Fox and Geese AI

MiniMax, alpha beta pruning and genetic algorithm for a middle age game

University of Padua - Master's degree in Computer Science
Jordan Gottardo 1179739 - Giulia Petenazzi 1180066

1 Abstract

The evolution in the IT sector has had an exponential growth in the last years even thanks to artificial intelligence. AI has been used to develop and advance a wide range of activities, fields and industries, including finance, healthcare, education, transportation, robotics, electronic trading and more. But if you think for a while about all these applications, you probably realize that we use them to respond to basic human needs: improving our health, communicating, getting information about a specific topic, having fun and simplifying the execution of certain tasks. In this paper we want to focus on the "having fun" part because games represent one of the most active kind of AI applications in app stores, in terms of number of applications, downloads and users.

Moreover, considering the theoretical part of gaming, game theory has been an interdisciplinary research field since its creation (mathematics, economics, logic, IT), changing unthinkable fields, such as political science or social psychology. Inexplicably, game theory manages to reach a grandiose "mathematical depth": it provides a myriad of very interesting problems to study. For example, the search for the winning strategy for a given game is generally a difficult problem at a computational level and at first glance it seems we are unable to solve most of the games.

Attempting to find a solution for these kind of difficult problems, we have created an AI for "Fox and Geese", an ancient board game, within a cross-platform application. The aim is to let a human play against an artificial player, as if he or she was playing against another human.

2 Application Requirements

Before discussing the project design, we need to describe the game and point out some basic requirements for the project, such as game mechanics, game classification and device compatibility. In this section, we present a list and a description of these main topics.

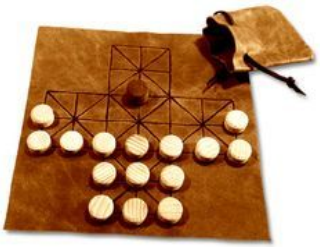

Figure 1: a real Fox and Geese board.

2.1 Game mechanics

This game is played in turns by two players, respectively the fox and the geese: the lone fox attempts to capture the geese, while the geese try to block the fox, so that it cannot move. Below are listed the game rules.

- The geese take the first turn. The player has to move one of them from its starting position, along horizontal or vertical (not diagonal) marked lines, and only forward, to an adjacent empty position.
- The fox then takes a turn, moving along any line, even diagonal ones, to an adjacent empty position. Play then alternates between the two players.
- No goose can kill the fox.
- Instead of moving as already described, the fox may kill an adjacent goose by jumping over it onto an empty position beyond the goose, provided that the points are linked by a marked line. The goose is then removed from the board.
- The geese win the game by trapping the fox, so that it is unable to move during its next turn.
- The fox wins by capturing most of the geese. When 4 or less geese remain on the gameboard they are unable to trap the fox, so the winner is the fox .
- The fox can also win if it reaches a position where the geese cannot trap it anymore due to movement limitations.

2.2 Game classification

Using the taxonomy described by Farooqui et al. [1], we can state that this game is a:

- two-player;
- zero-sum;
- asymmetric;
- perfectly informed;

game. It's a zero-sum game because each participant's gain or loss of utility is exactly balanced by the loss or gain of utility of the other participants. It belongs to the set of asymmetric strategy games, as there are different strategies for each player (the lone fox attempts to capture the geese, while the geese try to block the fox).

2.3 Device compatibility

This game has been designed to be a cross-platform application. For practical reasons we have tested the game on two Android phones (Nubia Z11 Mini S with Android 6.0.1 and Xiaomi Redmi Note 5 with Android 8.1.0) and on two laptops (Lenovo G500 and Dell Inspiron 7559, both with Windows 10).

3 Design and implementation

The application has been developed using the Unity framework and the C# programming language. The framework has provided a baseline for 3D game development and deploying on multiple platforms. To better promote decoupling and code reuse, we have split our code into modules:

- *UI*. This module manages the user's interaction while creating and joining games.
- *Board*. This module manages the rendering of the game board and the various pawns placed on it.
- *Game*. This module manages the game's logic by maintaining a local state of the game and modifying it whenever the player moves the pawns. It also provides a means to check the validity of moves and victory conditions.
- *AI component*. This module is the agent program which implements the appropriate agent function to play the game.
- *Controller*. This module contains a series of controllers that manages the interactions between the Game and the Board modules. These controllers respond to events coming from other modules and act accordingly.

With this separation into modules, suggested by the OOP approach [2], it should be fairly easy to extend the application with other functionalities and even reuse some modules in a future application. For example, we deem that changes in the game logic are simple to bring about due to the fact that the Game module is highly cohesive and other modules don't make assumptions on its behaviour.

In our implementation we decided to let the user choose whether he wants to play as the fox or the geese whenever he opens the application.

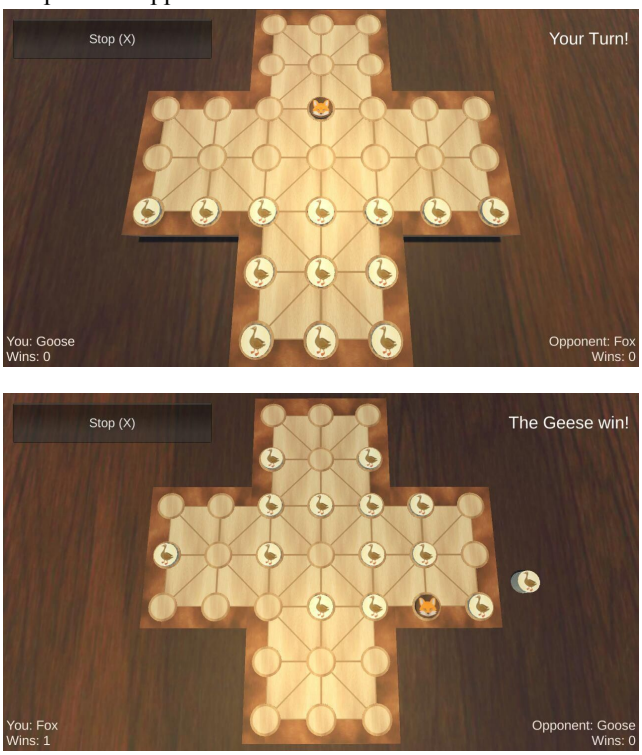

Figure 2: starting situation when playing as geese (above) and the geese win by trapping the fox (below).

4 Choice of AI algorithm

From now on, we will focus on the AI component, explaining design choices and implementation details. As a starting point, as we considered this game similar to the checkers/draughts game, we started analyzing Kitano et al.'s work [3], in which, after evaluating several possibilities, the MiniMax algorithm was cited. MiniMax is also used in an implementation of an AI for Backgammon by Sjöqvist et al. [4] (even if in this implementation the main algorithm used was ExpectiMax, a MiniMax variant more suitable for the stochastic scenario provided by Backgammon). We also found MiniMax used in a very basic and simplified version of the Fox and Geese game by Chisholm et al. [5].

Checking for confirmation, Russel and Norvig [6] states that MiniMax is good for zero-sum games, which is exactly our case, as described in section 2.2 "Game classification".

4.1 MiniMax

Minimax is an algorithm used in adversarial search problems, more commonly known as "games". More precisely, it deals with two-players, turn based and zero-sum games with perfect information, deterministic and completely observable environment. The two agents' actions have to alternate and their function utilities, at the end of the game, are always equals but with opposite signs.

We won't explain MiniMax in detail in this section since it is a classical AI algorithm. We will instead focus on complexity issues and how we have chosen to implement the algorithm in order to get good performance.

During a Minimax execution, the game tree is created starting from a certain state and the algorithm returns the best move for the agent, assuming that the adversary is infallible. In other words, by playing this move, the agent maximizes the result in the worst case.

Unfortunately the MiniMax complexity is exponential, O(b^m), where b is the branching factor and m is the depth of the deepest leaf. In our case: $b < 16$ for the fox and $b < 22$ for the geese, but m is more than 100. Such high values make it impossible to run Minimax in reasonable limits of time and space. For this reason we decided to stop the algorithm after 5 plies (a ply is a move of one player; two plies make up a turn) in order to have a move, albeit not the perfect one, in a reasonable time.

To sum up, the most important components of the MiniMax algorithm for our scenario are:

- the initial state, which in our case is the board configuration from which the algorithm starts;
- the next() function, which returns the list of legal moves and the resulting state by executing that specific move;
- the termination test, which in our case checks either if the ply limit has been reached or if one of the players has won;
- the evaluation function that evaluates the quality of a board configuration that appears in a leaf of the minimax tree.

4.2 Alpha-Beta pruning

In order to further reduce the algorithm's complexity, we tried to use a solution suggested by Russel and Norvig [6] and also used by Sjöqvist et al. [5]. This technique, called Alpha-Beta Pruning, consists in pruning some sections (leaves or entire subtrees) of the tree, returning the same solution one would usually get from MiniMax, as it prunes parts of the tree that are non-improving.

5 Feature elicitation

We realized that an important part of MiniMax is to define a good evaluation function that defines the quality of a particular game configuration. In order to provide an accurate representation, this function has to consider some particular characteristics of the board configuration, for example the number of remaining geese, combining them. We decided to combine this features in a weighted sum, as shown in figure 3, where w_i and x_i represent the weight and the value of feature *i* respectively.

$$y = \sum_i w_i \cdot x_i$$

Figure 3: high level evaluation function.

The next step was to discover the features that could be useful. Here there is a list of the features we considered:

- *Number of geese*: this represents the number of remaining geese on the board. This feature is important in order to evaluate the progress of the game. This numbers is an integer in the range [0, 15].
- *Number of ahead geese*: this is the number of geese that there are in rows in front of the fox. As the geese cannot move backwards, they pose a greater threat when they are still able to fully respond to the fox's movements.. This numbers is an integer in the range [0, 15].
- *Number of possible moves*: this is the number that indicates how many possible moves the fox has. This number is an integer in the range [0, 8].
- *Number of possible eating moves*: this is the number that indicates how many geese the fox can capture from its current position.. This number is an integer in the range [0, 8].
- *Geese freedom*: this is the number that indicates how much the geese are free to move. It is calculated using the mean of the *degree of freedom* of every remaining goose. The *degree of freedom* of a goose is calculated as ratio between the free positions around it over 8 (number of total positions around it). This number is a real number in the range [0, 1], where 0 indicates a board where the geese are compact and 1 a board where the geese are sparse.
- *Internality*: this is the number that indicates if the fox is in one of the positions which have a high number of possible moves, typically located near

the center of the board. This number is either 1 or 0.

- *Externality*: this is the number that indicates if the fox is in one of the positions which have a low number of possible moves, typically located near the edges of the board. This number is either 1 or 0.

After having discovered these features, we had to choose the right weight for each one of them in order to build a suitable board evaluation function.

6 Feature selection: genetic algorithm

Since the number of identified feature was quite high and a manual choice of the weights was unmanageable, we have looked for ways to automate this choice. This process consists in an informed local search through the set of various weights assigned to the features mentioned in the previous section.

We have chosen to implement this search using a genetic algorithm. We deemed this was a good choice for the problem in question. Despite the fact that it doesn't guarantee an optimal solution, even a suboptimal solution would have sufficed for our purposes. Moreover, the algorithm is fairly easy to understand and to implement.

We have followed Man et al.'s work [7] to get a good grasp of the basics, pros and cons of genetic algorithms..

We have implemented the algorithm as follows:

- Starting from an initial population of 50 evaluation functions with randomly generated weights, each function's initial fitness is evaluated through some matches against an opponent which executes completely random moves.
- The population then goes through a 50-generation evolutionary process. In each generation, each function's fitness is evaluated through a tournament, where it plays against all other functions. The win/loss score is then recorded and used as an indicator of a function's fitness.
- During each generation, the top 20% of evaluation functions is retained, while each one in the bottom 80% is retained only with a 5% chance.
- The mating process between two functions consists in a combining the weights of each feature of the two functions. This combination considers the bit representation of each weight (which is an integer). One of the two mating functions, which assumes the role of a "male", provides bits 1, 2, 5, 6, 9, 10, 13 and 14 of each weight, while the "female" provides bits 3,4, 7, 8, 11, 12, 15 and 16 . The resulting weight is then mutated with a 1% chance by switching one of its bits.

The execution of the genetic algorithm has proved to be very taxing on our resources: we had to limit the number of functions to be analyzed, since otherwise the space and time requirements would exceed the resources at our disposal. As a result of this, the use of the algorithm has been useful to determine which features to use in a reasonable evaluation function and the signs of their weights, but its effectiveness has been limited regarding the determination of exact values.

Some points that we discovered thanks to this algorithm are:

- *geeseNumber* and *aheadGeeseNumber* are the most important features, and their weights assume positive values;
- *foxEatingMoves* and *foxMoves*'s weights assume negative values;
- *gooseFreedom* and *internality*'s weights assume negative values;
- *externality*'s weight assumes positive values.

The function we obtained as a result of the algorithm, after normalizing the weights, is the following:

$$y = 4 \cdot geeseNumber + 3 \cdot aheadGeeseNumber - 1 \cdot foxMoves$$

The agent using this evaluation function behaves quite well as the fox player, playing the game at a reasonable level against the human player.

When playing as the goose player, the agent's performance is not great. We can probably relate this also to the fact that Fox and Geese belongs to the set of asymmetric strategy games. As said in Edelkamp's work[8] "the portions of tactic and strategy are not equal for both players". Further studies are required to confirm this hypothesis..

7 Future work and extensions

In this section we cover the future work and extensions we think could benefit this project.

7.1 New features for board evaluation

Although we have played and analyzed the game, we don't consider ourselves by any means experts of Fox and Geese. Our analysis and feature discovery could be improved by interacting with more experienced players.

7.2 Machine learning

The agent we have implemented does not learn in any way. Its behaviour is static and, since the evaluation function is

not optimal, it could be exploited by smart players by capitalizing on the agent's misplays to win games more easily. As discovered by Kitano et al. in [3] a properly trained neural network produces a way better performing AI for that specific variant of Fox and Geese. A future extension of our work could test whether the same holds true even for this version.

7.3 Cloud execution of MiniMax

Since the execution of the MiniMax algorithm is quite taxing on the resources, especially those in a smartphone, we had to limit its look ahead capability as discussed in section 4. A better way to handle this issue could consist in moving the execution of the MiniMax algorithm to the Cloud to exploit more performing processing capabilities. This should also lead to the possibility of increasing the look ahead to have a better performing agent.

7.4 Further studies for goose performance

Since the agent's performance is not great when playing as the goose player, further studies are requested to confirm that it is due to the asymmetry of the game, improve the performance of the AI and eventually develop new functions that take in consideration some ad-hoc strategies for particular game configurations.

7.5 Tutorial and "suggest best move" functionality

Even if the game is quite easy to learn, it's fairly hard to master. The application could be improved by providing a tutorial to help new players get accustomed with the game. Moreover, the application could benefit from a functionality which would suggest the next best move for the human player, to be used only a limited number of times during a game. Since MiniMax already tries to find the best move, this functionality should be fairly easy to implement.

8 Conclusions

In this paper we have described and analyzed our implementation of the Fox and Geese game and AI. The implemented agent is sufficient for our purposes since it can play the game at a reasonable level as the fox player. However, once the human player gets to know the game better, he or she can use certain strategies to beat the AI fairly easily. In conclusion, the agent's performance can be improved: we have described some ways in section 7 "Future work and extensions".

9 References

[1] Aisha D. Farooqui and Muaz A. Niazi, 2016. Game theory models for communication between agents: a review - [https://arxiv.org/ftp/arxiv/papers/1708/1708.01636.pdf]

[2] Bruce L. Horn, 1988. An introduction to object oriented programming, inheritance and method combination - [https://pdfs.semanticscholar.org/0cb2/1a77828db3c0065a5b745ade57145077a89d.pdf]

[3] Hugo Kitano, Daniel Gallegos and Marco Monteiro. An Artificial Intelligence For Checkers - [https://web.stanford.edu/class/cs221/2017/restricted/p-final/hkitano/final.pdf]

[4] Anders Sjöqvist and André Stenlund, 2011. Constructing an Evaluation Function for Playing Backgammon - [http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group3Johan/final/Anders.Sjoqvist.Andre.Stenlund.report.pdf]

[5] Kenneth J. Chisholm and Donald Fleming. A Study of Machine Learning Methods using the Game of Fox and Geese - [http://web.archive.org/web/20151029210210/http://cswww.essex.ac.uk/cig/2005/papers/p1003.pdf]

[6] Stuart J. Russell and Peter Norvig, 2009. Artificial Intelligence A Modern Approach Third Edition, - [chapter 5 - Adversarial Search]

[7] K. F. Man, K. S. Tang and S. Kwong, 1996. Genetic algorithms: Concepts and applications - [https://www.researchgate.net/publication/3217198_Genetic_algorithms_Concepts_and_applications]

[8] Stefan Edelkamp, 2017. BDDs for Minimal Perfect Hashing: Merging two State-Space Compression Techniques [https://materials.dagstuhl.de/files/17/17181/17181.StefanEdelkamp.Preprint.pdf]