Brandon Druschel
5/15/19

CSC 466
Research Paper

Peg Board Learning Machine

For the semester, I took it upon myself to develop a machine learning program which attempts to yield progressively better results in a game of peg solitaire. To accomplish this, I took a *genetic algorithm* approach to learning the game.

Peg Solitaire, also known as *Solo Noble,* is a board game in which one player moves a set of pegs on a board with holes. A standard game fills the entire board with pegs except for the central hole. The objective is to empty the entire board except for a single peg in the central hole. To remove a peg, the player must move a peg orthogonally over an adjacent peg into a hole two positions away, then the jumped peg is removed from the board. If a legal move is no longer possible (i.e. only one peg remaining or none of the remaining pegs can jump over one another), the game ends. The game has been played in a variety of board shapes – the cross and triangular boards being the more popular options.

The genetic algorithm approach was inspired by the notion that genetic algorithms can be utilized to have an AI learn and improve its performance in a linear game. In 1983, Nils Aall Barricelli – who is a pioneer in artificial life research – simulated the evolution of the ability to play a simple game (the game itself was never specified).[1] This was in fact one of the earliest tests performed regarding artificial life. A more modern example would be SethBling's *MarI/O*, a machine learning program that can complete a level in the *Super Mario* games using neural networks and genetic algorithms.[2] To further enforce its authenticity, he cites the *NeuroEvolution of Augmenting Topologies (NEAT)* algorithm as his primary resource.[3] Like my peg board program,

[1] Nils Barricelli (1962), *Numerical testing of evolution theories*, https://link.springer.com/article/10.1007%2FBF01556771.
[2] SethBling (2015), *MarI/O – Machine Learning for Video Games*, https://www.youtube.com/watch?v=qv6UVOQ0F44.
[3] Kenneth Stanley and Risto Miikkulainen (2002), *Evolving Networks through Augmenting Topologies*, http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf.

MarI/O makes extensive use of gauging generations and fitness, with each individual being the program's performance in a particular level. His program yielded successful results, which just goes to show that genetic algorithms can be utilized successfully even in environments as complex as video games.

Some other interesting things were found regarding the research done specifically on peg solitaire. There is a known thorough analysis of the game which introduces a notion called the pagoda function. The **Pagoda function** is a function specifically designed around peg solitaire which is useful for showing the infeasibility of a given generalized peg solitaire problem. The solution for finding a pagoda function is formulated as a *linear programming problem* and is solvable in *polynomial* time.[4] [5] With enough development time, the pagoda function could have been a nice implementation in the Peg Solitaire learning machine, but time constraints this semester wouldn't allow it. Had the function been implemented, the fitness of an early sequence of moves could have been pre-determined, therefore saving a lot of computational power in the long run. In 1999, peg solitaire was discovered to be solvable on a computer using an *exhaustive search* through all possible variants, using symmetries, efficient storage of board constellations, and hashing.[6] Of course, this is likely far less efficient than implementing an optimal genetic algorithm, as is the case for most brute-forced methods. Lastly, it was found that a solution where the final peg arrives at the initial empty hole is not possible in a triangular board, assuming that hole is in one of the three central positions.

[4] Masashi Kiyomi and Tomomi Matsui (2001), *Integer Programming Based Algorithms for Peg Solitaire Problems*, https://link.springer.com/chapter/10.1007%2F3-540-45579-5_15.
[5]Jefferson, Angela Miguel and Ian Miguel (2006), *Modelling and Solving English Peg Solitaire*, https://hugues-talbot.github.io/files/Peg_Solitaire_1.pdf.
[6] Eichler; Jäger; Ludwig (1999), *Spielverderber, Solitaire mit dem Computer lösen* (in German), p. 218.

For the sake of simplicity and efficiency, a triangular board was chosen to be represented in the program. In this case, there are given six different ways in which a peg can be moved: left, right, up-left, up-right, down-left, and down-right. A single **position** on the board is represented as list containing the *row* (top to bottom), the *column* (left to right), and the '*state'* of that position (* = peg, o = no peg). The **board** itself is a list of *positions*, representing all fifteen positions on a triangular board. The **move** initiation is represented as a list containing the row and column of the peg's position, in addition to the direction in which the peg will move. The **sequence** of moves in a game, from start to finish, is represented as a list of *moves*.

In the genetic algorithm, each individual is represented as a sequence of moves. The mutation involves modifying a random index the sequence to be a different, but still valid, move. The remaining sequence is then modified in order for the rest of the sequence to be valid. The crossover takes a *mother* and *father* sequence of moves, selects a unique move from the father sequence, then replaces a random index of the mother with the chosen father move. Just like the mutation, the remaining sequence is again modified to give us a valid sequence. The program's performance gauged in two different ways: on method is to track the number of remaining pegs – a rather simple metric. The other, more interesting method is to track the average distance between the remaining pegs, giving us a more sophisticated metric. Finally, it should be noted that most of the genetic algorithm coding structure is based closely on Craig Graci's RBG genetic algorithm.

Overall, the project has yielded mixed results. On the downside, the program can take a very long time (up to several minutes) to produce results for just five generations. In terms of peg count, the machine doesn't appear to change much after five generations, resulting in values jumping up and down for the most part. Also, in what appears to be entirely random cases, a stack overflow may occur. Unfortunately, due to time constraints I do not have enough time to debug the amount of code I have to figure out exactly what the source of this issue is. The program may also run indefinitely. The upside is that the machine *does* appear to be improving when using peg *separation* as a fitness metric, assuming everything runs fine. The average fitness tends to show a gradual decrease in most cases, with a chance of fluctuating upwards.

Perhaps with a more efficient approach to the algorithm, the program could potentially run far better than it does currently. A faster program would allow for more testing at far higher populations, yielding even better results than what the current program has managed to produce.

Resources

Eichler; Jäger; Ludwig (1999). *Spielverderber, Solitaire mit dem Computer lösen* (in German). p. 218.

Jefferson, Angela Miguel and Ian Miguel (2006), *Modelling and Solving English Peg Solitaire*. https://hugues-talbot.github.io/files/Peg_Solitaire_1.pdf.

Kenneth Stanley and Risto Miikkulainen (2002). *Evolving Networks through Augmenting Topologies*. http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf.

Masashi Kiyomi and Tomomi Matsui (2001). *Integer Programming Based Algorithms for Peg Solitaire Problems*. https://link.springer.com/chapter/10.1007%2F3-540-45579-5_15.

Nils Barricelli (1962). *Numerical testing of evolution theories*. https://link.springer.com/article/10.1007%2FBF01556771.

SethBling (2015). *MarI/O – Machine Learning for Video Games*. https://www.youtube.com/watch?v=qv6UVOQ0F44.