Artificial Intelligence
-
Making Abstract Logic Games unbeatable.

Cecil Woebker
Worcester Academy
81 Providence Street
01604 Worcester
cecil.woebker@worcesteracademy.org

May 30, 2013

Abstract

Artificial Intelligence has long been a topic that fascinated many. When John McCarthy initially coined the term in 1955 it was a subject reserved for only the brightest minds. In recent years this has changed and Artificial Intelligence became more easily accessible. As it became more common throughout Computer Science the need for simplification increased. This paper is concerned with abstract logic games and the applications of Artificial Intelligence to it. This paper explores the simple games of Tic Tac Toe and Connect Four to create a common base. It uses common structures and unifies approaches that can be used with many different abstract games. A library, called Anigmo, was created as a companion for this paper. It sues Shannon Type-B search so that computers can play these games perfectly; it can be seen in the Appendix.

Contents

1 Thesis

In today's world computers can be hard to grasp and understand. It can be even harder to give them the necessary intelligence to solve a problem. Computers are already very powerful but they need a framework that can enable them to solve problems more quickly. Specifically they should be able to beat the human in almost any abstract logic game given a basic set of functions and rules. These could be combined into one library for easy reuse throughout the subject area.

2 Introduction

> Intelligence is the art of good guesswork.
>
> --- H.B. BARLOW, The Oxford Companion to the Mind

Artificial Intelligence has been at the heart of many scientist's studies for the past century. While many believed that by now we should have achieved significant progress toward a true artificial intelligence that could accurately mimic the human mind, this unfortunately has not occurred. The difficulties involved in developing such a system turned out to be greater than initially believed. Although there has been minor progress to create a true Artificial Intelligence, the subject is still very valuable for scientists around the world. Nowadays its more about approaching problems intelligently and making it possible for computers to do specialized and specific decisions effectively.

Abstract Logic Games are a great place to start learning about Artificial Intelligence due to the fact that there are so many different games ranging from very simple ones to very complex ones that might almost be impossible to fully comprehend. The Game Tic Tac Toe is easily calculated and predicted by a computer and therefore it is simple to create a perfect playing opponent. Searching the entire game tree provides a fast and perfect solution. Although both games are entirely different in Theory, approaches to intelligently solving them are very similar. These similarities can be used to save time and effort and unify technologies and approaches into one framework. Mainly, Algorithms used throughout both games are almost exactly the same. Merely some basic game functionality functions are different. Efficiency is very important while searching game trees due to there huge size, this daunting task of keeping computer

software fast enough is repeated throughout many different games. Centralizing the effort to providing a fast and efficient ways to look at search-tree like problems with a general game playing library can help create a fast solution that fits many different problems all over Computer Science. Such a system would be easy to expand to other games and past developments could help in solving those games and making them play perfectly.

3 Tic-Tac-Toe

3.1 Statistics

There are two important statistics worth looking at: the size of the game tree and the number of possible states.

One could assume that there are $9!$ ways to play a game of Tic Tac Toe. But after one player won the game will stop, so we have to discard those states. Instead of the full $9! = 362,880$ states we therefore only have 255,168 possible games (Schaefer, 2002).

- 131,184 (1st player)
- 77,904 (2nd player)
- 46,080 (tie)

One could assume that there are $9!$ ways to play a game of Tic Tac Toe. But after one player won the game will stop, so we have to discard those states. Instead of the full $9! = 362,880$ states we therefore only have 255,168 possible games (Schaefer, 2002). This might seem like a lot but for a Computer this is a joke. If we assume symmetries (discarding states already present by rotations or reflections) this changes to just about 26,830 possibilities (Schaefer, 2002).

When looking at the possible states of a game of Tic Tac Toe one can easily find an upper bound. There are 3 possible states for each cell (empty, X or O) and 9 cells: 3^9. By using the statistic features of the Anigmo library the number of possible game states was strongly solved to be 5478. This number does include reflections and other variations.

Finally there are 138 terminal positions after assuming symmetries (Schaefer, 2002).

- 91 (player 1)
- 44 (player 2)
- 3 (tie)

3.2 A little bit of Game Theory

Mathematics has a fascinating branch called Game Theory. Game Theory is defined as the study of "strategic decision making". It even plays a very important role in economics, since that, too, is often about making different decisions. Many have seen the movie A Beautiful Mind about the Life of the schizophrenic John Nash who ended up winning a Noble Prize regarding his work in the field of Game Theory.

Game Theory is very important for the solving of abstract logic games. Tic Tac Toe is something called a Zero Sum game, where all benefits one player receives are losses for another player; there is no action that represents gains for both players. Furthermore the game state of the Tic Tac Toe game is completely open to you, this is called a Perfect Information situation. You know everything about it, no hidden cards or anything like that. Therefore it is possible to play it perfectly if one puts enough thought into it since no luck or unknown variables are involved. It is worth noting here Playing perfectly doesn't mean you will win. In Tic Tac Toe for example this means there will always be a tie.

3.3 Perfect Play

To develop an AI for Tic Tac Toe it has to be able to decide what to do next. It has to know how to play perfectly. We don't want the computer to just play randomly after all, do we? At any given point there will be a move that is better or equally beneficial for a player than all the other possible moves. Finding this perfect move is the main goal of any game AI. For some games it can be rather easy to find a perfect move. In the case of Tic Tac Toe it can easily be done in one's head. It is even possible to put all the perfect moves for Tic Tac Toe on a single piece of paper.

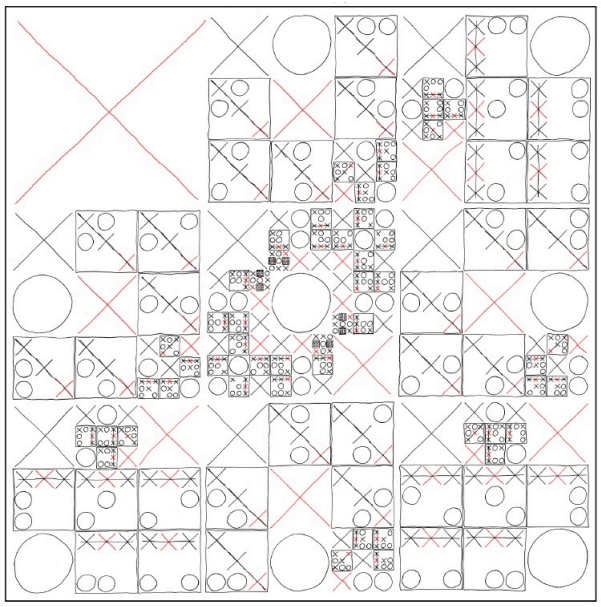

Figure 1: The famous Tic Tac Toe xkcd Comic. (#832)

3.4 Thinking like a Human

To develop an Artificial Intelligence you have to first understand what a human does; after all that is what we want the computer to do in the end. When every single one of us plays Tic Tac Toe it is easy to say what goes on in our minds: We look at the board and try to predict the future, attempting to find the move that will be best for us and worst for our opponent.

Essentially the computer does the exact same thing. It has one important advantage, though: it can see further into the future (given enough computing power). We could make the computer look far enough so it could see every possible move. This is how the computer can play perfectly and how many AI's related to abstract games (tic tac toe, connect four, chess, …) are implemented. If the games are too complex the computer might not be able to compute everything, but for a simple game like Tic Tac Toe there is no need to worry about that yet.

3.4.1 Minimax

The Minimax Algorithm, or a variant of it, is the crux of any computer-program. Creating an algorithm that is perfect and will never lose. This algorithm will walk through all the different game states and look into the future. We have to carefully craft it so that it always selects the best move. The Minimax algorithm follows a simple rule: Minimizing the possible loss in a

worst case scenario. Or in everyday language: making the best next move. This algorithm is especially helpful in Zero Sum games like Tic Tac Toe. Furthermore the algorithm basically recursively steps through all the states. Since the Minimax algorithm explores the whole game tree it is a Shannon Type-A algorithm (Shannon, 1950). Each of the traversed states is going to receive a value based on how good or bad this state is for the player. In case of Tic Tac Toe there aren't many possibilities so we can just iterate through all the possibility and see whether a chain of steps leads to a win, a tie or a loss for the player. All the different states are going to be the nodes in our game tree. The idea is pretty simple. The algorithm evaluates the current position and processes it step by step:

- Check if the node is completed. If yes, return the winner.
- Recursively call minimax with all possible moves.
- If it is the computer's turn return the highest result, otherwise the lowest.

And that is all there is to it. The Minimax algorithm is one of the most straightforward algorithms and sometimes it is implemented in two different functions. One of these would do the minimizing and the other one would do the maximizing.

Code in Appendix on Page: 31

3.4.2 The Wrapper

Both of the algorithms mentioned above will return a score associated with that position. What we need now is a wrapper function that calls our algorithm for different position and chooses the best one accordingly. This function is quite similar to the algorithm itself because its purpose is similar. It iterates through the different possible moves and keeps track of the best move. In the end it will return that move since it is an optimal move for the computer to play.

Code in Appendix on Page: 32

3.5 Result

The result of these basic ideas is an unbeatable tic tac toe program. After finishing all of the game logic and the Artificial Intelligence parts I decided to create a little website for the game.

It runs on the lightweight Python Web-FrameworkFlask and simply makes an AJAX request to the server to determine the next move. I made some minor additions to save moves that were already calculated so that the server does not crash when too many people play at the same time.

After posting the link at Hacker News, 2500 games were played by people all over the world. This should provide enough randomness. The final winning statistics (as of February 16th) were as follows:

- Computer: 763 (27.8%)
- Tie: 1980 (72.2%)
- Human: 0 (0.0%)

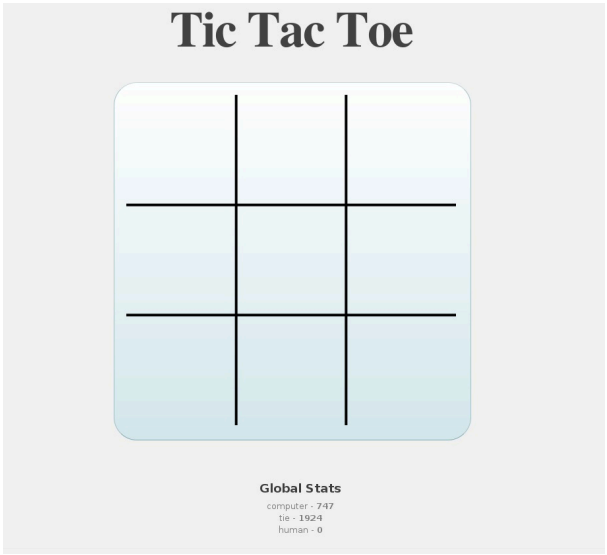

Figure 2: Screen (tic.anigmo.org)

4 Connect Four

4.1 Statistics

Connect Four is a game that was mathematically solved by Victor Allis in 1988 (V. Allis, 1988).

Just like before one could make a conservative estimate of $3^{42} = 1.094189891 * 10^{20}$ possible states, this, again, vastly overestimates the total amount of possible game states due to many invalid states being included. The true number of states turns out to be much lower, but still very large, at $4,531,985,219,092$ possible game states (Tromp, 1995a) (Edelkamp, 2008).

Due to the rather large amount of possible games that can be played in Connect Four (L. V. Allis, 1994) no true value of the game tree size could be found.

4.2 Advanced Artificial Intelligence

Due to the increased complexity of a game like Connect Four the methods used before are not going to be sufficient to provide a strong solution. Computers in today's world are simply not fast enough to search the entire game tree. Just like humans computers will have to stop looking ahead at one point and start evaluating how the situation looks. The function to do this is called a Heuristic Function. To further make this possible the search function needs to make smarter. It needs to be able to decide what part of the tree is more important than another. This is where Alpha-Beta Pruning comes in. While exploring these advancements it is important to see and to take note of how much of the code is redundant.

4.2.1 Heuristic Function

Heuristic Functions can assign relative values to certain game states compared to others. These values depend heavily on the game since they are entirely based on the rules of one game. For the game of Connect Four there are some basic rules that allow the development of a strong Heuristic Function (V. Allis, 1988). A very simplistic version of a Heuristic Function was implemented for this paper. Since the details of such function are heavily game dependent it is not worth describing them in detail for the purpose of creating a general library. For complex games they are needed, but have to be developed differently for each game.

4.2.2 Alpha-Beta Pruning

Alpha-Beta Pruning is a simple but significant expansion of the Minimax algorithm that can sometimes drastically improve performance. This might not be a big problem with Tic Tac Toe because of the small search-space but I will dive into it anyway since this is an easy example.

The basic idea behind both algorithms is very similar: The game states are being recursively iterated and the best state is found. The Advantage is that branches of the search tree can be eliminated and valuable computing power could be saved. The algorithm will focus on the better parts of the tree. Since the AlphaBeta algorithm explores only part of the game tree it is a Shannon Type-B algorithm (Shannon, 1950). If one move is worse than a previous one the algorithm is going to simply break. Therefore there are two variables that keep constant track of the upper and lower cutoff limits: Alpha and Beta.

Code in Appendix on Page: 35

4.3 Result

Although Connect 4 has a gigantic search space that cannot be searched by one computer efficiently enough to provide a result in a timely manner, the techniques described above resulted in a pretty effective program.

A website for this game was created as well. The link to this site was posted to Hacker News, which provided enough randomness to draw some basic conclusion on the quality of the program. The final winning statistics (as of February 16th) were as follows:

- Computer: 648 (94.7%)
- Tie: 2 (.3%)
- Human: 34 (5.0%)

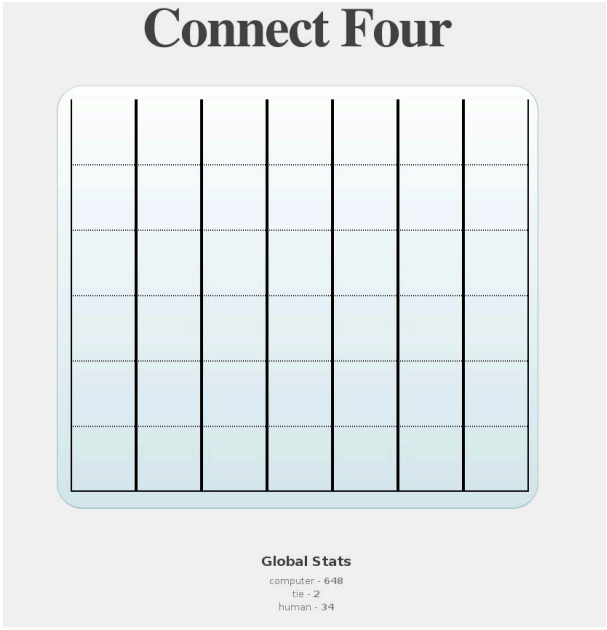

Figure 3: Screen (connect4.anigmo.org)

It is worth noting that these results are only partly helpful since many people most likely made simple mistakes that could have been easily avoided in a more focused environment. It would be worthwhile to test the AI against other programs in the future.

5 Efficiency

Hardware

The following Computer Hardware was used for all testing. Results can very when repeating tests performed in this section due to varying hardware and changing Python performance.

Model Name	MacBook Pro
Model Identifier	MacBookPro6,2
Processor Name	Intel Core i7
Processor Speed	2.66 GHz
Number of Processors	1
Total Number of Cores	2
L2 Cache (per Core)	256 KB
L3 Cache	4 MB
Memory (RAM)	8 GB
Processor Interconnect Speed	4.8 GT/s

Table 1: Testing Machine Hardware

Language Choice

Python was used for the majority of the code. Although it helped with creating a very dynamic library with minimal code reuse some sacrifices in regards to speed had to be made. The big function call overhead created a big problem since these function calls were needed in order to provide the possibility of simply turning some features on and off. Both Cython and PyPy were explored to improve speed. Both solutions offered substantial performance improvements. Rewriting some of the code in a more lower-level language like C will be hard since during the game tree search code from both the game and algorithm classes will be executed together. It would make more sense to completely rewrite the library with speed being the main goal. Throughout the process of creating this library there was often that confrontation of performance versus a general and portable approach. Due to the main goal of this paper is to explore the later of the two many sacrifices were made in regards to speed.

5.1 Selectivity Improvements

Game trees can become very large when more complex games are played. The larger the game tree the longer the search will take. Using more advanced algorithms can help decrease the size of the tree, but there are some further measures that can assist one in doing so.

5.1.1 Move Ordering

Move Ordering in itself does not give a performance boost. It nevertheless can drastically help in combination with the right algorithm. Algorithms like Minimax with the Alpha-Beta pruning enabled cut off nodes that are worse than ones already searched so that the full search space does not have to be searched. To increase the amount of cut-offs, a program can make the more interesting moves first and therefore decrease the size of the game tree that is searched.

Due to Connect Four's symmetric properties the positions closer to the center are often the better ones. After all, the first player always wants to start in the middle. Moves can be ordered from center to the borders. This of course is very simple and might often be wrong.

The kind of move ordering used here is very game specific therefore there is no general way to describe it. The easiest solution for this was to implement a standard move ordering function for each game class. History Heuristics or Killer Heuristics are move ordering patterns that have not been explored. It is worth noting that their general nature can be very much beneficial for an approach that will work throughout many games.

The more basic algorithms did not benefit from Move Ordering that much due to the high function call overhead from python. The more advanced algorithms were able to improve performance by a factor of 3.

Code in Appendix on Page: 35

Benchmark in Appendix on Page: 29

5.1.2 Quiescene Search

Although in itself Quiescene Search does not improve performance it can be used to increase the strength of the computer player dramatically. One could decrease the overall depth of the game tree search to balance this out and increase overall performance. Implementing this in a general

matter was trivial, Quiescence search is very similar to the algorithm itself with the difference that it only looks at specific moves. One can simply use an algorithm that already exist and apply a move filter that will fit to the desired properties. Changing the checking function of the algorithm base class to incorporate the Quiescene Search is enough to use it throughout any game. A general game library makes it easy to implement this search across the board. Unfortunately no benchmarking could be done since it is hard to judge the AI's strength.

5.2 Bitwise Game Logic

The Algorithms make use of game-specific functions repeatedly; therefore, they can only be as fast as these functions can be. The game setup and the different functions, therefore, are very important to be kept efficient so that no overhead is generated. A very popular method is to implement the game with bits instead of more high-level data structures. So called Bit Boards have many advantages. Bit operations are often a lot faster and allow for many tricks that can jump-start performance to unseen heights. Due to the fact that bit operations are often more complex in nature and do not allow for high-level features that a programming language will usually provide it can be hard to implement some of the game logic. Since much of the game code will be entirely different a complete new game class had to be created for the Bit-version of a game. Implementing Bit boards increased performance by a factor of 5 across all Algorithms and can likely be further improved. Unfortunately due to their game specific nature Bit boards do not have get an advantage from being implemented in a general game playing library.

Benchmark in Appendix on Page: 29

5.3 Obligatory Improvements

The improvements covered here can both lead to substantial speed improvements and are definitely necessary in any strong computer program. Furthermore they are very general and will work around all games that will be added in the future.

5.3.1 Hash Table

As explained by Victor Allis in his master thesis Hash Table can drastically improve performance due to the many transpositions of a specific game state; "one position can be reached in several different ways" (V. Allis, 1988). Now if one can store the positions already iterated performance can be improved significantly. When playing a game like Connect 4 there are multiple ways to end up at a given position, if the algorithm is not smart enough it will recalculate the same steps every time it reaches that position. Storing states already computed in a hash table can take care of that problem. Additionally it can help improve performance significantly when the same tree is computed again since all the hard computation has already been done before. Performance increases by a factor of 5 when simply using the hash table, but when the same search is performed ten times it's true power can be seen and speed improves by a factor of 20. A transposition is trivial to implement but some adjustments need to be made to the checking code of the algorithms. This can easily be done in a general game playing library like anigmo.

Code in Appendix on Page: 36

Benchmark in Appendix on Page: 30

5.3.2 Opening Books

Opening books are precomputed files that allow for perfect play. This means that if you follow the opening book you can have a very strong computer player right at the start of the game, often the most essential part. These opening books can be set up in different ways. The example book used here simply has a lit of all 8-ply game states and their corresponding outcomes if played perfectly. Therefore one just has to look 8 moves into the future for a desired outcome and can use that information to play perfectly. In a general game playing library one can simply extend the available algorithms to include the ability to search opening books since the basic search code will stay the same. Opening books will be different for each game and therefore would have to be created on a specific basis. There also are many opening books that are already available online and can be simply downloaded to be used further.

Implementing an opening book search led to increased performance; in case of an 8-ply

Connect Four database (Tromp, 1995b) the new code ended up being twice as fast.

Benchmark in Appendix on Page: 30

5.4 Complete Results

Summing up all of the previous results it can be seen that efficiency has been drastically increased when applying all the proposed changes to the basic code structure. It is note worthy that the opening book is not included in this analyzes since there were some complexities in the way it was structured that did not allow a fair comparison. Overall speed increased by a factor of 50 on a single run and by a factor of 300 over 10 iterations. Most of the code changes to achieve this were trivial since the library was set up in an extendable way right from the start

Benchmark in Appendix on Page: 31

5.5 Interface

Finally there needs to be some way for the program to know which of these performance improvements can be made use of for a given game. Some of the game implementations might not have all the features needed to make use of a transposition table or an opening book. Every game class therefore needs to define an interface that describes what features it provides. During execution a program can check whether the game selected and the features required can be used and executed together.

6 Basic Unification

After looking at both example games, Tic Tac Toe and Connect 4, many similarities can be seen. Much of the initial setup is different since the games work differently and have different rules. But the later Artificial Intelligence algorithms can be found in both examples. This common code should be able to be unified in one library which would simplify the process of creating computer players for other games. A simple solution would be to create Abstract Classes. There are three major occasions where this kind of unification could be helpful.

6.1 Game Logic

Many different games might share some common rules or there might just be different variants of from another. There are multiple instances were unification is helpful here:

- Changing the rules for a game
- Implementing different game dimensions or sizes
- Adding similar games that are variants of an original game, like the many different chess versions that are out there

Since only a few games were explored in this paper the full benefit of this could not be made use of. Since both games were based on 2-dimensional grids both Tic Tac Toe and Connect Four shared some minor structural code. Both games also share a similar Move Ordering function that relies on traversing through central positions first.

6.1.1 Standard

All games are going to be using the same algorithms therefore they all need to have some standard functions defined that serve the same purpose. This Abstract Class is going to be the backbone for all Games:

- init(): function executed at start of determination
- show(): show the game board

- moveOrdering(moves): sort the moves after pattern if needed
- available_moves(): possible moves
- over(): is the game over
- complete(): is the game complete/full
- winner(): is there a winner
- make_move(move, player): make a move
- remove_move(move, player): undo a move
- reset(): reset the game board
- heuristic(player): game specific heuristic function

6.2 Algorithm

There are basically 4 different algorithms that were explored: Minimax, Negamax, Alpha-Beta and Nega-Alpha-Beta. All these algorithms share certain features and have differences elsewhere. At their core they all do the same thing, they traverse a game tree and compute a positions corresponding value.

Negamax, unlike the Minimax algorithm covered earlier, makes use of a rather simple trick to improve performance: instead of computing the results specifically for the player that called the function it negates its return value every time its called, effectively removing the need for such checks. This leads to some minor adaptions that have to be made. The heuristic function will have to be called with the current player instead of the original Player since Negamax does not differentiate between the players during computation. Furthermore the final return value had to be negated to get the desired result.

Alpha-Beta Pruning on the other hand completely changes the way the game tree is traversed. Like explained earlier it skips entire parts of the tree that it finds being less valuable than others. Since there are two additional variables that have to be kept track of (Alpha and

Beta) the function calls had to be changed. Additionally the post-checking code had to be changed to include two variables instead of just one like in regular Minimax.

The final and most efficient algorithm explored in this paper, a Negamax Algorithm making use of Alpha-Beta Pruning, can simply inherit the majority of its functionality from the Alpha-Beta Pruning and Negamax algorithms. Only a small amount of post-checking code will have to be adjusted just like before. Due to the modular structure and extendibility of the Algorithm classes this final Algorithm could be successfully implemented in just a few minutes.

6.2.1 Standard

All algorithms are are split up into a number of simple functions which ensures extendibility. Just like before an Abstract Class is going to be the backbone and is going to define the following placeholder functions:

- init(): function executed at start of determination
- show(): show the game board
- moveOrdering(moves): sort the moves after pattern if needed
- available_moves(): possible moves
- over(): is the game over
- complete(): is the game complete/full
- winner(): is there a winner
- make_move(move, player): make a move
- remove_move(move, player): undo a move
- reset(): reset the game board
- heuristic(player): game specific heuristic function

6.3 Extra features

Many of the extra features that increased performance required simple changes in the Algorithm code to be available across multiple games. Due to the general and consistent way algorithms were defined in the previous section achieving such portability for the speed improvements is rather easy. For most of them simply some of the checking code has to be adjusted.

7 Implementation

A library called Anigmo was created that makes use of many of the ideas touched upon in this paper.

Code in Appendix on Page: 37

7.1 Purpose of Anigmo

In previous section a set of basic abstract classes was developed that allow code to be easily extendible and applicable to many games. Anigmo is an implementation of such framework that allows anyone to quickly dive into the world of game trees and learn about the problems that come along with solving games. It is a library that makes use of Shannon Type-B search to effectively estimate the perfect next move for a player. Having explored the issues and benefits of such library in regards to efficiency and time spent creating it the main goal of this paper has been achieved.

7.2 Extendable and General Nature

Anigmo was always meant to be a collection of everything that is needed to get started. From here on anyone can extend the documented classes to achieve their own desired results. Any such further extensions should adhere to the guidelines set forth in this paper so that the library stays general and extendible. The ultimate goal would be to create an open-source library that many people can use and work on together. This would effectively expand the purpose of this kind of library to include many different algorithms and many different games.

8 Future Challenges

Only a small amount of the topics were touched upon here and there are many more improvements that can be made. When looking at game trees with an general angle one always needs to keep a balance between integration and performance.

8.1 Future Developements

8.1.1 Expansion

So far only two games, Tic Tac Toe and Connect Four, were explored. Although they provided some insight on the possibility of a cross-game Artificial Intelligence there just weren't enough data points to look at the overall success. To make this assessment of game tree searches more meaningful the variety of games needs to be obviously increased. Some of the best candidates for such an expansion are:

- Checkers
- Othello/Reversi
- Chess

Additionally the games already existing can be extended in a simple way, let that be increasing the number of dimensions a game is played in or directly changing the rules a game is played with. For Tic Tac Toe one could try to implement it in 3 (van Cranenburgh, Smid, & van Someren, 2007) or even 4 dimensions (Patashnik, 1980). Although performance may change slightly those would be simpler to implement. Connect Four board size could be changed which also would increase the portability of this framework. It is very likely that this code will continue to work. The simple changes noted above often lead to whole new programs although a simple change in the setup of a Tic Tac Toe game class would be enough.

8.1.2 General Appraoch

There is also the possibility for a more general approach for a framework like this. One can move everything done here out of the context of games into the context of simple tree searching. This would allow for more applications let it be in simpler 1-player games or even other subject areas like Math. During the Spring I was approached by the math team coach of my school to help work on a problem for a school-wide math competition (Collaborative Problem Solving Contest) that was going on. The question asked for the number of possible paths of a given length between two points on a 2-dimensional grid system.

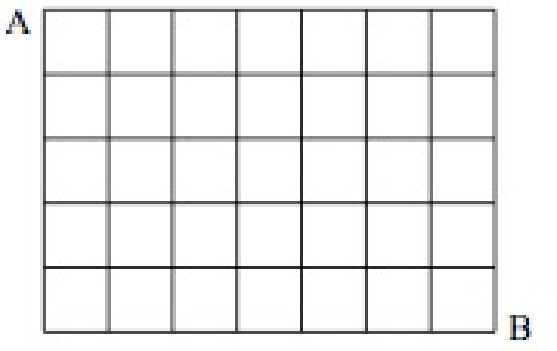

Figure 4: CPSC - Question 8 Grid

After having sat down and started the problem I quickly felt familiar with what I was doing, I solved the same problem already. This problem was all about the possible paths that are possible just like a Tic Tac Toe game was about the paths from having an empty to having a full board. This time instead of evaluating all the different states along the way the algorithms simply had to count them all. Eager to test the library on this I ran all 7 problems and reported the first results to my teacher within 2 hours of taking on the problem. A month or so later I would find out that all my responses to this question were indeed correct.

This opportunity gave rise to a new usage for this library by generalizing what it can do. I ended up starting to create a general Counter class that is able to dynamically solve problems like the one above.

8.1.3 Speed Considerations

During development the performance barrier was often hit which meant code needed to be optimized in order to continue receiving helpful results. This premature optimization worked but came with problems. In the future more of the performance heavy code needs to be rewritten in compiled programming languages like C or C++. In a performance depending environment that needs a solution like the one explained here it might be advisable to program the whole framework in such a language since doing so will save a lot of time later.

8.2 Suggestions for further Research

8.2.1 Game Description Language

There is a project that is aiming to create a general description language that is able to describe a game based on a relatively small set of logical rules (Thielscher, 2010). Provided such a Game Description File (Stanford, 2013) a computer should be able to figure out what to do and how to play the game. This could be very valuable since it removes the necessary of creating the Move Ordering and Heuristic Functions for each game playable by the computer. A Game Description file has the advantage of being language-independent and very general. Instead of developing Heuristic Functions for each game a computer should be able to deduce it from the general rules defined in this file.

8.2.2 Neural Networking

The most abstract and advanced possibility to consider in the future would be neural networking. This is a rather new development in Artificial Intelligence and therefore it comes with many difficulties. Since games differ in many ways separate neural networks would have to be created for each of them. This goes against the whole purpose of having a general library and therefore is not an optimal solution. A neural network is basically a simulation of neurons and synapsis from the brain. This allows for the computer to truly learn and train itself. Neural networks have been proven to be very effective at identifying objects in pictures and analyzing statistical data like that of stock markets. They can efficiently deal with a large amount of inputs and

return an answer in a matter of seconds. Training a strong neural network however can take a long time and can be hard depending on a problem. The structure of a neural network has to be tailored to every specific problem and there are many problems that can occur when doing so. Although they could help these specific requirements

9 Conclusion

9.1 Evaluation of the Project

Artificial Intelligence for abstract board games has been explored and algorithms and technologies have been evaluated to be used. These algorithms are similar to the way real humans think about games and decide upon their future moves.

The finished library is able to play any 2-player Zero Sum Game provided a defining class. This proves that general game playing is feasible and offers many benefits across the board for developing computer players.

Over 4552 lines of code were written for the Anigmo library. When looking at additions and deletions over 9578 lines have been coded. This does not include any of the code written for the websites or any other helper utilities.

Benchmarking the code revealed that some basic efficiency improvement led to an overall performance increase by a factor of 300. This improvement can be made use of by any game added to the library right away.

The goal of this paper, developing a library that is able to play any two-player Zero Sum game given a basic class, was achieved and a library, called Anigmo has been created.

9.2 Results of General Game Playing

When humans play games their mind tend to work on it in the same way; they look ahead and try to figure out the best outcome. Just like humans a computer will basically work the same way throughout different games which is why creating a general library makes perfect sense.

Developing a general library comes with many advantages: code can be reused throughout the project and therefore one is able to quickly develop new features and extend the project. For example when adding speed optimizations these were available for all games right away. When developing these algorithms and games with portability in mind constantly one can create a strong base that is easily extendible.

It can be seen that a framework can indeed further the knowledge about Artificial Intelligence. It faces the same problems a human would. Some games are just too different after all

too be unified and also some work effort can be saved there are just too many differences to unify them all at once.

9.3 Applying the Results for Playing

Using a library like Anigmo for playing these games, let it be for fun or professionally, comes with some issues. Much of the optimizations that is possible can be game-specific and creating a general library comes with some overhead. In the language used here, Python, there is an especially big function call overhead that leads to slow execution when many functions are called. In this recursive environment many million functions are being called which puts the library at a disadvantage compared to other solutions. To make use of these results in a face paced environment one would have to apply them in a different, more low-level language like C that does not face the same issues. Anigmo itself is easy to use and an example can be found in the Appendix (pg. 36). It is easy to create a game and play against a computer opponent of any of the included games.

References

Allis, L. V. (1994). *Searching for solutions in games and artificial intelligence*. Unpublished doctoral dissertation, Rijksuniversiteit Limburg te Maastricht.

Allis, V. (1988). *A knowledge-based approach of connect-four, the game is solved: White wins*. Unpublished master's thesis, Vrije Universiteit, Department of Mathematics and Computer Science, Amsterdam, The Netherlands.

Edelkamp, S. (2008). *Symbolic classification of general two-player games** (Tech. Rep.). Otto-Hahn-Str. 14, D-44227 Dortmund, Germany: Technische Universität Dortmund, Fakultät für Informatik.

Hochmuth, G. (n.d.). *On the genetic evolution of a perfect tic-tac-toe strategy*. Unpublished master's thesis, Stanford University, Stanford, CA 94305.

Patashnik, O. (1980). Qubic: 4× 4× 4 tic-tac-toe. *Mathematics Magazine*, *53*(4), 202--216.

Schaefer, S. (2002, January). *How many games of tic-tac-toe are there?* Retrieved 21 May, 2013, from `http://www.mathrec.org/old/2002jan/solutions.html`

Shannon, C. E. (1950, March). Xxii. programming a computer for playing chess. *Philosophical Magazine, Ser. 7*, *41*(314).

Stanford. (2013). *Game definition language.* Retrieved May 2013, from `http://games.stanford.edu/gdl.html`

Thielscher, M. (2010). *A general game description language for incomplete information games* (Tech. Rep.). The University of New South Wales, Australia.

Tromp, J. (1995a). *John's connect four playground.* Retrieved from `http://homepages.cwi.nl/~tromp/c4/c4.html`

Tromp, J. (1995b). *John tromp's connect four database.* Retrieved from `http://archive.ics.uci.edu/ml/datasets/Connect-4`

van Cranenburgh, A., Smid, R., & van Someren, M. (2007, January). Tic-tac-toe on fields of size n with m dimensions. *Encyclopedia of AI Project*, 13.

Glossary

Abstract Class A class that acts as a base for many other classes that share similarities . 16, 18

AJAX Asynchronus Javascript and XML - a way to load additional content into a website. 7

Alpha-Beta Pruning A more efficient version of the Minimax Algorithm that cuts of part of the search tree. 8, 9, 17, 18

Artificial Intelligence The study of making Computers act like humans. 1, 5, 6, 16, 24

Bit Board An alternative game board representation that works with bits instead of higher level objects.. 13

Cython an optimizing static compiler for python code. 11

Game Theory The study of finding the best possible outcome in a given situation. 4

Hacker News An online community for sharing startup and computer science related content. 7, 9

Heuristic Function An evaluative function that determines how good a current state is. 8, 22

History Heuristics Ordering moves based on a number of cutoffs a move had earlier. 12

Killer Heuristics A special version of History Heuristics. 12

Minimax An Algorithm to efficiently search a tree. 5, 9

Move Ordering Ordering moves after a pattern that will allow maximum cutoff. 12, 16, 22

Perfect Information The information known about a state does not change as the game progresses.. 4

PyPy a just in time interpreter for python that offers significant performance boosts. 11

Quiescene Search A supplemental search algorithm that is activated when terminal nodes are deemed interresting.. 12, 13

Shannon Type-A exploring all possible variations for a perfect but slow play. 6

Shannon Type-B exploring only important parts of the game tree for faster execution. 9, 19

Zero Sum For every positive gain there is a loss somewhere else. 4, 6, 24

List of Figures

List of Tables

Appendices

A Benchmarking

Move Ordering

```
Trying 3 iterations @ DEPTH: 5


Connect 4 Bit Test
Minimax: 1.667560s( 100.000000 % )
Negamax: 1.616630s( 96.945817 % )
AlphaBeta: 0.296737s( 17.794700 % )
NegaAlphaBeta: 0.298091s( 17.875895 % )
NegaAlphaBetaTT: 0.495325s( 29.703550 % )
Connect 4 Bit Test
Minimax: 1.808622s( 108.459185 % )
Negamax: 1.627230s( 97.581477 % )
AlphaBeta: 0.105092s( 6.302158 % )
NegaAlphaBeta: 0.106139s( 6.364905 % )
NegaAlphaBetaTT: 0.162600s( 9.750788 % )
```

Bitboard

```
Trying 3 iterations @ DEPTH: 5


Connect 4 Normal Test
Minimax: 7.635902s( 100.000000 % )
Negamax: 7.665586s( 100.388743 % )
AlphaBeta: 0.411665s( 5.391177 % )
NegaAlphaBeta: 0.425502s( 5.572382 % )
```

```
NegaAlphaBetaTT: 0.639861s( 8.379643 % )
Connect 4 Bit Test
Minimax: 1.645729s( 21.552520 % )
Negamax: 1.694620s( 22.192795 % )
AlphaBeta: 0.107524s( 1.408143 % )
NegaAlphaBeta: 0.105038s( 1.375585 % )
NegaAlphaBetaTT: 0.172701s( 2.261701 % )
```

Hash Table

```
Trying 1 iterations @ DEPTH: 8

Connect 4 Bit Test
NegaAlphaBeta: 5.304098s( 100.000000 % )
NegaAlphaBetaTT: 1.220764s( 23.015486 % )

-----------------------------------------------

Trying 10 iterations @ DEPTH: 8

Connect 4 Bit Test
NegaAlphaBeta: 2.985521s( 100.000000 % )
NegaAlphaBetaTT: 0.138866s( 4.651313 % )
```

Opening Book

```
Trying 10 iterations @ DEPTH: 8

Connect 4 Bit Test
NegaAlphaBeta: 3.025746s( 100.000000 % )
Loaded opening book: data/connect4bit.json with 77238 positions
```

```
NegaAlphaBeta: 1.722482s( 56.927507 % )
```

Complete Results

```
Trying 1 iterations @ DEPTH: 6


Connect 4 Bit Test

NegaAlphaBeta: 6.293436s( 100.000000 % )

NegaAlphaBetaTT: 0.142414s( 2.262899 % )


-------------------------------------------------


Trying 10 iterations @ DEPTH: 6


Connect 4 Bit Test

NegaAlphaBeta: 6.640882s( 100.000000 % )

NegaAlphaBetaTT: 0.020803s( 0.313260 % )
```

B Code

B.1 TicTacToe

Minimax

```
def minimax(node, player):
  if node.complete():
    if node.X_won():
      return -1
    elif node.tied():
      return 0
    elif node.O_won():
      return 1
        best = None
  for move in node.available_moves():
    node.make_move(move, player)
    val = self.minimax(node, get_enemy(player), alpha, beta)
    node.make_move(move, None)
```

```
    if player == 'O':
        if val > best:
            best = val
    else:
        if val < best:
            best = val
    return best
```

Wrapper

```
import random

...

def determine(board, player):
    a = -2
    choices = []
    if len(board.available_moves()) == 9:
        return 4
    for move in board.available_moves():
        board.make_move(move, player)
        val = board.alphabeta(board, get_enemy(player), -2, 2)
        board.make_move(move, None)
        print "move:", move + 1, "causes:", board.winners[val + 1]
        if val > a:
            a = val
            choices = [move]
        elif val == a:
            choices.append(move)
    return random.choice(choices)
```

B.2 Connect4

The Setup

```
class Connect4(object):
    def __init__(self, board=[], w=7, h=6):
        self.width = w
        self.height = h
        self.board = board
        self.xBoard = 0
        self.oBoard = 0
        self.values = [1, 8, 128, 256, float('inf')]
        for i in xrange(self.width):
            try:
                self.board[i]
            except IndexError:
                self.board.append([EMPTY] * self.height)
                continue
```

```
            for k in xrange(self.height):
                try:
                    self.board[i][k]
                except IndexError:
                    self.board[i].append(EMPTY)

    @property
    def available_moves(self):
        return [k for k in xrange(WIDTH) if EMPTY in self.board[k]]

    @property
    def complete(self):
        if len(self.available_moves) == 0 or self.winner != None:
            return True
        return False

    @property
    def winner(self):
        board = self.board
        #return reduce(lambda x, y: x + heuValue(game, [board[y[0]][y[1]], board[y[2
        for group in heuristics:
            one = board[group[0]][group[1]]
            if one != EMPTY:
                if one == board[group[2]][group[3]] == board[group[4]][group[5]] == b
                    return one

        return None

    def getLowest(self, column):
        board = self.board
        for y in xrange(HEIGHT - 1, -1, -1):
            if board[column][y] == EMPTY:
                return y
        return -1

    def make_move(self, column, player):
        lowest = self.getLowest(column)
        if lowest != -1:
            self.board[column][lowest] = player
        return lowest

    def make_bit_move(self, column, player):
        board = self.xBoard & self.oBoard
        pos = 1 << column * 7  # move to right column
        for i in range(HEIGHT):
            if pos & board == 0:
                if player == 'X':
                    self.xBoard |= pos
```

```
                else:
                    self.oBoard |= pos
                return pos
            else:
                pos = pos << 1
        return -1

    def remove_move(self, column):
        top = self.getLowest(column) + 1
        self.board[column][top] = EMPTY

    def remove_bit_move(self, pos, player):
        if player == 'X':
            self.xBoard ^= pos
        else:
            self.oBoard ^= pos
```

Heuristic

```
def heuristic(game):
    board = game.board
    value = 0

    for group in combinations:
        members = [board[group[0]][group[1]], board[group[2]][group[3]], board[group[
        oCount = members.count('O')
        xCount = members.count('X')

        if (xCount > 0 and oCount > 0) or (xCount + oCount) == 0:
            pass
        elif xCount > 0:
            if xCount == 3:
                row = group[2 * members.index(EMPTY) + 1]
                if game.oStart:
                    if row % 2 == 0:
                        value += -game.values[3]
                    else:
                        value += -game.values[2]
                else:
                    if row % 2 == 0:
                        value += -game.values[2]
                    else:
                        value += -game.values[3]
            value += -game.values[xCount - 1]
        else:
            if oCount == 3:
                row = group[2 * members.index(EMPTY) + 1]
                if game.oStart:
```

```
                    if row % 2 == 0:
                        value += game.values[2]
                    else:
                        value += game.values[3]
                else:
                    if row % 2 == 0:
                        value += game.values[3]
                    else:
                        value += game.values[2]
            value += game.values[oCount - 1]
    return value
```

Alpha-Beta

```
def alphabeta(node, player, alpha, beta, depth)
    if depth == 0:
        return heuristic(node)  # , player, alpha, beta)
    win = node.winner
    if win == 'O':
        return float('inf')
    elif win == 'X':
        return -float('inf')
    for move in distanceSort(node.available_moves):
        node.make_move(move, player)
        val = alphabeta(node, get_enemy(player), alpha, beta, depth - 1)
        node.remove_move(move)
        if player == 'O':
            alpha = max(val, alpha)
            if beta <= alpha:
                break
        else:
            beta = min(val, beta)
            if beta <= alpha:
                break
    if player == 'O':
        return alpha
    else:
        return beta
```

B.3 Efficiency

Move Ordering

```
def distanceSort(moves):
    for i in range(len(moves)):
        moves[i] -= 3
    moves.sort(key=lambda x: abs(int(x)))
    for i in range(len(moves)):
```

```
            moves[i] += 3
    return moves
```

Hash Table

```
# cython: profile=True
# -*- coding: utf-8 -*-
"""
anigmo.transpo - transposition table stuff for algorithms
"""

HASH_TABLE = {}

LOWERBOUND = -1
EXACT = 0
UPPERBOUND = 1


def storeEntry(hashes, best, typ, depth, player):
    for board in hashes:
        HASH_TABLE[board] = (best, typ, depth, player)


def getEntry(board):
    return HASH_TABLE.get(board[0], None)


def resetTable():
    global HASH_TABLE
    HASH_TABLE = {}
```

B.4 Anigmo

Example use

```
    >>> computer = anigmo.AI()
    >>> game = anigmo.games.connect4.Connect4()
    >>> computer.algo = anigmo.algo.AlphaBeta(depth=5)
    >>> move = computer.determine(game, 'O')
    >>> game.make_move(move, 'O')
    >>> game.show()
    1|2|3|4|5|6|7|
    -------------
    -|-|-|-|-|-|-|
    -|-|-|-|-|-|-|
    -|-|-|-|-|-|-|
    -|-|-|-|-|-|-|
    -|-|-|-|-|-|-|
    -|-|-|O|-|-|-|
```

Anigmo, the library

The whole code was included separately.