CSC111 Project Proposal: Traversing Game Trees Intelligently

Mark Bedaywi

April 16, 2021

Problem Description and Research Question

On the 15th of March 2016, Deepmind's AlphaGo artificial intelligence beat world champion Lee Sedol in a series of Go matches (Hassabis, 2016) both shocking the world, and inspiring me to pursue a career in computer science. What makes this such an incredible feat is due to the fact that there are $250^{150} \approx 10^{360}$ (Silver et al., 2016) possible games of Go. It would be near impossible to create a game tree of this size, let alone be able to traverse it! To be able to beat the best humans despite this, AlphaGo uses a method known as a Monte Carlo tree search to traverse the game tree and teach itself to play Go (Silver et al., 2016). The purpose of this project is to create an AI inspired by AlphaGo and to answer the question, **which strategies of traversing game trees and picking moves, such as a Monte Carlo tree search, work best at winning games of strategy without having to build the full game tree?**

A Monte Carlo tree search consists of four parts. The first is a selection phase where the leaf node with the largest probability of victory is chosen. The next is an expansion phase where, if the game doesn't end, the children of the leaf node are created. The one after that is an evaluation phase, where the probability of victory of the new leafs are calculated, either by a neural network, or by simulating games where the players make random moves. The last part is the backup phase, where the probability of victory of all the ancestors of each node is calculated (Silver et al., 2016).

This project focuses on three main parts. Firstly, focuses on creating a simplified version of the game tree traversal strategy AlphaGo uses to be able to beat humans at 9×9 Go. Secondly, focuses on investigating how different tree traversal strategies compare in performance and effectiveness to this simplified AlphaGo – such as a minimax tree search, or a Monte Carlo tree search without a neural network. The focus of this project is on the game of Reversi, but a variety of other games is used to make this comparison, namely Connect Four and Tic Tac Toe. How the performance of each algorithm changes is investigated. Lastly, the project focuses on creating different methods of evaluating tree traversal strategies to be able to rank them, as well as creating a user interface to display this information and to play games against the different algorithms.

Datasets

A database of tournament Reversi games played by humans is taken from https://i.cs.hku.hk/ kpchan/Othello/TrainingSet.html.

Computational Overview

A GameState class is created in the game module that implements all methods required to play the game. This includes finding legal moves, making moves, finding the winner, returning heuristics evaluating the current position. A Game class is created to manage GameStates.

TicTacToeGameState, ConnectFourGameState, and ReversiGameStates are classes that inherit from the GameState object, implementing their respective games. Note that Reversi is implemented where a pass is played when no legal moves are available, and the game ends after two passes.

Trees are used heavily in the program as game trees. A GameTree class is created in the game module that implements this and multiple different GameTree classes inherit from it. Each node in the game tree stores a GameState and GameTrees that its children. The GameTree is built through repeated expansions of leaf nodes. A Player class is created that uses the GameTree to find the best move to play.

A MinimaxGameTree is created that inherits form GameTree. Here, each node also stores the value of the node. The minimax algorithm, as described in Strong, 2011, is implemented to find the value of a node. The full game tree is not necessarily built, as the values are calculated up to a certain depth then a heuristic implemented in GameState is used to find the value of the leaf. This value is propagated upwards, assuming that each player plays the optimal move. This uses alpha-beta pruning to speed up the search, where the best and worst values of previously calculated nodes are remembered to check if the value of the rest of the children are worth calculating. Moreover, the calculated values are memoized.

A MonteCarloGameTree is created that inherits from GameTree, implementing a basic monte carlo tree search. Each node stores a value and the number of times it has been visited. Here, the value of a state is approximated by first starting at the root, then choosing moves worth exploring through combining the current calculated value of the move, and a term representing how little we explored the move compared to other children. If a leaf node is reached, then all children are added, and the value of the leaf node is then calculated and propagated up the tree. The instance attribute self.repeat holds how much times this is done.

A MonteCarloSimulationGameTree is created that inherits from MonteCarloGameTree, finding the value of a move by simulating a possible game with random players, and returning the winner.

A MonteCarloNeuralNetwork is created that inherits from MonteCarloGameTree, finding the value of a move by querying a trained neural network. Here the library sklearn is relied on to build the classifier. The standard library pickle is relied on to store and retrieve trained neural networks. The function train_neural_network is created to train a classifier by creating players that use the neural network and having them play games with themselves over and over. The neural network is then trained on each game state in the game with the value of each state being the winner at the end. The classifier is trained each game and updated if it can beat the version of itself from the previous iteration in a number of games, that is, the neural network is updated if it improves.

A NeuralNetworkPlayer is created that uses a trained neural network to find the move to play by simply playing a legal move that the classifier classifies as winning.

A ReversiOpeningsGameTree is created that reads the dataset of Reversi games and builds a game tree that where only the moves played in the dataset are present recursively. This is then used by a ReversiOpeningsPlayer until the tree reaches a non-terminal leaf node, or the opponent plays a move that is not in the game tree. In this case, the ReversiOpeningsPlayer reverts to some other default player.

The library pygame is used to display the games. Each GameState has a function allowing it to display itself on a pygame screen. The function display_game in the game module is used to display a sequence of states. The library Tkinter is used to build a simple GUI that allows the user to pit two AIs against each other in a chosen game, then display the game after it is complete.

Instructions

The dataset containing the Reversi games, as well as stored, trained neural networks, can be found in the .rar file submitted in MarkUs. For safe measure, it can also be found in the google drive, https://drive.google.com/drive/folders/1phzSJj40-NiiZyr1_aX8wvI70mYR92nJ?usp=sharing.

To run project, simply run the main file, click the desired game, the desired players, as well as inputting player dependent features like the depth if Minimax is chosen, or whether an opening game tree should be used if Reversi is chosen. Then, wait for the game to finish being played and a pygame display displaying the moves taken will pop up. Press the left and right arrow keys to go through the history.

If a human player is chosen, click on the pygame screen on your turn to play a move. The pygame display with the history of moves taken will still be displayed when the game is over.

To play another game, run the main file again.

To train the neural network, run the module monte_carlo_neural_network. The parameters num_games can be changed if the training takes too long.

Changes From Proposal

The biggest change from the proposal is that Reversi was chosen instead of Go, as well as Connect Four instead of Hex and Poker, to simplify the project. Additionally, Tic Tac Toe is implemented, as recommend, which was the perfect way to identify and fix errors. Moreover, instead of the library Keras, MLPClassifier in sklearn.neural_network is used, as recommend, which is a lot easier to work with. Moreover, as recommend, the GUI doesn't display statistics or the rules, as it distracts from the main focus of the project.

Discussion

The results of the computational overview do help answer the research question, and all three project goals were achieved. Firstly, a simplified version of alphaGo was created. This player is better than random, better that the classifier on its own, and better than a corresponding MCTS with a simulation to decide on move values in all three games. This tree traversal strategy managed to make the computer play Reversi very effectively compared to the other strategies.

Secondly, we can rank the performance of a range of the different players using this program. It seems that the Random player performs poorly consistently, as expected. Then, the neural network on its own comes next. Then, the MCTS with simulations, and the minimax tree search comes next, differing in terms of the time needed to choose the best move, depending on the parameters chosen. Of course, the minimax player with unlimited depth is the best, but is unfeasible, at least with my implementation, for Connect Four or Reversi.

It is also worth noting that how the AIs play the game differ. For instance, the Minimax player has a tendency to give up and play an arbitrary move when it realizes that it will be forced to lose, while the Monte Carlo search tree players try as best as possible to prolong a potential defeat.

These traversal strategies have a hard time playing Reversi however, which is not too surprising, as it is hard to determine who is going to win the game until the final few moves, at least for a novice like me.

The best player, that plays in a reasonable amount of time, seems to be the MCTS with a neural network, which is exactly the result hypothesized, answering the research question. Moreover, adjusting for relevant parameters, the relative performance of each algorithm seems to stay consistent when changing the game.

Thirdly, we can directly look at the evaluate the performance of different search tree traversal strategies by playing against them, watching them play against each other, and looking at the number of times it wins after a number of games against some opponent. Because of these algorithms, it is easy to train the neural network as it is easy to determine when an improvement has occurred after training.

The biggest limitation would have to be the training time for the neural networks. It took a very long time for the classifier to learn to play a game as simple as Tic Tac Toe ideally. In addition, the MCTS relies on random simulations of games, which is not a problem in Tic Tac Toe, but is costly in the much larger Reversi, especially when multiple hundred random simulations are performed. Moreover, while the performance of the classifier assisted by the Monte Carlo search tree is great, the performance of the classifier alone is, while still better than random, not as large as the other tree traversal strategies. This may be offset by the fact, however, that it chooses a move almost instantaneously. Lastly, the addition of an opening did not seem to have as much of an effect on the performance of the AI in Reversi as I had expected. This may be due to the size of the dataset, or the fact that the winner of the game is not considered when taking a predetermined move.

The next steps, which I will continue to work on during the summer, would be to follow alphaGo more closely and have the neural network also learn which states are worth exploring in MCST, instead of relying on the number of visits. Another step for further exploration would be implementing and comparing different heuristics for the minimax search tree. Moreover, it would be interesting to find how these algorithms perform in different games, such as games with randomness like BackGammon, or games with imperfect information like Poker. It would also be worthwhile to continue to try and compare different neural network architectures to find the one that results in the best performance with the least amount of training. Lastly, a good next step would be working on speeding up the code that runs the game so that random simulations occur faster.

References

Sources:

- Hassabis, D. (2016, March 16). What we learned in Seoul with AlphaGo. Google Blog.
  https://blog.google/technology/ai/what-we-learned-in-seoul-with-alphago/
- Silver, D., Huang, A., Maddison, C. et al. (2016). Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489.
  https://doi.org/10.1038/nature16961
- Strong, G. (2011). The minimax algorithm. Trinity College Dublin.

Dataset:

- Othello Training Set. (2016). Othello Training Set.
  https://i.cs.hku.hk/%7Ekpchan/Othello/TrainingSet.html

Tutorials & Documentation for libraries:

- Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011. http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html
- Graphical User Interfaces with Tk — Python 3.9.2 documentation. (n.d.). Python Documentation. https://docs.python.org/3/library/tk.html

Tutorials, Articles, and Papers that helped build the AlphaGo inspired Monte Carlo Tree Search:

- Monte Carlo Tree Search for Tic-Tac-Toe Game in Java. (2020, October 3). Baeldung. https://www.baeldung.com/java-monte-carlo-tree-search
- Nair, S. (2017). Simple Alpha Zero. Stanford. https://web.stanford.edu/%7Esurag/posts/alphazero.html
- Silver, D., Huang, A., Maddison, C. et al. (2016). Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489. https://doi.org/10.1038/nature16961
- Silver, D., Schrittwieser, J., Simonyan, K., et al. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354–359. https://doi.org/10.1038/nature24270

The first two were especially helpful for understanding fully how a Monte Carlo tree search worked.