A Distributed Genetic Algorithm for Learning Evaluation Parameters in a Strategy Game

18.337/6.338 Final Project

Gary M. Matthias

1. Introduction

The concept of parallel algorithms can be applied to a particular problem in artificial intelligence and game theory. In two-player strategy games such as chess, checkers, etc., computer programs usually give each board position a numerical value algorithmically determined a specific set of evaluation parameters. However, it is not always obvious to the developer which set of parameters would produce the strongest computer-based opponent, especially for less-known games (e.g. chess variants). For this reason, it is a common practice for game developers to tweak the evaluation parameters in an attempt to create the strongest possible automated opponent. This can be done in a manual, recursive way by testing the strongest current automated opponent with a "tweaked" version of itself to find out which of two opponents comes out victorious. However, it can be much more efficient if the process of determining the strongest possible opponent in a strategy game is carried out automatically. One way of discovering the optimal set of evaluation parameters is by means of a genetic algorithm.

1.1. Genetic Algorithms

Genetic algorithms, inspired by evolutionary biology, are an optimization method for discovering an approximate or exact solution to a specific problem. Each possible solution has its own "genetic code" or "genome," which usually consists of binary strings that represent a certain attribute, or "gene." The process of natural selection is simulated by having solutions compete against other solutions to survive into the next generation of solutions and pass their genetic content onto offspring. In this project, we would like to find the set of evaluation parameters that result in the strongest opponent, that is to say, the set of evaluation parameters that have the highest probability of winning, or play with the most optimal strategy.

1.2. Parallelism in Genetic Algorithms

Each round of this genetic algorithm consists of a tournament between all of the current competitors. In this system, we would like to carry out a round-robin tournament such that each competitor plays two matches against all other competitors, one playing as the first player and another playing as the second player. This results in a total of n*(n-1) games played each round, with an order of $O(n^2)$. In order to ensure a diverse population of competitors in each round, which is a precaution against to prevent local solutions, it is necessary to have a sufficiently large population, meaning that a significant number of games would have to be completed in each

round. Therefore, the tournament very easily becomes the most time-consuming portion of the algorithm, as each simulated game takes an extensive amount of computation time due to the tree search algorithms that are ubiquitous in game theory. The main focus in this project will be parallelizing the simulated game play that occurs in the natural selection segment of this genetic algorithm.

2. The 5,5,4 Game (Tic-Tac-Toe Variant)

The strategy game used for this project will the 5,5,4-game., a 5x5 variant of Tic-Tac-Toe, where the first player to obtain four consecutive pieces horizontally, vertically, or diagonally wins the game. The 5,5,4-game belongs to a class of games called m,n,k-games in which two players trade moves against each other on an mxn board, trying to connect k consecutive pieces. Theoretically, the second player cannot have a winning strategy due to the advantage that the first player obtains by moving first, therefore an m,n,k-game can only be a win for the first player or a draw if both players utilize a perfect strategy. The 5,5,4-game, like the more popular typical 3,3,3-game is a draw with perfect play. An ideal solution for a set of evaluation parameters should result in a player that can achieve at least a draw under any circumstances.

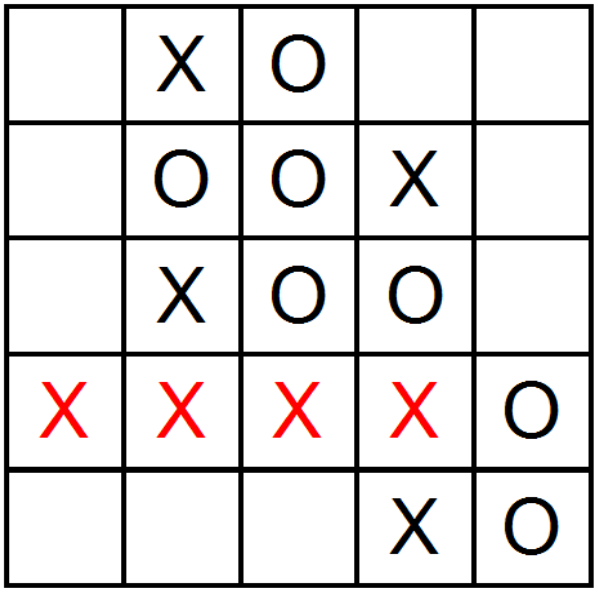

Figure 1: The 5,5,4-game

3. Game Engine

The simulation engine for game play required an internal implementation of the game board and possible moves. Board positions are given a specific value based on a each competitor's unique set of evaluation parameters; the best move is chosen based on a minimax tree search with alpha-beta pruning.

3.1. Game Representation

The board is represented by the following attributes:

`board_boxes`: status of the board, 5x5 matrix, {1,0,-1}
`board_turn`: player who moves next, {1,0,-1}
`board_winner`: player who has won, {1,0,-1}
`board_emptyboxes`: number of remaining boxes, {0,1,...,25}

Possible moves are represented as follows:

`move_x`: x-coordinate of move, {1,2,3,4,5}
`move_y`: y-coordinate of move, {1,2,3,4,5}
`move_player`: player making move, {1,-1}

3.2. Evaluation Parameters

Each competitor has a set of evaluation parameters by which a numerical score is given to each board position.

Positional parameters determine the weight given to each player when specific points on the board are played. Due to the horizontal, vertical, and diagonal symmetry of the board, only 6 of these attributes are needed for each player, resulting in a total of 12 8-bit positional parameters.

`sq11x, sq12x, sq13x, sq22x, sq23x, sq33x`: First player, {0,1,...,255}
`sq11o, sq12o, sq13o, sq22o, sq23o, sq33o`: Second player, {0,1,...,255}

11	12	13	12	11
12	22	23	22	12
13	23	33	23	13
12	22	23	22	12
11	12	13	12	11

Figure 2: Mapping of positional parameters to board locations

Streak parameters determine the weight given to each player for obtaining one, two, or three consecutive pieces horizontally, vertically, or diagonally. A special weight not determined by evaluation parameters is given when a player obtains four consecutive pieces. There are 12 of these 8-bit streak parameters.

`h1x, h2x, h3x`: Horizontal and vertical streaks for first player, {0,1,...,255}
`h1o, h2o, h3o`: Horizontal and vertical streaks for second player, {0,1,...,255}
`d1x, d2x, d3x`: Diagonal streak parameters for first player, {0,1,...,255}
`d1o, d2o, d3o`: Diagonal streak parameters for second player, {0,1,...,255}

8-bit scale factors determine how much emphasis is placed on positional and streak parameters

`scale_sq`: Scale factor for positional parameters, {0,1,...,255}
`scale_cons`: Scale factor for streak parameters, {0,1,...,255}

Finally, the 3-bit `search_depth`, {0,1,...,7}, determines the maximum number of future moves ahead in the tree search that a player can view in order to decide on the best move.

All parameters are represented as integers then later converted to binary strings for genetic operations.

3.3. Alpha-Beta Search

A minimax search with alpha-beta pruning is initiated by each player, using its specific evaluation parameters, to choose the best move by maximizing the minimum future gain. Some players will be able to search farther ahead than others. However, modifications were made to make the depth more shallow earlier in the game since the tree of possible positions for a given depth is much larger earlier in the game. This change was made to shorten the amount of time per game. However, since the number of possible positions for a given depth gets much smaller later in the game, players are able to search farther ahead as the game progresses. The maximum search depth is a deterministic function based on the number of remaining squares on the board.

Remaining squares	Search depth
20 to 25	2
13 to 19	3
10 to 12	4
9	5
Less than 8	7

Figure 3: Maximum search depth based on the number of remaining squares

4. Genetic Operations

The genetic algorithm for optimizing the evaluation parameters involves a series of genetic operations in each stage of the algorithm.

4.1. Natural Selection

Natural selection in the algorithm is carried out by playing all competitors against each other in a round-robin tournament. The tournament is designed such that each player competes against all other players twice—once as the first player, and another time as the second player. This results in a total of n*(n-1) games played during each round.

At the end of the round, fitness scores are calculated for each competitor based on the number of wins, losses, and draws accumulated during the course of the round. The fitness score formula was chosen to be:

fitness score = 5*wins + 1*draws - 4*losses

At the end of the round, those with nonpositive fitness scores are removed from the population. Thus, this fitness score is designed such that a player with an equal number of losses and wins (and no draws) has a positive fitness score, and also such that a player with two draws has more fitness than a player with one win and one loss.

4.2. Reproduction

At the end of natural selection stage of the algorithm, those competitors with nonpositive fitness scores are taken out of the population due to their poor performance. The empty spots left behind by those competitors that failed to survive the round are occupied by new competitors who are created by combining the genetic content of surviving competitors. For each empty spot in the pool of competitors, two parents are randomly chosen from the surviving competitors with probabilities proportional to their fitness scores. Then, each bit of the genetic content of the offspring is randomly chosen from one of the parents with a 0.5 probability of acquiring the bit from each parent.

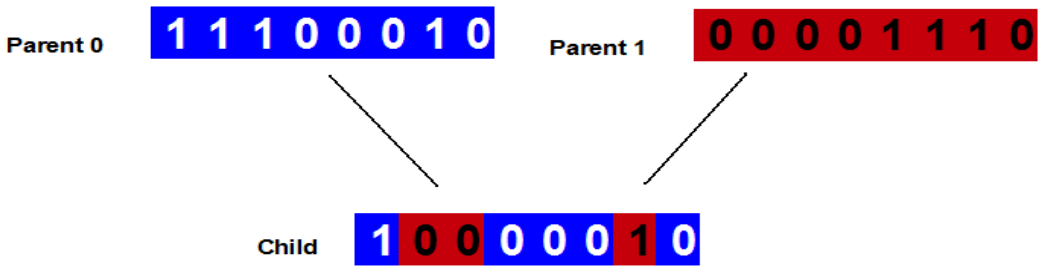

Figure 4: Reproduction of two parents

4.3. Mutation

After the population has been filled with children, all competitors go through a mutation stage where each bit in the evaluation parameters has a chance of being flipped. The rate of mutation per bit was chosen to be 0.01.

Figure 5: Mutation of a gene

4.4. Convergence

Convergence was not considered for this project though a reasonable proposal for convergence would be when every match results in a draw for several rounds. This is a reasonable criterion for convergence because the 5,5,4-game is a theoretical draw, so if every game ends in a draw, this could be indicative of a near-optimal strategy. However, there is a possibility of discovering local maxima. The population size and mutation rate need to be properly chosen to prevent this.

5. Serial vs. Parallel

Since the natural selection segment of the algorithm is the most computationally intensive, this will be the benchmark to compare serial and parallel performance. These results were obtained by running Star-P with 8 processes with beowulf.csail.mit.edu as the client and starp.csail.mit.edu as the server. A specific number of identical games were played in serial and then in parallel. The result was that as the number of matches increases the ratio between parallel running time and serial running time decreases. This indicates that there would be a substantial speed increase in a round-robin tournament of a much larger size when running in parallel rather than in serial

	n = 8	n = 16	n = 32
Parallel	19.33 s	36.08 s	67.29 s
Serial	27.26 s	77.60 s	178.45 s
Ratio	0.71	0.46	0.38

Figure 6: Parallel vs. serial running time results for a series of n matches

6. Conclusions

The simulated game play has been parallelized to shorten the running time of the natural selection segment of a genetic algorithm. This will expedite the process of learning optimal parameters for playing the 5,5,4-game.