

# Developing a Memory Efficient Algorithm for Playing $m, n, k$ Games

Nathaniel Hayes and Teig Loge  
Computer Science Department  
Simpson College  
Indianola, Iowa 50125

nate.hayes@my.simpson.edu, teig.loge@my.simpson.edu

## **Abstract**

An  $m, n, k$  game is a game where there is an  $m \times n$  grid where two players alternate turns trying to earn  $k$  pieces in a row. Currently, the best approach for never losing such a game is using a minimax algorithm. The minimax's downfall is the amount of memory a tree of all possible moves uses. Due to the large memory consumption, grid sizes greater than 4x4 or 5x5 cannot be used in real time with common computing power. Our approach to solving this issue is implementing a heuristic game tree that evaluates moves on a turn by turn basis. The algorithm assigns a helpfulness value to each available space, and a piece is placed in the space with the highest potential to produce a win.

# 1 Problem Description

An  $m, n, k$  game is a game where there is an  $m \times n$  grid where two players alternate turns trying to earn  $k$  pieces in a row. Tic-Tac-Toe is simply the 3, 3, 3 configuration of this game and Connect4 is a 7, 6, 4 configuration. Because these games are quite popular, an algorithm that could quickly and efficiently solve them could be very useful.

Currently, the best approach for never losing such a game is using a minimax algorithm. This algorithm uses a brute force method to construct all the possible moves, given any first move by the player, and has a time complexity of  $O(b^m)$ , where  $b$  is the branching factor, and  $m$  is the maximum depth of the tree [1]. The first problem the minimax algorithm runs into with larger grids is in runtime when calculating all possible moves. The construction of the tree takes much more time than would be desired. The algorithm also runs into a storage issue, but memory can be added to the computer running the algorithm to help combat this. The minimax has a memory complexity of  $O(bm)$ [1], and though the nodes in a  $3 \times 3$  grid's tree can easily be stored in most computers' memories, but when the size is increased beyond a  $4 \times 4$  grid, the number of nodes becomes too large for the computer to store them all.

## 2 A Heuristic Based Algorithm for Playing $m, n, k$ Games

We decided that we wanted to develop a versatile algorithm that did not run into the same problems that other popular methods employ, and included these characteristics:

- *The algorithm should be mutable.*  
It should be able to be applied to any configuration of an  $m, n, k$  game with little to no restructuring
- *The algorithm will never lose (it can draw).*
- *The algorithm should be fast on all reasonable grid sizes*
- *The algorithm should be memory friendly*

Figure 1 represents the program flow of our algorithm. As the program loops through every open space on the board, the algorithm first asks the question, "Should I move here?" If that move *results in a win*, then the piece is played and our algorithm wins the game, breaking out of the loop. If the move does not result in a win, the algorithm asks "What would happen, if the opponent went here?" This allows the computer to prevent the opponent from winning the game. The loop does not terminate immediately, as the *winning the game* branch does, because there might be a move further down the loop that would allow the computer to win. *It is better to win than to simply block the opponent.*

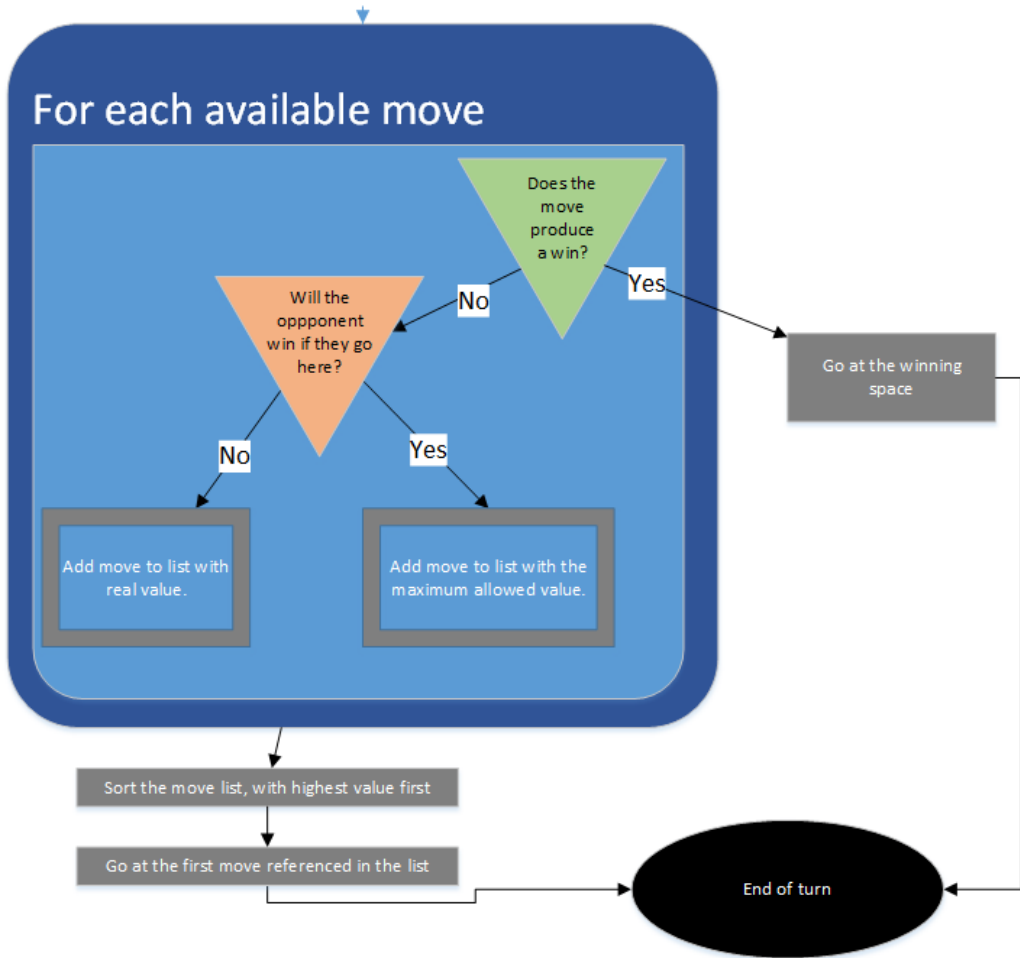


Figure 1: The game tree used to play an  $m, n, k$  game

During the course of a game, most of the time there is not an imminent win or loss at hand. It is in these times that a move should be evaluated for how helpful it would be in the future in terms of how likely it is to produce wins and deter losses. Figure 2, 3, and 4 illustrate how the helpfulness of a space is determined.

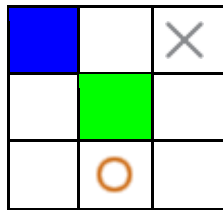


Figure 2: The algorithm is deciding where to play between the green and blue space as  $\bigcirc$ .






Currently, it is 's turn and the algorithm is deciding between the blue and the green space on the board. As the algorithm evaluates the blue space, it sees that there are two potential winning situations that are dependent on the space in question, and the space is then assigned a helpfulness value.



Figure 3: In board (a),  has the possibility of winning vertically, and in board (b),  has the possibility of winning diagonally.

The helpfulness value is a positive integer based on the number of possible wins. For this example, the algorithm assigns the blue space a helpfulness value of 2, because there are two potential winning situations that are dependent on the blue space. After assigning a value, the algorithm will move to the next space. When the green space is evaluated, there are three possible wins that can spawn from it, one of which includes an  in it already.

The green space is given a value of 4. Three of the points come from the three possible wins that the move produces, not counting the direction the opponent is blocking. It is also given one extra point, because one of those potential wins has an  in it already, thus allowing the helpfulness value to account for how close the player is to winning. Because the green space has a higher value, the algorithm picks the green space over the blue space.

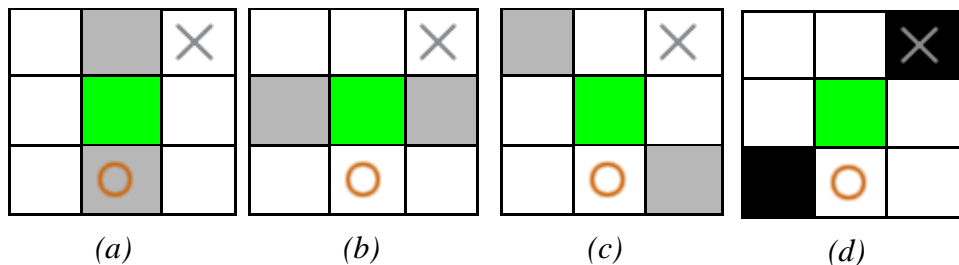



Figure 4: In (a-c),  could eventually produce a win vertically, horizontally, and negative diagonally respectively. Grid (d) shows that the positive diagonal is not counted as a potential win because the opponent is blocking it.

Below is the function that assigns helpfulness as described above.

```

function get_helpfulness
  Place piece where we are contemplating

  helpfulness <- count(# of our pieces in column) +
                  count(# of our pieces in row)
  if place is on positive diagonal
    helpfulness <- helpfulness + count(# of our pieces in
                                      positive diagonal)
    if place is on negative diagonal
      helpfulness <- helpfulness + count(# of our pieces in
                                      negative diagonal)

  Remove piece from where we are contemplating
  return helpfulness

function count
  return_value <- 0
  loop through the specified area of the grid
    if the piece found is the piece we are looking for
      return_value <- return_value + 1
    else
      return_value <- -1000000 // Cannot be positive
  if return_value > 0
    return return_value
  else
    return 0 // There are not negative helpfulness values

```

After testing the algorithm on a 4×4 grid, we found that this approach for determining helpfulness has a slight oversight. For instance, consider Figure 5.

0	○	○	×
×	×	○	○
×	○	×	○
0	×	0	0

Figure 5: None of the open spaces highlighted grey have a possibility of producing a win for ○, so all spaces have a helpfulness value of 0.

Since all open spaces have equal helpfulness value, a move is chosen at random, and this is where a mistake *can* be made. Let's say the computer places its piece in the following location:

	○	○	×
×	×	○	○
×	○	×	○
	×	○	

Figure 6: View of a game board with a possible move for ○, randomly selected from the list of possible moves.

The next best move for the opponent is in the top left corner, producing two ways to win for the opponent.

×	○	○	×
×	×	○	○
×	○	×	○
	×	○	

Figure 7: The opponent could go in the top left corner and secure a win.

We then came up with the idea to not only have the helpfulness level include the potential for a win, but also how well that move blocks the opponent from producing a win. Our revised algorithm will assign helpfulness values illustrated in Figure 8.

6	○	○	×
×	×	○	○
×	○	×	○
5	×	2	5

Figure 8: Helpfulness values assigned with the revised algorithm.

Because we are adding the value of helpfulness from each space from the perspective of the first player to the value of the opponent's perspective, we are also taking into account blocking the opponent's move.

*The new way to calculate helpfulness is:*

```
helpfulness <- get_helpfulness(computer) +  
                get_helpfulness(opponent)
```

This means that it would be better to go in a place that would help the player and hurt the opponent as opposed to going in a place that only benefits the player.

## **4 Experiments and Results**

We evaluated our method against minimax and analyzed the results. Because the minimax algorithm is supposedly the best way to solve a game of Tic-Tac-Toe, it stands to reason that it should never lose. This, however, does not mean that the minimax algorithm will always win.

The tests were performed as follows:

1. The minimax algorithm will go second to reduce time requirements.
2. The first moves<sup>1</sup> of both players will be randomly placed on a grid
3. Tally the results after 1000 games.

If we did not randomize anything, then both players would go in the same place every game, and there would be no variation from one game to the next. Even if one player is at a disadvantage because of an unfavorable random first move, then in later games the opponent should suffer from a similar circumstance, and the win/loss ratio should not be significantly affected by this.

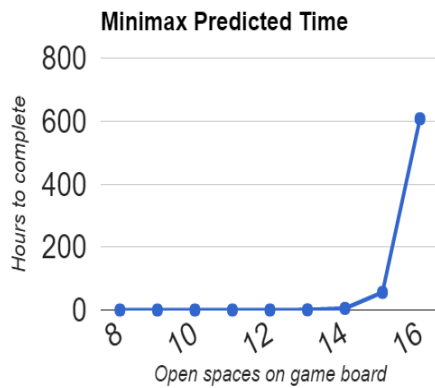
---

<sup>1</sup> The first move and first two moves of a 3×3 and 4×4 grid were randomized respectively.

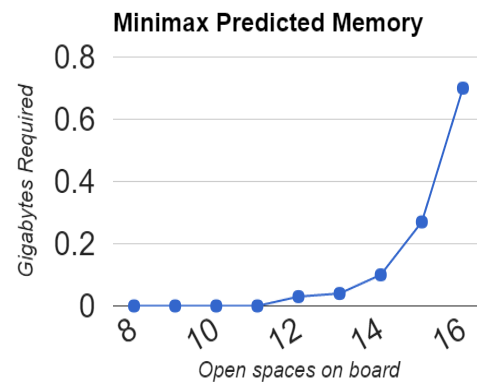
Table 1 illustrates how our algorithm compares to the minimax on a 4x4 grid with a varying number of open spaces.

Open spaces on board (4×4 grid) <sup>2</sup>	Minimax Time (h)	Algorithm's Time (h)	Minimax Memory (gb)	Algorithm's Memory (gb)
8	<b>0.00</b>	<0.001	<b>0.00</b>	<0.001
9	<b>0.00</b>	<0.001	<b>0.00</b>	<0.001
10	<b>0.00</b>	<0.001	<b>0.00</b>	<0.001
11	<b>0.00</b>	<0.001	<b>0.00</b>	<0.001
12	<b>0.04</b>	<0.001	<b>0.03</b>	<0.001
13	<b>0.55</b>	<0.001	<b>0.04</b>	<0.001
14	<b>5.16</b>	<0.001	<b>0.10</b>	<0.001
15	<b>55.97</b>	<0.001	<b>0.27</b>	<0.001
16	<b>607.25</b>	<0.001	<b>0.70</b>	<0.001

Table 1: The bold columns were measured and averaged over 10 runs. The gold cells were predicted from the measured data.



(a)



(b)

<sup>2</sup> The test are on a 4x4 grid So, for example, when the graph says “8 open spaces” what this means is that half of the spaces have been taken by the players pieces, and there are only eight spaces on the board in which a move may be played.



Figure 9: These two graphs represent the time and memory that would be required the closer a 4x4 grid gets to being empty, created from the data in Table 1.

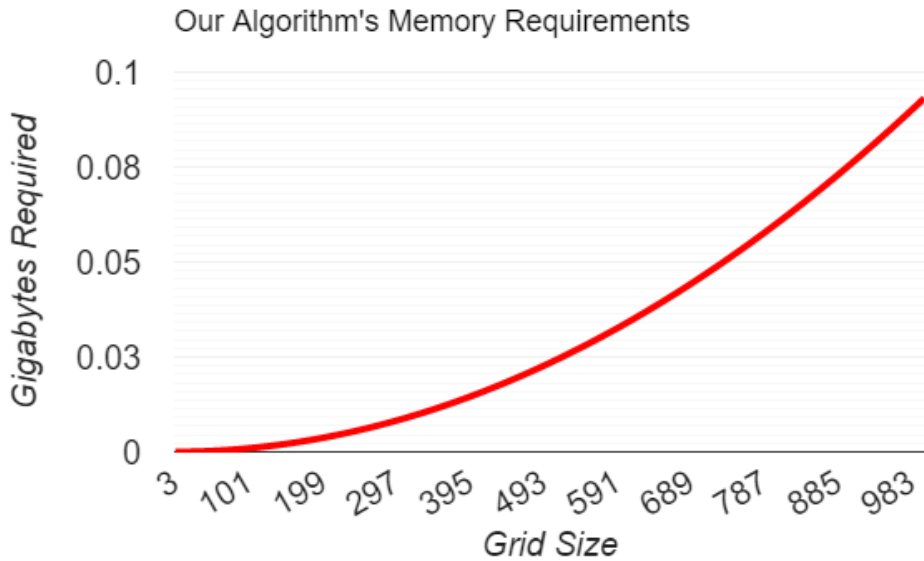


Figure 10: Memory requirements for our algorithm<sup>3</sup>. Note that for a 1000x1000 grid size the memory requirements are still less than one tenth of a gigabyte.

Because of the long run time of the minimax algorithm, we determined that the largest size we could test our algorithm against the minimax on would be a 4x4 grid. We would also need to randomize the first few moves for both players to reduce the time taken by the minimax for each move. We played 1000 games in this manner, and the results are below in Table 2.

	3x3		4x4	
	Our Algorithm	Minimax	Our Algorithm	Minimax
Wins	0	0	11	0
Losses	0	0	0	11
Draws	1000	1000	989	989

Table 2: Results of running 1000 games of 4x4 Tic-Tac-Toe

<sup>3</sup> The amount of memory required for our algorithm is a single integer for each space on the board to hold its helpfulness value. There for a grid size of  $n$ , it would take  $n*n$  bytes of memory, giving the formula for the curve  $f(n) = n^2$

The wins in the 4×4 games above were not expected. We predicted that all of the games would result in a draw as it was in the 3×3 games, or that both algorithms would have won an equal amount of games, because any handicap acquired in the first two turns by a player might also be acquired by the other player in a later game. Our team does not believe that the 11 wins scored by our algorithm are statistically significant however. We believe that a slight bias came from the minimax algorithm always going second to our algorithm in game play. Our team tried to alternate which player went first each game but found the time and memory restrictions too great for a large number of games if the minimax were to go first in half of them.

The next step in our research would be to move testing to larger sizes for  $m \times n$  grid sizes to see how the trend changes as the grid size increases.

## 5 Conclusion

In conclusion, our algorithm for Tic-Tac-Toe is a better alternative to the minimax algorithm for playing the game when considering memory and time consumption. Our algorithm does not have the two large problems minimax has, being a long runtime and large memory consumption and according to our test almost always results in a win. With the only memory consumption being a small list of possible moves, and a negligible time consumption, our algorithm can be implemented on any reasonable  $m, n, k$  game with only slight modification per game to determine how helpful a move would be. In the future we would like to see how our algorithm runs in other  $m, n, k$  games such as Connect4.

## Acknowledgements

We are thankful to Dr. Lydia Sinapova for providing helpful guidance throughout the course of this project. We are also thankful to Dr. Mark Brodie for helping prepare the manuscript for this paper.

## References

- [1] Russell, Stuart J. (Stuart Jonathan), and Peter Norvig (2003). *Artificial Intelligence a Modern Approach*. New Jersey. Prentice Hall.