An Investigation of Erickson’s Square Game using the Minimax Algorithm

Or… “Are we smarter than 50 lines of code and Ireland’s most powerful supercomputer?”

September 26, 2016 · Mathematics

Cormac Larkin FRAS[1], Nathaniel Grant, Conor McKeown

1 Armagh Observatory

Larkin C, Grant N, McKeown C. An Investigation of Erickson’s Square Game using the Minimax Algorithm: Or… “Are we smarter than 50 lines of code and Ireland’s most powerful supercomputer?”. Young Scientists Journal. 2016 Sep 26 [last modified: 2017 Jan 29]. Edition 1.

Abstract

Our project is a theoretical Computer Science and Mathematics project that aimed to explore the outcomes of optimal play scenarios in Erickson’s Square Game by applying the Minimax algorithm to the game.

Funding Statement

We received a bursary from the Ireland Chapter of the Parenteral Drug Association in order to compete at the BT Young Scientist and Technology Exhibition.

Introduction

Erickson’s Square Game is a problem that was originally proposed by the late Martin Erickson as follows:

“Two players alternately write Os (first player) and Xs (second player) in the unoccupied cells of an nxn grid. The first player (if any) to occupy four cells at the vertices of a square with horizontal and vertical sides is the winner. What is the outcome of the game given optimal play?”

We decided to try applying the Minimax algorithm with Alpha-Beta pruning to this problem in order to investigate it. The Minimax algorithm is as follows:

“A decision rule used in decision theory, game theory and statistics for minimising the possible loss for a worst case (maximum loss) scenario.”

Using the Minimax algorithm would allow us to play the game against a human opponent or another computer using a Python computer program. We adapted a pre-existing program used to play Tic-Tac-Toe that we sourced online for the basis of our program.

The Minimax algorithm rates every possible string of moves up to a specified depth, seeking the move string with the highest value whilst considering that the other player will also try to minimise their loss.

If the Minimax algorithm values for all move strings are calculated the computation is prohibitively long, so we used Alpha-Beta pruning to make the calculations more efficient.

The Alpha-Beta pruning ensures that moves are not calculated if they result in a non-optimal scenario after part of the search, thereby making the computation more efficient.

We ran our program on both our consumer-grade laptops and the most powerful High-Performance Computing (Supercomputer) facility in Ireland, the *Fionn* supercomputer from the Irish Centre for High-End Computing (ICHEC). They granted our investigation a Class C Project designation, which provided us with the necessary access to computational resources and support to make use of Fionn in our project.

This project won third prize in the BT Young Scientist and Technology Exhibition 2016 in the Senior Chemical, Physical and Mathematical Sciences group category

Literature Review

Prior to our investigation we checked to see if any previous work had been done on this topic and found two particularly relevant papers on the arXiv tool.

First Paper

The first paper, titled “Guaranteed successful strategies for a square achievement game on an n x n grid”[1] was written by Thomas Jenrich of Trier University in Germany. It deals with the formulation of the problem into a workable code and talks about guaranteed strategies for a small n.

One notable thing that we had identified before reading this paper is that only ¼ of all possibilities need to be computed as all other situations are a multiple of a 90° rotation of a previously computed permutation. That idea is discussed here and we decided to try to incorporate this into our work in order to optimise our computation so as to achieve results in the most efficient and timely manner.

Another thing that we gleaned from this paper is that, if played optimally, when n=3, where n is the number determining the size in units of the play area square, the game will be a draw. This gave us a reference point so that we could check if our program played optimally by comparing our generated results to this certain outcome.

The same goes for n=4, where Jenrich established that Player 1 will always win. This is also true for n=5. These valuable benchmarks aided us greatly when the time came to run our program. These were vital when we moved to n=>5 as we would have no way of knowing if our gameplay was necessarily optimal as nothing had been proven before.

The second part of Jenrich’s paper gave us the suggestion that coordinate Euclidean geometry would be a good way to represent our problem in code, since these coordinates can be easily converted to numerical values later on. This made computing the outcomes easier later on and also gives us an input method we are already quite familiar with from the school Mathematics curriculum.

We also found a formula which we managed to tweak to improve it. Jenrich’s paper has a formula for showing complete squares. However, we are interested in the playing process and therefore needed a formula for partially completed squares. We adapted Jenrich’s formula to get a formula that would give us a partially completed square.

Second Paper

Moving on from Jenrich’s paper, which was of great use to us in establishing a foundation for our investigation, we now looked at another arXiv paper, titled “Extremal binary matrices without constant 2-squares”,[2] which was written by Roland Bacher and Shalom Eliahou, of Université Joseph Fourier Grenoble and Université Côte d’Opale Littoral respectively.

This paper proved that for n=15 the end result prevents a draw result and that therefore no matter what moves are made by either player that a win results for one of the two players. This told us that in some cases we test, we may only have one type of outcome (win for either player or draw) regardless of moves made, demonstrable by computational means irrespective of optimal play or not.

Method

We began to look at how we would tackle the problem. Firstly, we decided to try to generate a Tic-Tac-Toe game based on the Minimax algorithm and base the rest of our work on that. Then we would try to adapt that program to play the Square Game.

Minimax Algorithm Basics

The Minimax algorithm attempts to (min)imise a loss in a (max)imum loss scenario. The following is an explanation of the Minimax algorithm:

the Minimax algorithm is a decision rule used in decision and game theory for minimizing the possible loss for a worst case (maximum loss) scenario. It was originally created for two player games such as ours, covering both the cases where players take alternate moves and those where they make simultaneous moves.

Minimax Algorithm Value

The Minimax algorithm value is the value used to make the decisions. This is a numerical representation of how likely a given string of moves will result in a maximum loss for a given player. Depending on the player, the lower or higher the value, the better that string of moves is to play.

In our case there are only three states; a win for P1, a win for P2 or a draw. This means our Minimax values are 1, -1 or 0.

It is important to note that these values will change after every move due to an opponent's moves. That is why the computation of the Minimax algorithm values need to be performed for every move for both players. The Minimax algorithm we use here will generate the best move at any given point in a game for both players, assuming that search depth isn't a barrier.

Each possible move has a Minimax algorithm value calculated for it by the program, accounting for every string of moves up to the specified search depth and the move with the lowest value is selected and played. In our program, we programmed it so that if two or more moves have the same value, then one move is selected at random.

Alpha-Beta Pruning

Calculating every possible string of moves exhaustively is one way to find an optimal play solution, but is very inefficient.vThe Alpha-Beta pruning process stops a string of moves from being calculated to conclusion if the move string stops being optimal. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. The end result is completely identical to the unaltered Minimax algorithm, but it allows for more efficient computation of the optimal move for a given player at a given point in the game.

Heuristics

Due to a limited search depth, the program finds it difficult to make opening moves. However, we found that the middle squares are the best places to start. To incorporate this finding into the program, we integrated a heuristic to prioritise middle squares if it cannot find a better solution. Due to the search depths, in practice the heuristic is always activated for the opening moves. A heuristic is a shortcut taken to speed up a calculation. Since we know the program will pick the middle squares, we save time using this process.

Adapted Program

Having the necessary fundamental understanding of the Minimax algorithm and the necessary software additions, we adapted a pre-existing code from cwoebker.com that played Tic-Tac-Toe optimally using a version of the Minimax algorithm.

This code worked as follows:

• The "random" library is imported

• Board size "n", # of squares "nxn" defined (for Tic Tac Toe n set as 3)

• Search depth (how many moves in advance are examined) is set to 9

• Winning combinations (every combination of winning moves) defined in a list

• Advantage combinations (every pair of squares that can lead to a win in one move)

• Win is defined as player occupying one of the winning combinations

• Draw is defined as all squares occupied with neither player having one winning combination

The best move for the PC is worked out as follows:

• The Minimax algorithm rates every move, seeking the move with the optimal value whilst considering that the other player will try to minimise their loss

• The Alpha-Beta pruning ensures that moves are not calculated if they result in a non-optimal scenario after part of the search, thereby making the computation more efficient

• The determine (a specific part of code) selects the optimal move based on the Minimax algorithm computation, ensuring the space is not already occupied by either player.

We now had a working Minimax algorithm playing Tic Tac Toe optimally. We needed to find a way to get this program to play the Square Game in as optimal a way as possible, which would require several changes. We began to identify and implement these changes.

Square Game Program

The changes we made to the Tic-Tac-Toe program were as follows:

• The winning combinations, which tell the program which moves equate to a win situation for any given player. Since the winning combinations for Tic-Tac-Toe are different to Erickson's Square Game, these needed to be changed to suit both the game and the different board sizes as different sizes will give different win possibilities. These would be the co-ordinate positions of all winning squares.

• The advantage combinations, which tell the program which moves are close to a win for any given player. Similar to the winning combinations, these needed to be changed for each board size and game. We ended up not using these as the

Minimax algorithm would work without needing these.

• Board size, which was changed from 3×3 to other sizes when necessary.

One thing we noted was when we set the program to play against itself, both players co-operated for a Player 1 victory. This meant that we had to play AI vs AI games across two different computers, manually inputting the computers' successive moves using the keyboards.

When we ran the game against a human it played without bugs, but not particularly well. We wanted to improve the code's ability to play the game. We decided to try to improve the program's intelligence by increasing the search depth.

Search Depth

The search depth is the amount of possible moves into the future that the program calculates. For example, a search depth of 5 calculates the Minimax algorithm value for every possible string of moves of length 5.

This would dramatically increase our computation times for larger boards, and so if we wanted to make sure our program played optimally before exploring more powerful computational systems we would need to reduce our board size to compensate for the increased computation. Increasing the search depth would make the program examine every possible combination of moves up to the specified number and any set of moves that are not optimal are not computed as they are removed by Alpha-Beta pruning, which reduces the time needed to compute.

We used search depths of between 3 and 5 on board sizes of between 3 and 6 to try and see if our code was playing better than it did previously. In order to do this, we set some games with a search depth of 0 so that the program would play randomly and if our program played better than random we took that as showing an improvement in play over pure chance. Our program performed much better than when playing at random and the program's ability increased with an increased search depth. The problem is that for any large board and/or a large search depth the time taken to compute is not realistic for one of our laptops. The time to play full game was on the order of thousands of years on some settings, making this method impractical for most scenarios, especially ones of interest.

At this point we realised that our own computational power was no longer sufficient for our investigation's needs and so we began to look at obtaining the use of a High-Performance Computing facility, otherwise known as a supercomputer.

Use of High-Performance Computing

We were awarded a Class C Project[3] by the Irish Centre for High-End Computing (ICHEC). We applied to them because they have had previous experience with the BTYS Exhibition and access to the facility could be interfaced from our school over the internet without needing to travel to the facility itself. We were introduced via email to Dr Simon Wong, a computational scientist working for ICHEC. He kindly agreed to provide us with the technical expertise and requisite permissions in order to access ICHEC's facilities.

Fionn Supercomputer

Dr Wong provided us with advice and support for our use of ICHEC's High-Performance Computing facility, named Fionn. We were awarded a Class C Discovery project classification by ICHEC for our investigation. As a result, we were provided with the following resources for our work:

• 60,000 core-hours of processing time

• 100 GB of storage

• 3 User accounts

Using the Putty computer program we were able to interface onto Fionn from our school with the help of our teacher, Dr Kerins.

One significant issue we encountered was the firewall used to protect ICHEC's network infrastructure from unauthorised usage. As we could not access a fixed IP address at school we thought we would not be in a position to use*Fionn*. Niall Wilson, the Infrastructure Manager of ICHEC, was very helpful to us in arranging the IP exceptions for us to access*Fionn* from our school.

Fionn uses a Linux-based operating system. It doesn't have a Graphical User Interface (GUI) like Windows and OSX has, instead using typed commands to complete functions. Therefore, we used command line inputs to transfer our program onto our folder from the PC in the Physics laboratory.

The command line system was very strange at the start but with Dr Kerins' guidance we managed to get our program up and running on Fionn. We were only able to use the login node on Fionn because we were unable to package our program in a suitable way.

In order to be able to utilise the full computational power of Fionn we would have needed to parallelise our program. Parallelisation is where a computational task is broken down into smaller problems solved simultaneously by multiple cores.

This is why High-Performance Computing is effective in solving otherwise impractical computations. We were not in a position to do this for two reasons.

Firstly, by the time we were satisfied with our program and arranged a relaxation of the firewall it was mid-December and we simply had no time remaining to parallelise our program. Secondly, for a programme to be parallelised it needs to be an executable file. This means that it can be run independently without any user inputs. By the time we had access to Fionn we were still unable to make the programme play effectively against itself and so we had to manually play one computer against another. This issue prevents the parallelisation of the program for now.

We instead investigated smaller board sizes where optimal gameplay was already established in order to see if our game played effectively for these established n and to ascertain whether High-Performance Computing facilities could be applied to our investigation sometime in the future as a proof-of-concept. Having had positive results from this, we increased the size of the board to sizes where optimal play results were unknown, but to compensate for this we had to lower the search depth, meaning that moves that were less optimal would ensue.

Results and Conclusions

Control of Central Squares

The first result we obtained from our investigations is that control of the central squares of the board is advantageous, irrespective of board size. In addition, the setting up of several possible winning move combinations is important for attempting to create a dilemma situation for an opponent From this, we can conclude that the opening moves of an optimal game will be to the middle squares and that the control of these would form part of an optimal gameplay strategy.

Use of Dilemmas in Gameplay

A dilemma, as discussed above, is where a player propagates a scenario where they have more than one winning move available to them, ensuring a victory. The applications of dilemmas were discussed at length earlier, and whilst we were not able to incorporate the dilemmas per se into our program, if the search depth is sufficient the Minimax algorithm will recognise an upcoming dilemma for an opponent and attempt to force it. Equally, it will try to propagate them for the opponent. As in chess, draughts and other deterministic two-player games, dilemmas are a vital part of any successful gameplay strategy and are surely a part of an optimal one. As successful creation of a dilemma for an opponent creates a certain win, we can conclude that they form a part of an optimal gameplay strategy.

Full/Near-Full Board at end of mutual Optimal Gameplay

In the course of our investigation, we discovered that the AI never won quickly when the opponent, either AI or human, played well. The board was always full or nearly full before a conclusion to the game emerged. This happened more when the search depth increased and the game played against itself better. From this, we believe that no shortcuts to a win in mutual optimal play exist and that optimal games need to be played almost, or completely, to exhaustion before a win for either player results.

Improvements and Further Work

Resolution to the AI Problem

We feel that the main problem we had in our investigation was our inability to make our program play against itself automatically. Whenever we tried to do this, both AI players co-operated for a Player 1 win. In order to make further progress towards an optimal strategy, we will need to fix this issue. We believe the issue is that the program is maximising Player 1's Minimax algorithm value with both players' moves.

Parallelisation of the Program

If the above problem can be dealt with, then parallelisation of the program will be possible. This means that the full computational power of a High-Performance Computing facility could be utilised to solve games for higher values of n and possibly find an optimal strategy in the future.

Further Investigation using High-Performance Computing

We think that if the above recommendations can be implemented, this investigation could be taken a step further and some definite strategies could be discovered if computed at full search depths by a High-Performance Computing facility.

We intend to continue our work in the future and implement these improvements.

Acknowledgements

We would like to gratefully acknowledge the contributions to our investigation from the following individuals: • Dr Tim Kerins, Coláiste an Spioraid Naoimh (Supervising Teacher) • Dr Anca Mustata, University College Cork (Project Supervisor) • Dr David Goulding, Tyndall National Institute (Project Supervisor) • Dr Simon Wong, Irish Centre for High-End Computing (Research Support) We would also like to gratefully acknowledge the allocation of a Class C research designation for our investigation from the Irish Centre for High-End Computing. ?

References

1. Jenrich, T. “Guaranteed successful strategies for a square achievement game on an n by n grid” 2012
REFERENCE LINK

2. Bacher, R., Eliahou, S. “Extremal binary matrices without constant 2-squares” 2010

3. Combinatorial analyses using the Min-Max algorithm.
REFERENCE LINK