An AlphaGo-Zero Inspired Quoridor Player
Senior Project (CPSC 490)

By: David Liu
Advisor: Dr. James Glenn

*Special Thanks to Professor James Glenn for his Advice and Wisdom*

Abstract
Quoridor is a board game that involves both spatial intuition and strong logical analysis -- in brief, each of the players try to get their piece to the other side of the board while walls are being placed by each player. This game has a complexity similar to that of chess and previous attempts at creating a strong computer player for Quoridor using methods such as alpha beta search with heuristics and methods using genetic algorithms have not been very successful. Though I believe methods involving genetic algorithms still hold a lot of potential and is a promising path, I decided to attempt to develop a strong Quoridor computer player using an AlphaGo-Zero based framework. Due to limits of the computational resources I have access to, I limited the game to a board of 9x9 (including wall slots) with 3 walls per player instead of the usual 17x17 with 10 walls per player. I used a greedy Quoridor algorithm as a metric point, and the AlphaGo-Zero based model that was trained for 84 iterations (25 hours) beat the greedy player 94% of the time, lost 3% of the time, and drew 3% of the time. Against humans with little exposure to Quoridor, models like this one pose an interesting match but still can be beaten with limited cognitive exertion from the human players. It’s likely that more training would eventually lead the model to overcome human play but the improvement of the model over iterations seems to follow a logistic function curve, so in the later stage, growth is very slow. Some challenges of using an AlphaGo-Zero based approach include slowness of training due to bad self play data from the early iterations, the difficulty of testing different neural net architectures quickly, and incorporating data that’s not part of the board into the neural network architecture. Additionally, through analysis of the 9x9 version of Quoridor and the corresponding models trained for it, I have realized the limitations of this reduced board and have started investigations into the 13x13 version of the game.

*The rest of the report will refer to the AlphaGo-Zero Quoridor as AlphaQuoridor.*

Associated Project Items
- AlphaQuoridor Code Files
  - Includes readme that details how to run the models and how to play against the models.
- Includes the greedy Quoridor player and the random-move Quoridor player.
- AlphaQuoridor Models (35 Iterations, 49 Iterations, 84 Iterations, 146 Iterations)
- Human vs Human Quoridor Web-based Interface Code Files

Introduction
Quoridor is an abstract strategy board game which was invented in 1997 by gigamic and previously held the honor of Mensa’s Game of The Year title in the USA, France, Canada, and Belgium. This game involves a 9 by 9 grid-styled board (17x17 when including wall slots), and the objective of both players is to get his walker piece to the opposite side of the board. The grid lines on the board are indented such that a wall can be placed. Walls have a length of 2 squares, and each player only has 10 walls. Walls can be placed anywhere as long as it’s spanning two squares, and as long as both players have a path to victory. This is a turn based game and during one’s turn, the player could either move his walker piece into an edge adjacent square, or place a wall. If the edge adjacent square is taken by the opposite player, this player may jump over the opposite player. The first player to reach the opposite end wins. Quoridor has an average branching factor of 60, a game-tree complexity of 162 (log 10), and a state-space complexity of 42 (log 10). Unlike chess and Go, there’s limited literature on Quoridor and the evaluation of computer players for this game is more difficult. In chess, one can quickly tell if a computer player is going awry, but in Quoridor, one has to play the computer player for longer to make a proper judgement. This is due to the fact that there’s a lot of uncharted territory in Quoridor and an early move that may seem bad can turn out to be great.

In the smaller version of Quoridor that this project is based, there are 5x5 grid spaces which player pieces can move onto and there are wall slots in between those grid spaces, producing a 9x9 matrix of spaces. In this smaller version, each player has 3 walls to use.

Few people have attempted to build a strong computer player for Quoridor due to the relative youth of the game, and none have been able to build a computer player strong enough to beat a median Quoridor human player. Mertens and Glendenning are two individuals who have done research involving
constructing a strong Quoridor computer agent, and they explored approaches in the topics of evolutionary computation, 1-layer neural network, and various search methods using different combinations of features.

**Brief High Level Overview of AlphaGo-Zero**

AlphaGo-Zero, built by Google DeepMind, came into the spotlight on October 19th, 2017 for its ability to gain a high level of mastery of the game Go without any prepared training data. It quickly started conquering other games afterwards. It generates data through self-play which is then used to train a neural network with two outputs, a continuous value of the board state from -1 to 1 and a policy that is a probability vector of the possible actions from that state. The original AlphaGo-Zero uses 20 residual blocks each with 2 convolutional layers, and trained on 64 GPU workers and 19 CPUs. Then this current model is matched up against the best previous network, and if the current model achieves a win rate of 55%, then the network is updated to the current model. This process is repeated for a set number of iterations. AlphaGo-Zero had 25000 episodes of self-play per iteration and 1600 simulations per turn. The MCTS used to select the next move obtains the value of a new node through the neural network instead of having a rollout. Also, the MCTS used was asynchronous and could perform the simulations in parallel. DeepMind manages to engineer AlphaGo-Zero to perform the self-play, neural network training, and matches against the older model all in parallel. More details on the original AlphaGo-Zero can be explored through the original paper by DeepMind, “Mastering the Game of Go without Human Knowledge”.

**AlphaQuoridor Design**

The base of the synchronous single-thread single-GPU AlphaGo-Zero framework was created and made accessible by Surag Nair and I cannot credit him enough for his work. I wrote the files that defined the Quoridor logic and Quoridor game as well as making substantial changes to how the Monte Carlo Tree Search interacts with the game on an implementation level, adapting the Coach class to work with Quoridor, and augmenting the neural net architecture to account for the various Quoridor pieces and some information corresponding to those pieces. Overall, there was a lot in the framework that needed to be adapted for use in Quoridor.

The neural network for AlphaQuoridor is composed of 4 convolution layers that process the wall layout of the board and player positions on the board, as two separate channels. Convolutional networks should work very well with Quoridor since the shapes that the walls form are important to examine in strategy, and convolutional networks are excellent at capturing shapes. The input to these layers is a 9x9x2 tensor (disregarding batch size for a moment) with the first 9x9x1 layer being a 2D matrix representation of the board with 1’s where there are walls and 0 everywhere else. The second 9x9x1 layer has 0’s everywhere except 2 in the spot where the current player is and -2 where the opponent is (the choice to use 2 and -2 instead of 1 and -1 is purely a choice to improve the readability of the code so that walls are easily distinguished from players). To make it easier for the neural net to train quickly, symmetry is heavily used through flipping the board and the signs of the player piece representation values whenever it’s player 2’s turn so the neural net is always trying to move the positive player piece in the downwards direction. Each of these 4 convolutional layers has 512 filters and there’s no pooling layer. After these 4 convolutional layers, the result is flattened and fed through 2 dense layers with dropout. At the end of that
point, the 512 neurons from the last dense layer are concatenated with the “walls remaining” information, a 1-D vector of two values, resulting in a 1-D vector of 514. This combined layer then feeds into a dense layer representing the possible actions and another dense layer with 1 neuron representing the value output.

It was a challenge figuring out how to best incorporate the walls remaining information and I still have ideas on what might work better given more time to play around, such as adding 1 dense layer between the combined layer and the final action and value layers. Prior to this neural network design, I initially tried having a 9x9x3 tensor input that had the board as one 9x9x1 layer (with both wall and player information present in that layer), the walls remaining for player 1 as every value for the second 9x9x1 layer, and similarly with the walls remaining for player 2 in the third layer. The model produced by this architecture consistently lost to the model produced by the first architecture given similar amounts of training. I did create a model that completely disregards information about the number of walls remaining for each player and this model did noticeably worse than the two architectures described above given similar amounts of training.

I also explored versions of Quoridor that allowed for unlimited walls but that proved to change the nature of the game dramatically. These versions of the game often quickly turn into a game of Nim where the players are on the same path (because of a board saturating number of wall placements in the beginning) and the first player that has to move his player piece loses because the other player could jump over that player piece and be closer to the finish line. In order to stall moving his player piece, the player would need to place walls to use up his turn. Walls can reduce the possible available wall openings by 1 or 2 and therefore, the game becomes very much like Nim.

Training
Training works through a process of self-play, feeding that data from self-play into the neural net, and play against the best previous version to decide whether to update the model to the current version or to stick with the best previous version. This entire sequence makes up 1 iteration. To speed up training, symmetry is used to generate additional data by flipping the board horizontally in the left-right direction and using that as another training example. Wins are worth 1, losses are worth -1, and draws are worth 1e-4 (.0001). AlphaQuoridor was trained on an Intel i7 6700k desktop with a GeForce GTX 1080 graphics card and 16GB of RAM.

Training Parameters
Models are trained with 85 episodes of self-play followed by 10 epochs of training followed by 40 matches against the best previous model to decide if the current version is better than the best previous model. Monte Carlo Tree Search (MCTS) of 25 simulations per move is used during self play, and 50 simulations per move is used during matches against the best previous version. The Adam optimizer was used with its learning rate parameter set to .001. The batch size used was 64, and only data from the most recent 20 rounds of self-play (85 episodes per round) were used for training. 20 rounds of data is about 75,000 data points.
Experiments with fewer episodes of self-play per iteration led to more models being rejected due to losses against the best previous version. A high rate of rejections slows down training because matches against the best previous version take a substantial amount of time (much more than the 10 epochs of training), and if the most recent model isn’t that different due to less data from fewer episodes, those matches are a waste of time.

Greedy Quoridor Player
The Greedy Quoridor Player was created to be used as a metric for how well the AlphaQuoridor models were improving. This greedy player uses minimax to a depth of d, and upon reaching the terminal depth, applies a heuristic based on the difference of the current player’s shortest path to victory and the opposite player’s shortest path. This greedy player beats the random player 90% of the time and draws 7% of the time. Due to the high branching factor of the game, increases in the depth of the minimax search exponentially increases the time needed to calculate the next move.

Observations
- Similar to the experience of human players, it’s difficult to decide which moves to make in the beginning and easier to decide which moves to make in the late stage of the game when there’s more structure through the maze of walls. This difficulty in the early stage and ease in the later stage is expressed by younger models. Some lightly trained models would have no clue what to do in the beginning but once some walls are placed, it starts realizing the path it needs to go down to win. The mature models are able to make early game moves that make a lot more sense.
- Very rarely AlphaQuoridor will self-sabotage and continue moving between two squares and not attempt to win. This happens very very rarely but I don’t quite understand why that happens.

Results
*Each comparison is based on 100 matches

1. Greedy Vs Random - Win: 90%, Loss: 3%, Draw: 7%
2. 35 Iterations (9 Hours):
   a. Vs Random - Win: 98%, Loss: 0%, Draw: 2%
   b. Vs Greedy - Win: 3%, Loss: 26%, Draw: 71%
3. 49 Iterations (12 Hours):
   a. Vs 35 Iterations Model - Win: 13%, Loss: 0%, Draw: 87%
   b. Vs Greedy - Win: 37%, Loss: 3%, Draw: 60%
4. 84 Iterations (25 Hours):
   a. Vs 49 Iterations Model - Win: 61%, Loss: 0%, Draw: 39%
   b. Vs Greedy - Win: 94%, Loss: 3%, Draw: 3%
5. 146 Iterations (46 Hours):
   a. Vs Greedy - Win: 35%, Loss: 2%, Draw: 63%
   b. Vs 49 Iterations Model - Win: 6%, Loss: 0%, Draw: 94%
   c. Vs 84 Iterations Model - Win: 16%, Loss: 1%, Draw: 83%
   d. Vs Human - Win: 10%, Loss: 75%, Draw: 25%
One interesting thing to note in the results is that it seems like the 84 iterations model beats greedy significantly more than the 146 iterations model, but the 146 iterations model beats the 84 iterations model. My hypothesis on this is that since the 146 iterations has been pitted against stronger players than greedy in its matches against the best previous versions, this model has learned to avoid moves that exposes itself to the enemy. Consequently, this model plays more conservatively and wins against the greedy player less but also loses against the greedy player slightly less.

**Conclusion**
An AlphaGo-Zero approach produces decent models for playing Quoridor, and theoretically would be able to produce a model that can consistently beat human players given enough time to train.

**Slow Training**
The data that the model is trained on is based entirely on self play and though this offers versatility in what this AlphaGo-Zero framework can accomplish, the training often takes a very long time. Part of this is due to the poor data produced by self play in the beginning iterations. The policy and value network are randomly initialized to begin the process and moves in the early stages are near random. Eventually a player wins in the games, but the moves that led to that win by the winning player aren’t necessarily good moves in response to the corresponding state of the game. However, the win back propagates over the moves sequence and rewards those moves. Consequently, the training in the beginning is very rough and creates inertia that needs to be overcome in later training with better data. One way the framework addresses this is by holding only the data from the x most recent self play rounds. In my case, x was 20. Though this alleviates this issue to an extent, the neural net edge weights still hold on to some of that history.

Another issue is that the random initialization biases the model to make wall moves in the beginning training stages. This is problematic because many human Quoridor players would agree that being conservative with wall usage is generally a good heuristic. In the AlphaQuoridor framework, since each move has about the same probability to begin with and there are many more possible wall moves, most of the early data has a lot of early wall placements and therefore, the performance of delayed wall placement is not evaluated very much. In examining AlphaQuoridor’s moves as it developed, I noticed that it took a long time for the model to realize that saving some walls for the end is great. Delayed wall placement seemed to be a later development. A possible solution for this issue is biasing the initial initialization towards a more balanced probability distribution between player piece moves and wall placement moves.

**Realizations about 9x9 game**
These experiments and further analysis of the 9x9 game have shown me that this smaller version of Quoridor has serious gameplay flaws that take away from its ability to represent the 17x17 original version. In this smaller version, the smaller size of the board makes it much easier for a player to force a draw. Whenever player one realizes that he’s going to lose, he can often retreat to his starting row (player two’s victory row) and block player two from winning. Eventually, a draw is called. AlphaQuoridor is very good at doing this. More generally, the player piece has a lot more power in this smaller version because player pieces run into each other more often and that dynamic plays a much larger role in the outcome of the game. The strategy of blocking and player jumping (a player can jump over another player
that’s in the block he wants to move to) becomes much more significant. In the original 17x17 version, player pieces are often much further apart so blocking is difficult and jumping is more rare.

**Draws Taking Away from Data Potency**
Since it’s easy to force a draw in this smaller version of Quoridor, in the later iterations, the models have learned this fact and the matches between the most recently trained model and the best previous model result in all draws. This is bad for learning because the moves that are made in that game are neither punished or rewarded. A player that makes poor moves and then resorts to forcing a draw does not have its sequence of its moves punished. Similarly, the player that’s doing well and making strong moves does not have its sequence of moves rewarded if the opposing player forces a draw. Although, another perspective on this dynamic is that it could be a good thing that players need to learn how to prevent their opponent from being able to force a draw.

**Slow Exit from Local Minima**
After 100 iterations, the model seemed to be hovering around a local minimum because plays against best previous versions led to all draws, but a human player could achieve wins playing against these models. To get out of forced draw situations, a sequence of moves outside the norm is usually needed and for the exploration term of MCTS to select all those moves is very unlikely. If only a few of those moves from the sequence are selected, they could lead to a loss so they’re discouraged. Due to this, a lot of training time is needed for the network to discover the better minimums and develop in that direction. One potential solution to this issue is to code a system to increase the weighting of the exploration term of MCTS based on how often all the most recent matches are yielding draws.

**Looking Forward**
Going forward, I wish to explore my other ideas for neural net architectures that can better incorporate state information that isn’t represented on the board itself. I also wish to see how applicable this design is for the 13x13 Quoridor and will continue training the model for that version. I’d also like to improve the speed at which Quoridor games can be played by computers. The bottleneck right now seems to be the get_legal_moves() function because to check if a wall placement is valid, the program does a breadth first search for both player 1 and player 2 to ensure there still remains a path to victory for each player. This validity check happens for each potential wall placement move.

**References**