Submitted: *In partial fulfillment of the requirements of*
CS 4701: Practicum in Artificial Intelligence

A Genetic Programming Approach to Modified Chinese Checkers

Anirvan Mukherjee
Computer Science, 2012
am767@cornell.edu

Aperahama Parangi
Computer Science, 2013
alp75@cornell.edu

Peter Bailey
Mathematics, 2013
pwb53@cornell.edu

– December 4, 2011 –

ABSTRACT

We present an approach for evolving an algorithm to play a two-player variant of Chinese Checkers. Our approach uses minimax search to a depth of three turns, along with α-β pruning (including best-first search-node expansion), to evaluate a board state. We evolve the heuristic with which the board state is evaluated using Genetic programming. Over the course of the semester, we explored many patterns for evolution, and present our final approach in this paper, as well as motivate some of the design decisions we took along the way. Key features of our approach include relative ranking of heuristics against each other (and fixed population units) using a PageRank variant, recursive mutation and cross-breeding, inter-generation fitness estimate comparison using a beam-search inspired method for identifying a useful benchmark heuristic, as well as fitness landscape analysis using fitness variance and Kendall's τ distribution. The latter two are methods of analysis that motivate our ultimate incremental evolution approach, in which we do not persist successful parents and maintain a single major population segment. We present samples of our motivating data in support of our approach, and lay out avenues for future development of our project.

I. INTRODUCTION

Chinese checkers is a rich and complex game of strategy that poses an interesting problem from an artificial intelligence standpoint. With a game tree too large for exhaustive minimax search, the creation of an intelligent player can be reduced to the formulation of an intelligent static evaluation function, or heuristic. Our approach to this problem involves the use of genetic programming to evolve a good heuristic. We draw heuristics from a space of semantic tree of metrics, which return numeric statistics regarding the current the board state, in conjunction with arithmetic operations. Over the course of the semester, we experimented with several evolutionary techniques, including steady-state vs incremental models, population segmenting, killing parents each generation versus persisting them, individual probabilities (e.g. crossover and mutation probabilities), and many other factors. This paper details our approach as it currently stands, its architecture, and its methodology. We motivate our work using practical considerations, research, and data, and present avenues for future development.

II. PROBLEM DEFINITION

The objective of this project is to develop artificial intelligence to play a modified version of Chinese Checkers, which is able to play, based on any symmetric initialization of the board, against an opponent using a maximum total of 10 minutes of computation time. Valid methods include any artificial intelligence techniques that are self-contained (e.g. don't require connecting to the web), are not unduly defensive (e.g. keeping all players at home to prevent other players from winning), and refraining from using system resources during the other player's move. It is given that, should either player run out of computation time, their subsequent moves will be determined by a provided GreedyAI which myopically picks the move that optimizes immediate forward progress.

The player is to adhere to the bindings of the given Java game server, which launches the player in its own process and communicates with the player by calling it via a specified interface. In particular, to make a move, we write the coordinates of the marble to be moved, its desired destination, and the location of a grey marble to drop into standard output, space-separated. The returned inputs through standard input are a single status value 1 if the player wins, -1 if the player loses, -2 if the player committed an error, -3 if the player ran out of time, and elsewise 0 if a valid move was returned and the player must wait for a response from the opponent. After the opponent's move, the server returns, via standard input, another status value (as before), followed by the remaining times in milliseconds of both players, followed by the opponent's move in the same format.

III. METHOD

III.1. Overview

At a top level, our algorithm uses minimax search to a depth of three turns, using α-β pruning with best-first expansion of nodes (this has been a historically popular approach for high-branching-factor board games – see [3] Samuel, 1967). The player performs the move that results in the highest depth-three minimax board value. Evaluation of a board state is performed using a heuristic, which is used both for ranking possible moves as well as in the best-first expansion of nodes. We used genetic programming to develop a useful heuristic to measure the board state with.

III.2. Notes on Minimax

A depth of three turns was chosen because of tractability concerns (since the branching factor of the game is very high) and because we found an odd number of turns to be desirable (having leaf nodes consisting of the opponent evaluating the board state using the player's heuristic proved to be too unstable).

The heuristic is taken symmetrically – e.g. a board state is evaluated from the point of view of the evaluating player. The symmetry assumption is valid because the termination rule of the game is symmetric and because the game is memoryless. Note that the evaluating player is always the player who is (either actually or, in the case of minimax, hypothetically, making the next move).

III.3. Heuristic Elements

Heuristics are taken as functions that map a board state to a float, and are represented as a semantic tree. There are two major types of components in a heuristic: metrics, and functions.

Metrics are the leaf nodes in the semantic tree, and map a board state to a float, on the basis of the evaluating player. The metrics we used were:

1. **Constant:** Evaluates to a constant regardless of the board state.
2. **Individual cells:** Returns whether a board cell is the evaluating player, the opposing player, an empty cell, or a blocked cell.
3. **Maximum forward progress:** The maximum forward position among all of the evaluating players' pieces.
4. **Minimum forward progress:** Defined likewise.
5. **Total forward progress:** The sum of the forward progress of all the evaluating player's pieces.
6. **Variance in forward progress:** Calculated using the usual variance formula.
7. **Mean radial position:** The average radial distance from the middle of the board of all of the evaluating player's pieces.
8. **Connected components:** The number of connected components among the evaluating player's pieces. Pieces are regarded as adjacent if each is in one of the six adjacent cells of the other. The notion of a connected component is the same as the graph-theoretic notion.

Note that constant metrics were generated separately from the rest for efficiency, but are still leaf nodes.

The function components we used were the four arithmetic operations, exponentiation, min, and max. This set of functions can express any piecewise-continuous function, including step functions (useful for Boolean conditions).

III.4. Genetic Programming Overview

We used Genetic Programming to evolve a heuristic of the prescribed format. Major considerations revolved around construction of the alphabet; performing of mutations, crossovers, and reproduction; and evaluation of fitness. Each of these is described in the following subsections.

At a top level, our approach is to use incremental, “hard” generations (rather than steady-state evolution), with population segmentation (e.g. the island model – see [1] Floreano and Mattiussi, 2008). Each generation, every player plays at least a set prescribed number of games (which we varied), most (but not necessarily all) of which are within the player’s population segment. A modified version of PageRank, adjusted to penalize players with high runtime, is used to rank the players, and poorly ranked players are killed off. The remaining players are mutated and crossbred to create the next generation. Reference players, including the provided GreedyAI, our developed AlphaBetaAI, and at this stage, EigenBot, are included in the population as well and affect players’ ranks, but are not killed off or otherwise maintained by the updater.

Since our latest restart, we’ve run 285 generations of evolution, with a kill-parents policy and a population slightly over 100. Note that, since our latest restart, we have not segmented the population.

III.5. Genetic Alphabet

Our heuristic ‘genomes’ are expressed directly in the form of a function (e.g. semantic tree, though easily expressible serially in prefix notation), and are interpreted literally (e.g. there is no complex mapping from genotypes to phenotypes). Thus, the elements of our alphabet are simply the metrics and functions present in the final semantic tree.

The goal was to make a lean but sufficient alphabet (e.g. with high expressive power), as suggested in *A Field Guide to Genetic Programming* ([2] Riccardo, Langdon, McPhee, and Koza, 2008). Our alphabet supports arbitrary numerical functions (via constants and arithmetic operations, due to Taylor expansion), arbitrary branching (due to min and max, which can create step functions), and arbitrary aggregate metrics (due to inclusion of individual cells as valid metrics). However, in order to speed up progress, several sensible board metrics are also included, as noted in the Heuristic Elements section.

We paid particular attention to alphabet closure (the combination of type consistency and evaluation safety), as defined in [2] Riccardo, Langdon, McPhee, and Koza (2008). Type consistency is achieved by requiring that the output of all functions and metrics, as well as all inputs of all functions, be floats. This does not materially limit our expressive power (e.g. we did not see a need for recursive evaluation of an individual board state, or other complex computational operations), but allows us flexibility in crossover and mutation, since we may crossover or mutate at any node without excessive type-checking or type-related constraints.

We found that evaluation safety is only an issue in certain arithmetic operations (e.g. divide by zero, raise a negative constant to a fractional power, etc.). Rather than pre-screening for risks, we apply the game server's policy of assigning a loss to population units that result in an error at runtime. This loss in turn hampers the unit's fitness, eventually removing it from the population.

III.6. Mutation, Crossover, and Reproduction

We modify functions in two ways: mutations and crossovers, both of which are performed recursively on the semantic tree in a similar way.

Mutations operate on single units, beginning at their root, performing a mutation with some probability, and then recursing into the children's subtrees. We use three varieties of mutations: insertions, deletions, and modifications, each of which occurs with a fixed probability. The insertion and deletion rates are balanced to prevent tree-size bloat in heuristics.

- Insertions replace the current node with a new function node, making the current node one of its children. The other child is either a metric or another insertion.
- Deletions delete the current node, replacing it with any one of its children.
- Modifications swap the function in the current node with another function (or metric with a metric).

Due to the structure of our alphabet, there is no need to specially check for type consistency or evaluation safety.

Crossovers are performed similarly. They begin with two trees, considering their roots. At each level, we pick one of the children to persist, and then, if the chosen node is a function node, recurse into the corresponding subtrees of each. In the case that one subtree is empty (e.g. if we picked a function node in one tree over a metric in the other tree), we persist the entire remaining subtree.

The combination of crossovers with mutations allows arbitrary combinations of any two semantic trees into a new semantic tree (all of which are valid), encouraging diversity.

III.7. Fitness and Ranking

We were faced with several challenges in ranking population units. In particular, assigning an "absolute" fitness is difficult, and picking the wrong absolute fitness metric risks disaster. Instead our generation selection mechanism focuses on ranking individuals based on their performance against each other. Running a round-robin for ranking is too slow on a large population, so we devised an alternate approach to relative ranking based on PageRank, described below. This is similar to the relative-ranking mechanism prescribed for sparsely played games in coevolution by [6] Reisinger, Bahceci, Karpov, Miikkulainen, (2007) (the authors of this paper do not penalize runtime, and calculate a one-iteration PageRank approximation with teleportation). We also developed an instrumental benchmarking method for evaluating fitness between generations for the purpose of tracking the fitness of our population – this is detailed in section VI.

We use a modified version of PageRank to attain an easy, approximate rank of players within a generation. Under this scheme, each player plays a fixed number of games (which we experimentally tuned to 6 as a compromise between runtime and accuracy), and "links" to all players it loses to. Players who win all games are treated as 'dead-ends', and link to every other page. There is no teleportation used. A player's relative fitness is a combination of its PageRank as well as a long runtime penalty (e.g. to penalize heuristics that quickly default to GreedyAI). At the end of all games in a generation, we calculate the PageRanks, penalties, and overall fitness of each population unit, and iteratively draw the best units until we reach our target size for the next generation. These units are then crossed over and mutated to create the new generation. We experimented both with persisting successful parents and with killing them off, and ended up choosing the former strategy for diversity considerations (see section VI).

While the above approach is crucial in comparing players within the same generation and building the next generation, we also needed a benchmark for each generation so that players could be compared between generations. This proved instrumental in our evaluating changes in population fitness over time. We discuss this methodology in section VI.

IV. Related Work

Samuel (1967) prescribes intelligent α-β pruning in conjunction with a good heuristic as an effective approach to Chinese Checkers, and details its use in high-branching-factor gameplay. His use of α-β search is exactly what we use. However, while he lays out a machine learning method to develop a polynomial heuristic model, we use genetic programming to evolve a much more expressive heuristic. However, Samuel also lays out an additional forward-pruning technique, which we have not implemented.

Reisinger, Bahceci, Karpov, Miikkulainen (2007) coevolve players for various games (Tic-Tac-Toe, Connect Four, and Chinese Checkers, among others). Their approach segments the population into two groups and has each player play the best players in the opposite segment. Their approach bears a number of similarities with our approach – most notably, their ranking algorithm uses a similar notion of estimating relative fitness based on sparse games (see III.7). Two key differences are between their population segmentation scheme (our segmentation followed the island model with a leak factor), and in their benchmarking (we benchmarked against GreedyAI and AlphaBetaAI, whereas they benchmarked against a random move algorithm).

V. SYSTEM ARCHITECTURE AND IMPLEMENTATION

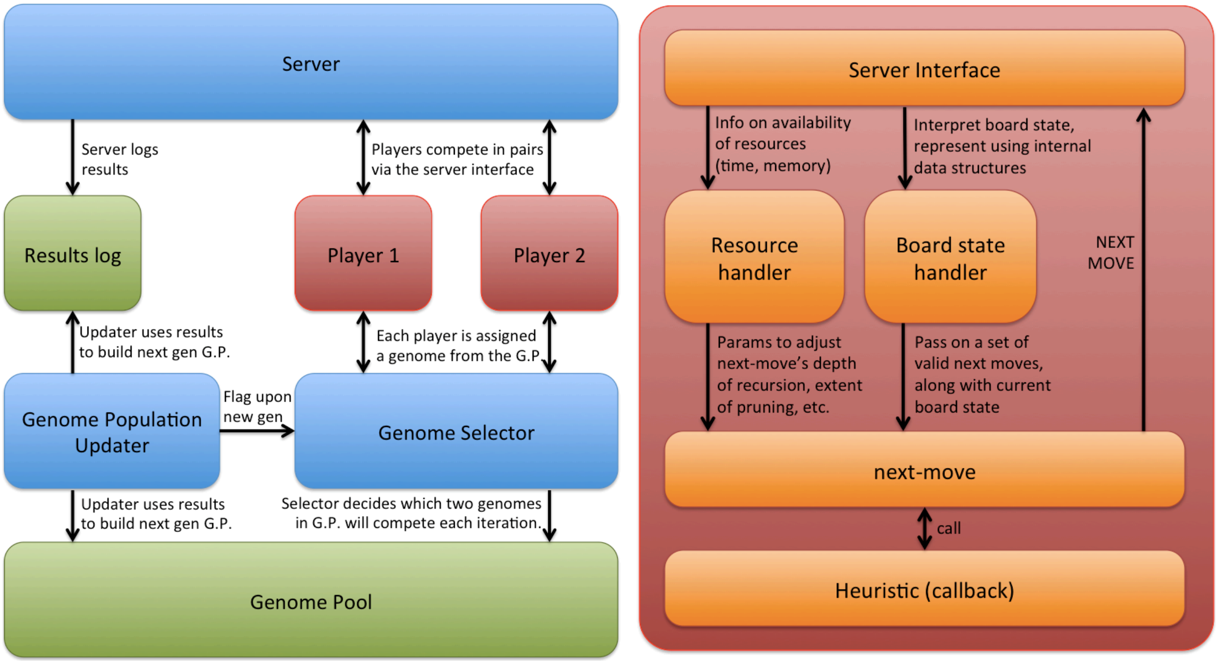
An overview of the major independent components of our system is shown below. Resources are shown in green, players in red, and logical modules (or black-boxes containing sub-modules, in the case of the server) in blue. The goal of our architecture is to allow players to be separated from our evolution system and uploaded to the 4701 server. Thus, our evolution system, gene pool, etc. are designed to sit "on top of" the base code, interacting only by means of reading from a results log file, and building instances of players to compete. This system is closed – no outside data is incorporated.

Interactions between all major elements are facilitated by a controller class gController, which facilitates the following process:

- Draw pairs of units in a generation from the gene pool
- Play them against each other using the GameServer (left unmodified)
- Log the results
- At the end of the generation, compute penalized PageRanks of each unit
- Kill off units with poor penalized PageRank (poor relative fitness)
- Mutate and cross-over remaining units to create the next generation

Note that the external controller approach allows us to leave the server code unmodified, resulting in less debugging and more modular code.

Below to the left is an overview of our entire system. To the right are details of the logic and components within a player.



Each unit takes the form of a Java class (extends a Player), and contains the basic code to interface with the server and ingest the board state (board state handler), as well as the think function (shown above as next-move). The think function invokes minimax search, which in turn calls the heuristic of the player. When players are evolved, the heuristic is parsed from Java code into a semantic tree, which is mutated and/or crossed-over, as required, and then written into a new Java source file. This gives us a standalone file representing a player.

VI. Experimental Evaluation

VI.1. Methodology

VI.9.a Overview

Our primary goal was to produce a population with high absolute fitness. Because we don't have an exact measure for absolute fitness (the problem of quantifying fitness to a fine resolution is intractable), our secondary goals were to effectively estimate *relative* fitness for evolution, to produce a population with an optimal fitness landscape (e.g. high spread, high diversity), and to effectively estimate absolute fitness between generations in order to evaluate and improve our evolutionary parameters and policies. In particular, we want both a high variance in fitness within the population, as well as a suitably high underlying diversity (these were our dependent variables). This methodology is supported by [5] Schmidt and Lipson (2008).

Tuning the parameters of our evolution (our independent variables) to optimize these was an important part of our project – in particular, it inspired key decisions such as whether or not to persist parents in the population after they reproduce, and how large of a population to maintain. Our methods for evaluating the fitness landscape for fitness spread and diversity are outlined in the following two sub-subsections.

VI.9.b Inter-Generation Fitness Estimation

We needed to approximate the fitness of a heuristic independent of the population itself, which is important for evaluating the fitness variance as well as comparing fitness between generations. In order to do this, we used a beam-search inspired method to find an "authoritative" heuristic that we could compare other heuristics to. This certainly does not impose a wholly accurate absolute ordering on the space of heuristics, but is sufficient for demonstrating fitness variance and diversity, as well as retrospectively benchmarking the fitness of older generations against the current generation. From this perspective, we treat heuristics as ranking algorithms that rank a set of board states. Our approach is summarized below:

1. Build a large set of N random board states (we use 50 for the analysis below).
2. For each heuristic, rank the N board states using a search depth of three.
3. Pick k (we used 5) well-ranked heuristics from the current generation. Let these be $x_{01} \ldots x_{0k}$.
4. Iteratively, let $x_{(i+1)j}$ be the heuristic which, run at depth 3 minimax, most closely matches x_{ij} run at depth 5 minimax.
5. Repeat until beams converge.
6. Majority vote of these is a good benchmark when run at depth 5 minimax.

The measure of closeness we used was Kendall's τ coefficient, a value between 0 and 1 which compares how closely two rankings are correlated. This value is frequently used as a measure of ranking closeness, and as a method of evaluating the fitness of a ranking algorithm against a benchmark – for example, [4] Schmidt and Lipson (2010) use this metric (albeit a scaled version) to evolve fitness predictors.

Once this authoritative benchmark is established (note that the benchmark is always run to depth 5), we estimate the fitness of any heuristic by how closely that heuristic's rankings of the N board states matches those of the benchmark. This misrepresents the fitness of good heuristics (since the fitness of any heuristic is represented as something that does not exceed the fitness of the benchmark), but works well for characterizing the fitness of heuristics that are not quite as good. We note the importance of guaranteeing that the benchmark heuristic used is of some minimum quality. See VI.2 for our results.

VI.9.c Diversity

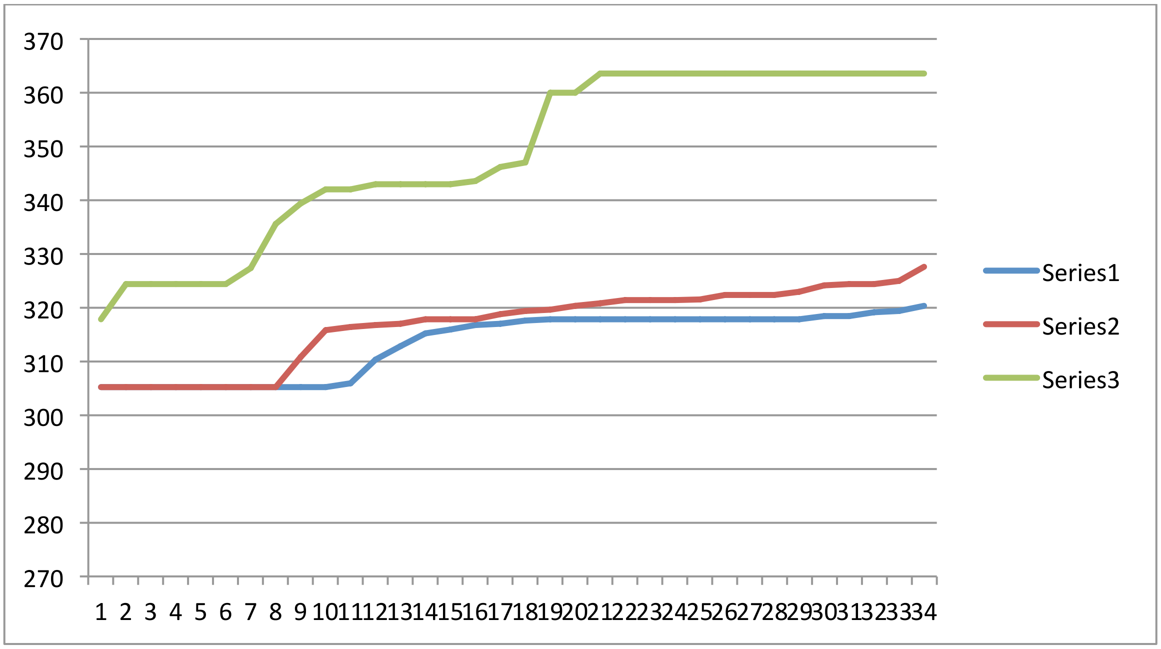
We use two measures of diversity within the population: variance in fitness, and Kendall's τ coefficient distribution between rankings. Fitness variance is calculated using fitness estimators from VI.9.a, since PageRank scores do not indicate variance in absolute fitness. Kendall's τ coefficient, is also calculated between each heuristic's depth 3 minimax ranking of a set of board states – a distribution of τ coefficients with greater weight towards middle and middle-low coefficients shows greater diversity. Note that coefficients near zero are dominated by noise from very poor heuristics, which is uninteresting. See VI.2 for our results.

VI.2. Results

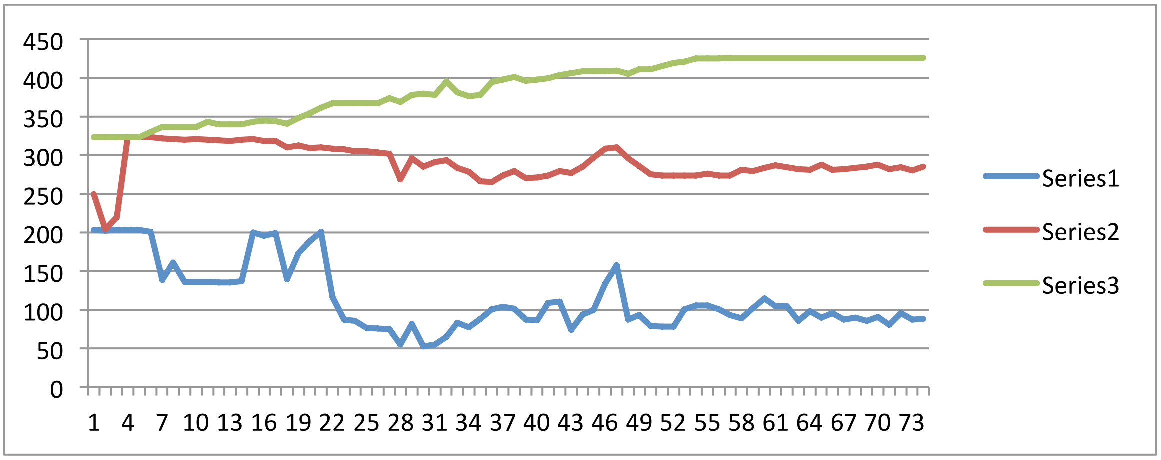
Some representative samples of our data are given below. The data below motivates our decision to kill off parents at the end of generations, rather than persisting them into the next generation and allowing them to compete against their children. This was one of the major decisions we had to make – the latter guarantees us nearly monotonically increasing fitness, but we believed the former is more suitable due to higher diversity, and our data supports that. Both are considered valid approaches in the literature ([1] Floreano and Mattiussi, 2008).

In the graphs below, Series1 represents the 50th percentile of fitness, Series2 represents the 75th percentile of fitness, and Series3 represents the maximum fitness. The horizontal axis represents number of generations passed, and the vertical axis is 500 minus the insertion-sort distance between the produced ranking of a set of board states and an authoritative ranking.

Fitness over time in incremental evolution with parents persisted

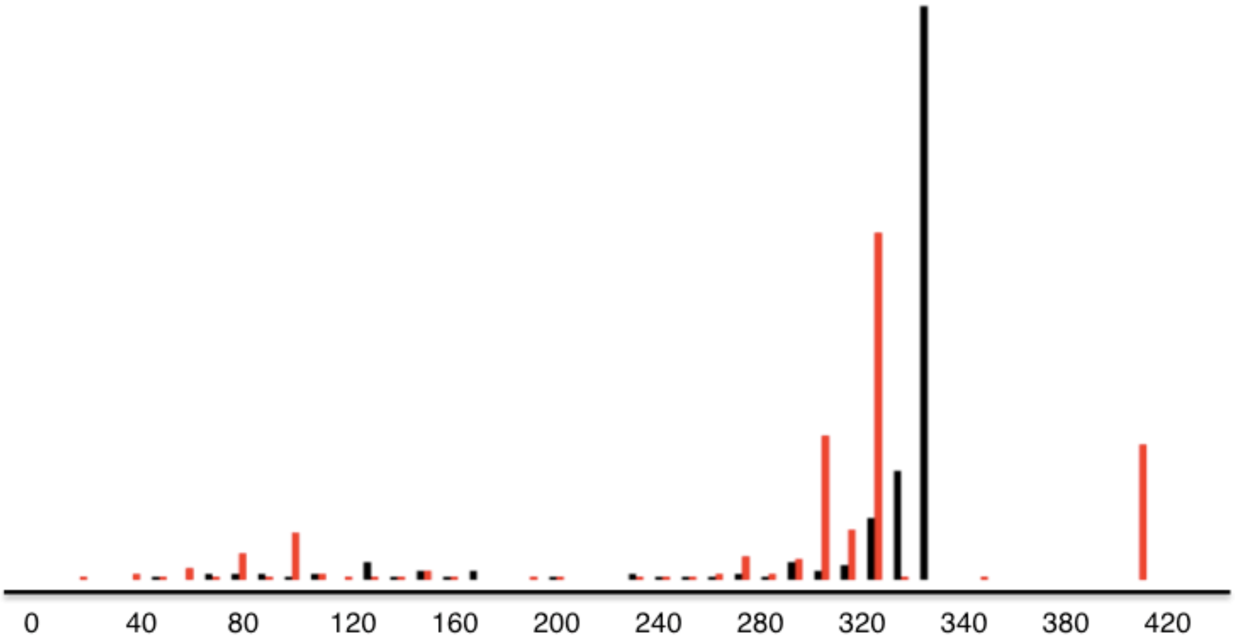


Fitness over time in incremental evolution with parents not persisted



Note that, by using incremental evolution *without* persisting parents, the maximum fitness increases (recall that this measure is relative to a benchmark, so the plateau effect near the end is artificial), even though the overall fitness is no longer monotone and median population member gets considerably worse!

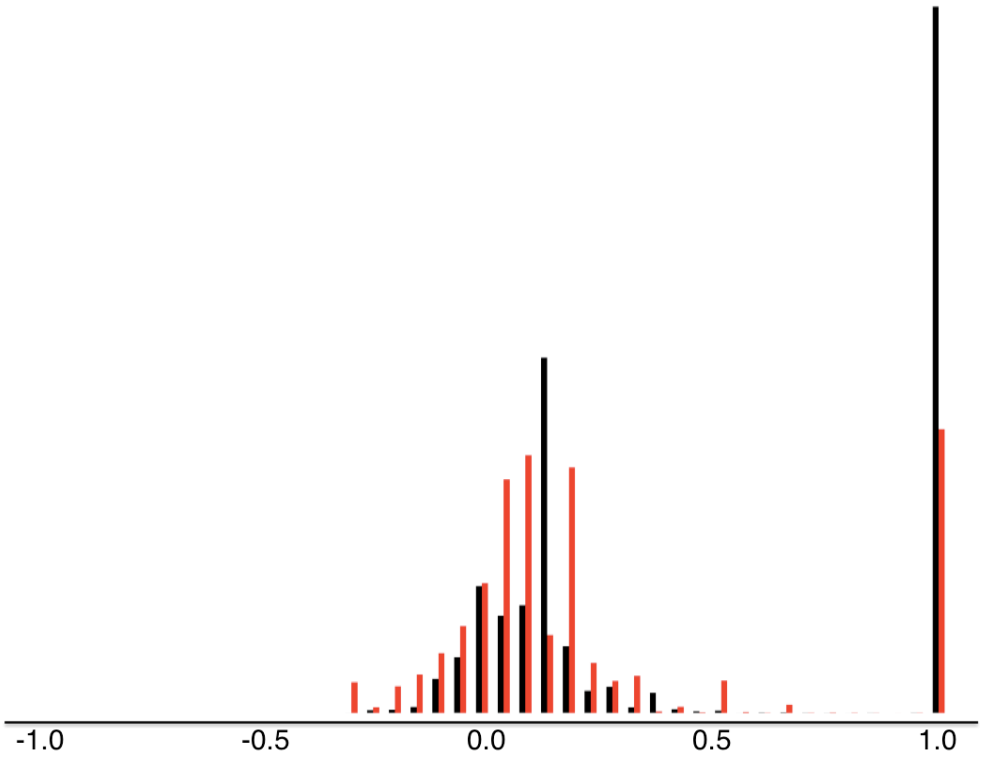
Benchmarked fitness-estimate distribution for both policies



The histogram above shows, side-by-side, a snapshot of another pair of populations evolved incrementally, both to 200 generations, with the red bars expressing the fitness distribution of the population with a kill-parents policy, and the blue bars expressing the fitness distribution of the population in which parents may be persisted. Note that, while the 'red' population has many more poor population units, the fitness spread is much higher (a standard deviation of 93.7336, versus a standard deviation of 62.3973 among the 'blue' population), and the maximum value of fitness is also higher (412.2, versus 339.4 in the 'blue' population – please excuse the histogram axis alignment).

The different spikes among the 'red' population show pockets of local semantic similarity, which suggest materially (e.g. not just among the poor units) higher semantic diversity as well. Below is a histogram of the distribution of pairwise Kendall's' τ coefficients for both populations.

Kendall's τ coefficient distribution for both policies



The above Kendall's τ coefficients have been normalized to fit a -1 to 1 scale, where 1 represents a perfect match (population unit pairs that agree complete for every pair of board states), -1 represents an inverse match (unit pairs that disagree completely), and a 0 represents no correlation/mutual information. The key thing to notice here is that the 'red' population (the one with the kill-parents policy) has a higher frequency of Kendall's τ coefficients between 0 and 0.5. These represent population units that are correlated (presumably because they in turn correlate with absolute rankings), but are not identical. On the other hand, the 'blue' population (which persists parents) has over 1/3 of its unit pairs that return the same result (these likely correspond to the same, suboptimal spike in fitness seen in the previous graph). This is one of many ways of demonstrating higher diversity in the 'red' population.

Quantifying the spread in Kendall's τ is tricky. The standard deviation of all τ within a population turns out to be a bad measure because it is heavily skewed by the spike at 1.0. However, the standard deviation of all $\tau < 1.0$ leaves out this outlier, as well as quantifies what is visually obvious – that the kill-parents policy results in higher diversity. In particular, the kill-parents policy results in a $\mathrm{stdev}(\tau|\tau<1)$ of 0.1552, while the persist-parents policy results in a $\mathrm{stdev}(\tau|\tau<1)$ of 0.1095.

VI.3. Discussion

The sample data above, in addition to the literature, supports and motivates our pursuit for higher fitness spread and diversity within our fitness landscape. In particular, Kendall's τ analysis motivated us to abandon experimentally persisting parents in favor of exclusively killing them off. These conclusions were also backed up by the percentage of our population that was successful against GreedyAI – our top 1/3 succeeded against GreedyAI 73% of the time at the time of Code Review 1, versus 82% of the time at semester's end (see Future Work for an explanation as to why our final bot lost to GreedyAI in the tournament).

VII. Future Work

There were two main shortcomings to our work, discussed below: first, our heuristics exploited weak symmetry rather than imposing strong symmetry, and secondly, our implicit incorporation of GreedyAI in the end game was not well gauged.

One of the implicit assumptions in designing the heuristic was that it was symmetric – e.g. flipping the evaluating player as well as toggling the color of all of the marbles and rotating the board would result in the same evaluation for both players. We used this assumption to justify our approach to minimax search, and it is in fact correct. However, this scheme does a poor job at making the opponent's position a factor in the evaluation of the board state – i.e. it encourages a player to be somewhat agnostic toward the opponent's position. An easy solution would be to include corresponding board metrics measuring statistics regarding the opponent's position into the genetic alphabet, which would make it easier for defensive strategic components to manifest themselves in the heuristics.

Secondly, the computer on which we evolved our players was not as fast as the game server, so players on our server often defaulted to GreedyAI even if they did not on the game server. We've realized that, despite penalizing players by how many moves in they default to GreedyAI, some of our most highly ranked players defaulted to GreedyAI. Examination of the game logs show that several of our heuristics (especially our most recent submissions, which happened to attach weight to radial distance from the middle) show poor end-game behavior – behavior, which, on our server, was masked because that player had defaulted to GreedyAI by the end-game. Possible ways to combat this would be to either attach a more severe penalty for defaulting to GreedyAI, or (possibly more effective) to cascade the GreedyAI valuation of a board state into the heuristic as a new metric, conditional on other metrics (which could indicate that we've reached the end game).

VIII. CONCLUSION

Over the course of this class, we approached the problem of Chinese Checkers by reducing it to the related problem of finding a good state heuristic through genetic programming. We generate heuristics by applying genetic operators to trees of functions on a set of predefined board state values. We experimented with several evolutionary strategies with respect to defining and estimating fitness (relative and benchmarked actual), the fecundity of the successful individuals versus the unsuccessful, varying mutation and mortality rates, and others. Our empirical results show that measures which preserve diversity rather than promote short-term fitness provide measurable long-term gains in overall peak fitness. The motivating example of choosing between a kill-parents or persist-parents policy shows the positive impact that promoting diversity has had on the efficacy of our evolution policy. We believe that despite the hiccups mentioned in section VII, our method (and especially the relative and absolute ranking-approximation approaches) is both interesting and fruitful.

IX. REFERENCES

Cited references:

1. Floreano, Dario, and Mattiussi, Claudio (2008). *Bio-inspired Artificial Intelligence: Theories, Methods, and Technologies.* Cambridge, MA: MIT.

2. Poli, Riccardo, W. B. Langdon, Nicholas F. McPhee, and John R. Koza (2008). *A Field Guide to Genetic Programming.*

3. Samuel, A. L. (1967). Some studies in machine learning using the game of checkers II—Recent progress. IBM Journal of Research and Development, 11(6), 601–617.

4. Schmidt, M. D., and Lipson, H. (2010) "Predicting Solution Rank to Improve Performance", Genetic and Evolutionary Computation Conference (GECCO'10), pp. 949-956.

5. Schmidt, M. D., Lipson, H., (2008) "Coevolution of Fitness Predictors," IEEE Transactions on Evolutionary Computation, Vol.12, No.6, pp.736-749.

6. Reisinger, J., Bahceci, E., Karpov, I., Miikkulainen, R., (2007) “Coevolving Strategies for General Game Playing,” Computational Intelligence and Games, 2007. CIG 2007.

Additional written references:

1. Russell, Stuart J., and Peter Norvig (2010). *Artificial Intelligence a Modern Approach*. Boston.

2. JR Koza (1992). Genetic evolution and co-evolution of computer programs. Artificial life II.

Code references:

1. Michelangeli, E. (2009) Fast O(n log(n)) Implementation of Kendall's Tau. Retrieved December 1, 2011 from http://projects.scipy.org/scipy/ticket/999.

X. ACKNOWLEDGEMENTS

We extend our gratitude to our TA Jason Yosinski for his support and guidance throughout the project, and for his input on several key issues – most notably, swaying us in the direction of using only numerical functions as a means of type consistency, and for the suggestion of incorporating a long-runtime penalty into our relative fitness calculation, which proved much simpler and more effective than our initial strategy of using a complicated computational resource handler.

We would also like to acknowledge Mr. Enzo Michelangeli, whose efficient implementation of Kendall's τ we adapted for use in our data analysis. A citation is provided in the References section above – see Michelangeli, E. (2009).

Appendix A

Some major contributions of each of the team members are given below.

Aperahama Parangi

Developed expression function tree along with function tree mutation operators (insertion, deletion, crossover). Conceived and deployed fitness estimation using benchmarking of heuristics based on comparing their rankings of a set of board states with those produced by a high quality heuristic used at a higher depth. Modified GameServer to run without memory leaks in development environment. Contributed significantly to literature review, overall system architecture, and project vision.

Peter Bailey

Implemented minimax search with best-first-node-expansion α-β pruning, which served as the template for all players produced. Performed several iterations of updates to improve performance and modularity of this template as well as other modules. Contributed significantly to literature review, overall system architecture, and project vision.

Anirvan Mukherjee

Designed and implemented the modified PageRank-based tournament model for relative fitness ranking. Migrated system to Python for flexibility and easier testing. Improved fitness benchmarking algorithm. Performed fitness landscape and diversity analysis to motivate project direction. Contributed significantly to literature review, overall system architecture, and project vision, as well as the written reports.

Appendix B

The major files in our evolutionary code are given below. Many of these files correspond to the sections laid out in System Architecture, as described below.

AlphaBetaAI.java

AlphaBetaAI is an AI player modeled on GreedyAI which can interact with the server. The design allows for an easily insertable heuristic and performs minimax search with best-first-sorted α-β pruning to some arbitrary fixed depth (we use 3 during gameplay) to choose the move with the greatest value. AlphaBetaAI serves as a template into which we can insert a heuristic to produce a player. In particular, inserting the greedy heuristic results in a greedy player that looks 3 moves ahead, thus consistently beating GreedyAI.

expression.java

The expression class represents a node in a function tree. Originally we had support for relatively complicated function trees with arbitrary numbers of children nodes and expensive unary functions such as powers, square roots, and logs but we decided after experimentation to stick with the cheap binary operators of addition, subtraction, multiplication, division. The expression class was designed such that it may be evaluated in real time or printed to a string of java code which can then be written to a file inside of AlphaBetaAI, compiled, and then run as a player on the server.

gController.java

gController maintains a population of players and performs the necessary update operations on them such as evaluating their fitness, killing off the weak players according to some scheme, and producing a new generation of players according to some scheme. Mortality rates, mutation rates, populations sizes and the like are all controlled from here.

gHeuristic.java

gHeuristic contains the methods which manipulate expression objects. The various types of mutation such as insertion, deletion, and crossover are defined on the expression tree structure here. Continuous rather than single point crossover was used.

gPlayer.java

gPlayer is a used by gController to keep track of specific instances of an expression and match it with a calculated fitness score for the purposes of maintaining the population. gPlayer also contains methods to transform an expression function tree into a runnable java class file.

AI_285.java

Our final submission. As with all population units, AlphaBetaAI serves as the template for this file, with the heuristic inserted. See section VII for an important note regarding the performance of this submission.

kendallstau.py

A derivative of Mr. Enzo Michelangeli's implementation of Kendall's τ, used for quantifying similarity between heuristics, as well as diversity.

analysis.py

Parses output files of our test setup to obtain rankings of board states for population elements as well as their finesses. Used to obtain fitness variance, the fitness distribution, and the Kendall's τ distribution. The resulting data is presented in histogram format in section VI.