Artificial Intelligence for 'Odd' Final Project Report

Maxim Gorshkov
260397155
COMP 424

April 22, 2013

1 Introduction of the Problem

The objective of this project was to formulate an AI for the game of Odd.[1]. The game uses a hexagonal grid on which two players take turns moving. At each turn the player may place a piece of any colour (black or white). This continues until all spots on the board have been filled. The object of the game is the formation of groups. Each group is constituted of 5 or more adjacent pieces of the same colour. At the end of the game, if there are an odd number of groups, the first player wins, otherwise, the second player wins. In the figures below, Figure 1 denotes an empty board, prior to the beginning of the game and Figure 2 denotes a completed game. In Figure 2, the first player would have won since there are 3 groups.

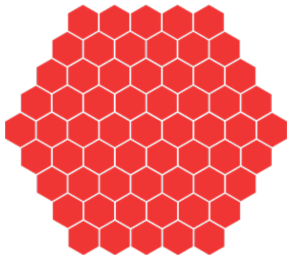
Figure 1: Empty Board

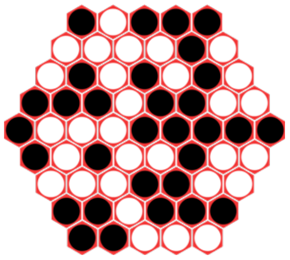
Figure 2: Completely Filled Board

2 Methods

2.1 Motivation: Approach and AI Growth

Upon discovering that the game of Odd is a perfect information game, I concluded that I need to use an algorithm that surveys the possible moves that a player can make and subsequently identifies which one could be the most optimal. Given parameters such as these, the most intuitive, albeit naive method was the Monte Carlo Simulation, implemented as described in class. The algorithm simulates a given amount of games for every possible step that could be taken at any given moment. My original Monte Carlo Simulation followed the process in Figure 3. I started with $N = 1$ as the number of roll outs. I noticed that this only lead to a very slight increase in performance over the random player. Just as arbitrarily, I chose $N = 150$ and noticed that after many games the MCS won every time.

To gather more information, I decided to do a large scale test. I defined perfect results as any branching factor that had no losses to the random player over a 50 game period. Simulating a few thousand games of various roll-outs, I produced a threshold for which factors produced perfect results versus those that did not. Figure 4 represents 2500 games ranging from $N = 0$ to $N = 100$ with intervals of $N = 2$. Every roll out had 50 games played against the random AI. I concluded that only within the branching factors $N < 30$ would you get any results that were not perfect. As such, I concluded that taking the branching factor to be $N = 100$ would be substantial to ensure that the AI won against the random player every time. In order to have a balance between running time (efficiency) and performance of the game (winning), I decided to implement the discrete roll-out value, rather than waiting for 5 seconds to go by at each move.

[1]Bentley, Nick. "One of my better games: Odd." Nick Bentley's Abstract Game Design Blog. N.p., 11 Feb. 2010. Web. 21 Apr. 2013. http://nickbentleygames.wordpress.com/2010/02/11/one-of-my-better-games-odd/

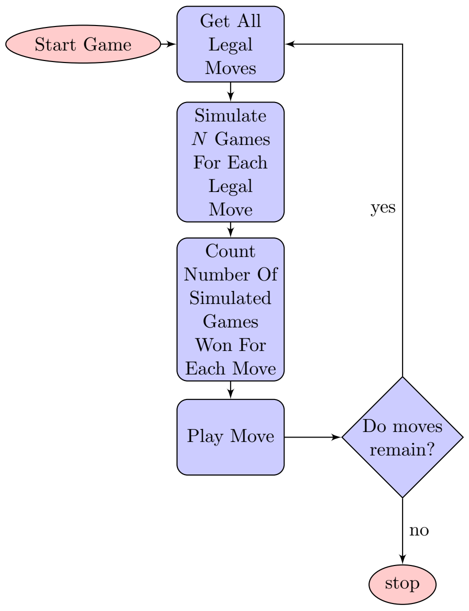
Start Game
Get All Legal Moves
Simulate N Games For Each Legal Move
Count Number Of Simulated Games Won For Each Move
Play Move
Do moves remain?
yes
no
stop

Figure 3: Monte Carlo Simulation

Although the AI was winning against the random player every game at $N = 100$, it didn't seem like it had any advantage over a simple brute force approach. At the beginning of the game, the AI took about $900ms$ to complete the first move and then dropped off towards the end (as in Figure 4). This means that there was a lot of computational time that was being wasted towards the end of the game. This lead me to think that rather than implementing the Monte Carlo Simulations with static roll-outs, I could use a method that would increase the computations made late in the game. This would both potentially add to the complexity of the AI, but also use most of the computational time. Progressive roll-outs will be discussed further in Section 2.2.

At this point, I did not have a good benchmark to use for my AI. Since I beat the random player with the static roll-outs, I would subsequently be able to beat the random player with the progressive roll-outs as well. My goal became simple: to develop a more complex AI that beats my simple Monte Carlo Simulation a majority (90% or more) of the time.

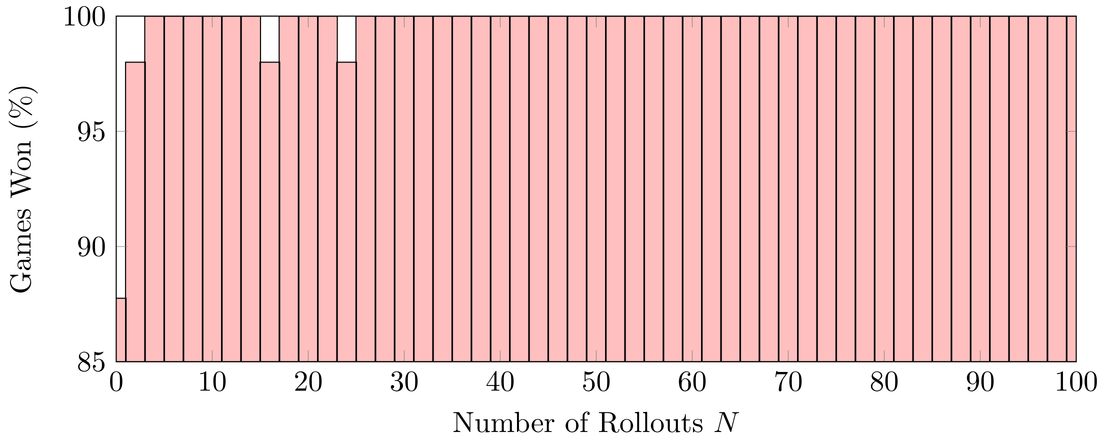

Figure 4: Fixed Roll-Out Large Scale Testing

2.2 Progressive Rollouts

Rather than having a static N value for the roll-outs, I implemented a changing roll-out factor which increased further into the game. Since we know how many moves were played and remaining, I defined the factor which changes for each turn to be as follows:

$$N_{factor} = N_{factor} + ValidMoves$$

As we get further into the game, we do not, however, increase the change in the roll out factor as to ensure that the turn does not time out. Using this modification alone, the more advanced AI now beat the Monte Carlo Simulation AI 144/200 times. By doing trials over various changes in the progressive roll-out factor, there was no tweaking that could be performed to make this value better. Although a significant difference, it did not meet the goal that I was working towards. By making my program more modular, I may have a smarter generation of the random moves within the Monte Carlo Simulation, and thus reach my goal.

2.3 Quasi-Random Numbers

While searching for relevant algorithms and implementations, I came across the notion of Quasi-Random numbers. Regular random number generators use a "Pseudo number" generation algorithm, which can be less efficient and present the numbers in a much less uniform order.[2] As such, I thought that the way that the Quasi-Random numbers generate the next move might be different, and thus better than a traditional random function. I implemented an algorithm similar to that embedded in Matlab.[3] It works as follows:

1. We start with a sequence of numbers. For example, let's use $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$.
2. We perform a **scramble** operation, in which we mix up the sequence. In my implementation I go through each of the values in the sequence and perform $swap(i, pSequenceSize - i)$. The above sequence is thus $[9, 1, 7, 3, 5, 4, 6, 2, 8, 0]$.
3. We now perform a **skip** operation, where we simply skip a predefined amount of characters from the beginning of the sequence. In my implementation the values that get skipped are $pSkipPercent \times pSequenceSize$ where $pSkipPercent$ is a randomly generated value between 0 and 1. The above sequence, given that $pSkipPercent = 0.25$ will skip 2 elements and will be $[1, 3, 4, 6, 2, 8, 0]$.
4. We now perform a **Leap** operation where we want to jump over a certain number of elements and ignore them. This is similar to the operation above except we ignore values in the middle of the sequence instead of those from the beginning. In my implementation, the values that are left after we perform leap are $pIgnorePcnt \times pSequenceSize$ where $pIgnorePcnt$ is a randomly generated value between 0 and 1. The above sequence, given that $pIgnorePcnt = 0.25$ will leap every 2 elements and will be $[3, 4, 2, 8]$.
5. Now, to return the sequence, say for the purpose of getting the next move, we randomly select one of the remaining elements to return. For example, we can return [2].

[2]Dutang, Christophe, and Diethelm Wuertz. "A Note on Random Number Generation." CRAN (2009): Web. 20 Apr. 2013.

[3]"Generating Quasi-Random Numbers - MATLAB & Simulink." MathWorks - MATLAB and Simulink for Technical Computing. N.p., n.d. Web. 21 Apr. 2013. http://www.mathworks.com/help/stats/generating-quasi-random-numbers.html.

By comparing 1,000,000 values from each of the regular util.Random package and the Quasi-Random generator that I created, it is clear that there is a different between the distribution of numbers. The results are depicted in Figure 5, below. We can see that the blue diamonds (Random) is much less uniformly distributed than the red squares (Quasi-Random)

However, with the integration of just Quasi-Random generation for all of the moves, there is barely a difference in performance over the Monte Carlo Simulation player. The Quasi-Random is almost on par with the other player and had 104/200 wins. It seemed like it was better during the first moves than it was for the full game.

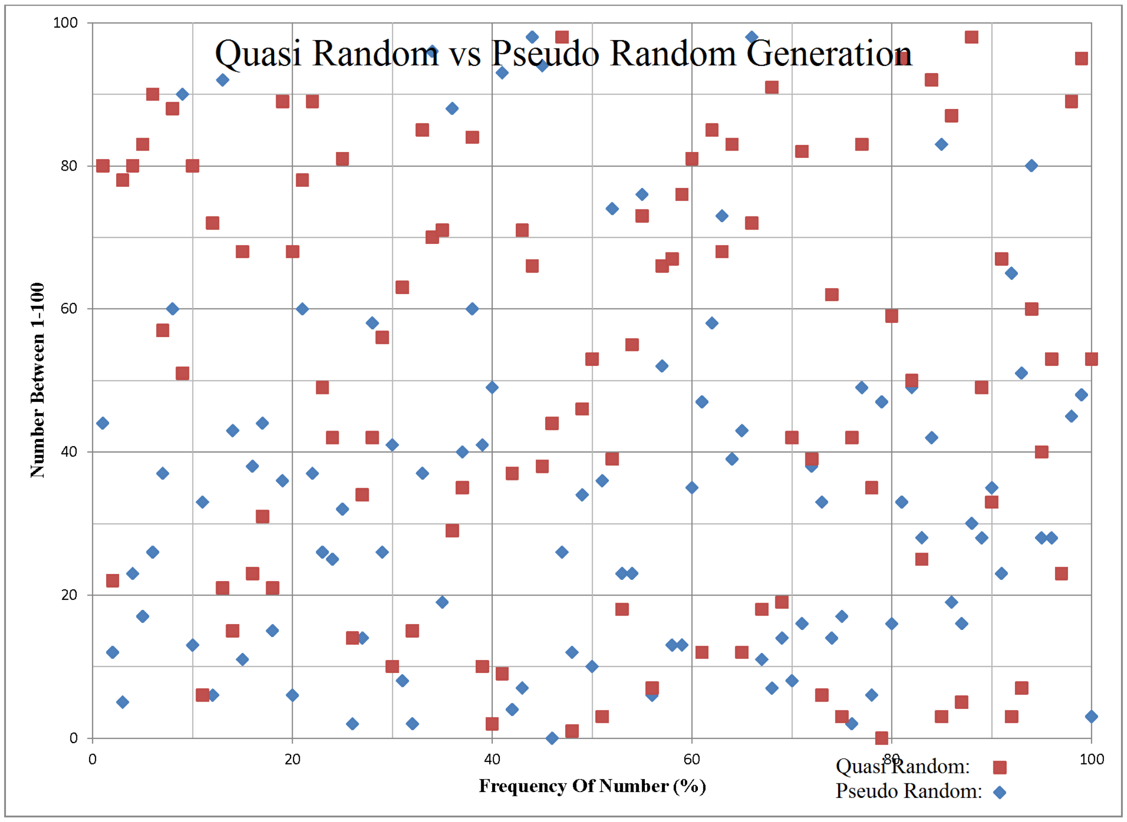

Figure 5: Quasi Random vs. Pseudo Random Generation

2.4 The Final AI

The next step was to try the Progressive Roll-out and Quasi-Random concepts together. By letting the AI have advantages of both the uniform random number generation and an increasingly better roll-out factor later in the game, it outperformed the simple Monte Carlo Simulation AI. Figure 6 shows the final approach to the AI.

With the power of the two concepts, I was able to beat the MCS in AI 99/100 trials. It made more sense to have the Quasi player for the opening moves of the game and then continue with a MCS later in the game with the progressive roll-outs. By changing the number of moves that the Quasi player gets and the factor by which the roll-outs increase, I was able to tweak the win-to-lose ratio.

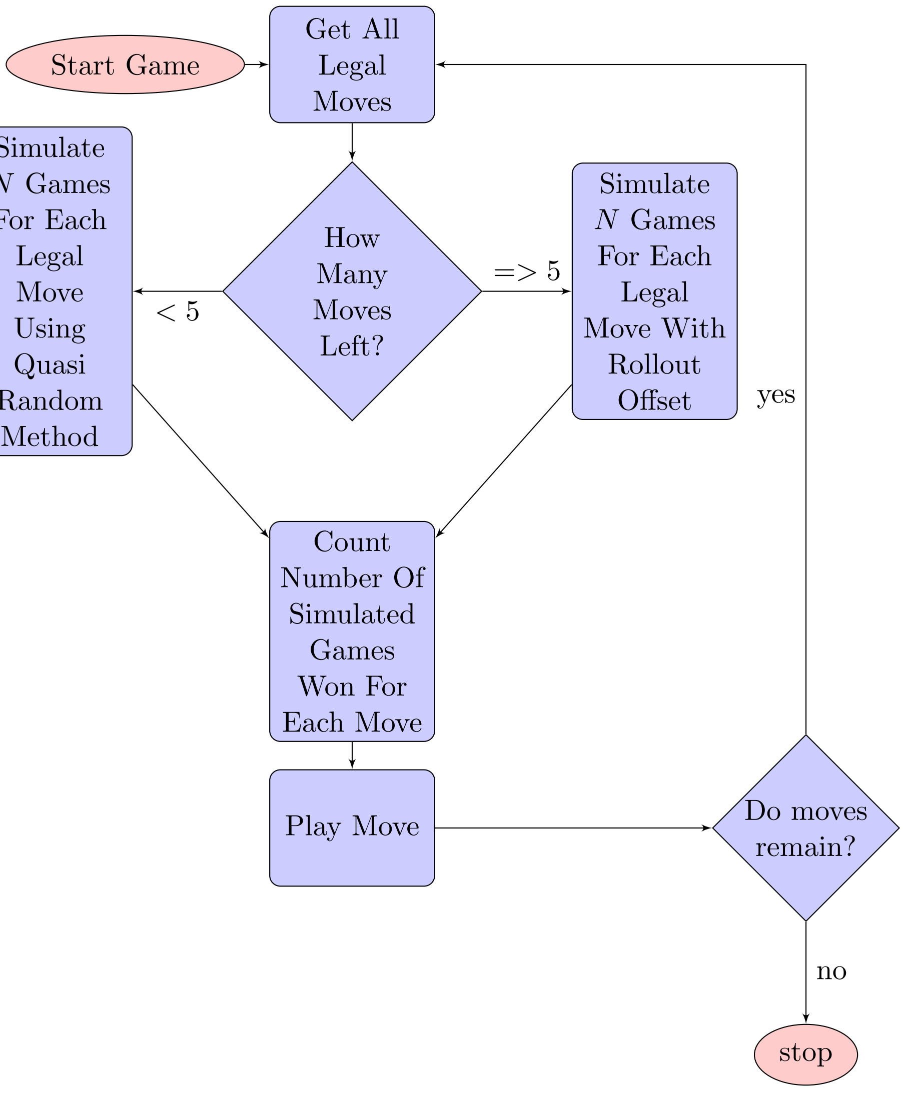
Start Game
Get All Legal Moves
How Many Moves Left?
< 5
=> 5
Simulate N Games For Each Legal Move Using Quasi Random Method
Simulate N Games For Each Legal Move With Rollout Offset
Count Number Of Simulated Games Won For Each Move
Play Move
Do moves remain?
yes
no
stop

Figure 6: AI Combining Progressive Roll-out And Quasi Randomness

3 Advantages of Approach

In general, the advantage of this approach is that not a lot information needs to be stored or known about the game beforehand. This means that the time complexity in running the AI will be relatively large: between $O((logN)^K N^{-1})$ and $O(N^{\frac{1}{2}})$, but that the space complexity will be nominal since we calculate what we need to know at each step. We don't need to have extra space for remembering the moves that were played, or going to be played past the upcoming one. By using a mixture of the types of random number generation, as well as the modularity, even if the Quasi player cannot find the best steps, the MCS might be able to take over and go on to win the game.

A great weakness to this approach is the need to recalculate the next movements at every step. This means that even though we may have already calculated the same branch multiple times, since we don't save full branches but just specific moves, we need to redo the calculation. Another weakness involves the fact that a similar algorithm which implements the full 5 seconds to do the simulations rather than a preset amount of roll-outs might be able to outperform this AI since it can consider the possible movements more thoroughly. If the AI is ported to 'Odd' but with more players and thus a larger board, although the AI would be able to complete the game in most cases, it might be possible that it would timeout since there would be more valid moves on the board. This would affect the progressive roll-out factor which could make the N value great enough to take more than 5 seconds to complete.

4 Improvements

Several improvements can be made to the approach in order to have the AI more competitive and potentially increase the performance with less computation. By keeping track of a whole branch, rather than just specific moves that are going to be played, not only can the AI more accurately predict the sequence of moves in advanced but it would also not need to recompute a large potion of what it does now. More pre-processing of data can be done before the next move from the AI as well. For example, if there are multiple threads running within the same AI, one thread could simulate what the moves of the other player might be (in a much better form than random). This way, the algorithm would not just maximize the winning rate by making groups but can also predict which groups the opponent may want to make, and thus block it. With the idea of multi-threading we can also imagine a way that while the opponent is taking the turn, the AI may continue to formulate the potential moves of the game and have a better idea of where to go after the opponent makes the turn.

Instead of using multi-threading, a popular approach to expand the quality of the algorithm would be to use boosting. With boosting and other machine learning algorithms, the performance of trees and other like AIs can be improved. With these learning algorithms, rather than having to predict the moves with a certain degree of uncertainty, the AI would be more mature after past games played, and thus, would be able to make decisions based on previous strategies.

5 Works Cited

Bentley, Nick. "One of my better games: Odd." Nick Bentley's Abstract Game Design Blog. N.p., 11 Feb. 2010. Web. 21 Apr. 2013. http://nickbentleygames.wordpress.com/2010/02/11/one-of-my-better-games-odd/

Dutang, Christophe, and Diethelm Wuertz. "A Note on Random Number Generation." CRAN (2009): Web. 20 Apr. 2013.

"Generating Quasi-Random Numbers - MATLAB & Simulink." MathWorks - MATLAB and Simulink for Technical Computing. N.p., n.d. Web. 21 Apr. 2013. http://www.mathworks.com/help/stats/generating-quasi-random-numbers.html.