seconds while it can't be solved in twenty minutes without the introduced pruning techniques.

Though it hasn't solved 3×3, the branching factor seems not large and might be close to be solvable with faster implementation on other language and machine. Besides, there are other node ordering techniques that isn't implemented but might be helpful. Since the operation for two player is not symmetric, we can consider different ordering techniques on two players.

As for 4×4 board, its game tree is surely too deep to solve yet (over tens of thousands) and that's why the original game is set to this size to be playable.

An interesting observation is that if only tile 2 can be placed, the optimal end state can be achieved on 2×2 and 3×2 board.

[2]There can't be any end state with tile of larger than 16 but achieve the same score. At least 96 points needs to be obtained to generate a tile 32 . Thus tile 16 should be the maximum tile that can be generated on 3×2 board when both player play optimally. It's also true that tile 8 is the maximum tile that can be generated on 2×2 board.

Acknowledgements

I would like to thank professor Richard E. Korf for his teaching and guide that provide helpful inspiration for this project.

References

[Breuker 1998] Breuker, D. M. 1998. *Memory versus search in games*. Maastricht university.

[Sturtevant and Korf 2000] Sturtevant, N. R., and Korf, R. E. 2000. On pruning techniques for multi-player games. *AAAI/IAAI* 49:201–207.