Optimizing Stratego Heuristics With Genetic Algorithms

Ryan Albarelli

December 5, 2003

Abstract

This paper describes research into the application of genetic algorithm towards evoloving a strategy for competing in Stratego. Normally, heuristic development is a "guess and check" process whereby the programmer must gauge the relative weights of certain aspects of any given board state. The GA implementation will attempt to evolve an optimal heuristic for the Stratego minimax algorithm using the genetic operations of recombination and mutation in tandem with the natural selection process. In these experiments, it is shown that the GA is effective at improving heuristics given good enough opponents in the fitness function.

Keywords

Evolutionary Algorithms, Stratego, Board Games, Board Game AI, Genetic Algorithms

Contents

1 Introduction

1.1 Motivation

When heuristics for Stratego are written by hand, they are enhanced by human creativity and yet limited by the lack of raw computational power of the human mind. Unlike people, Genetic algorithms (GAs) don't have a sense of intuition: they treat all solutions as possibilities regardless of "obvious" flaws. This allows for non-intuitive paths that may lead to optimal Stratego heuristics. The human resources available to create viable Stratego heuristic are limited and expensive; a GA to optimize a heuristic could run for an indefinite period of time attempting to improve the parameters. Furthermore, this research could be expanded towards developing a genetic algorithm for the general case adversarial board game heuristic. This could drastically cut down on the time required to develop decent AI players for new games as well as obscure games that don't have a large following.

1.2 Background

Genetic algorithms are a versatile set of methods employed by many systems today. GAs generally use an objective function along with the genetic operations of crossover and mutation on a set of solutions to attempt a directed search of an the exponential search space of a hard problem. Unlike hill climbing methods, GAs tend to avoid getting stuck at local maxima when searching the solution space. GAs were introduced in the 1970s but have only recently gained widespread use as their computationially intensive nature has been overcome by recent advances in computing power.

In order to apply a GA to a problem, the problem must be represented as a chromosome, generally a binary string. An objective function must be defined

that evaluates a given candidate solution and determines its value as a solution to the problem. A fitness function merely maps the objective function such that the best objective values give higher absolute values in a global sense.

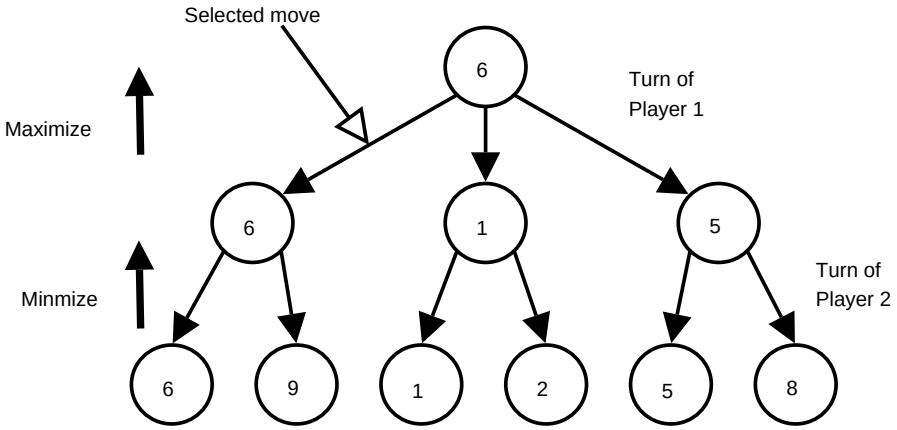
The genetic operators must also be defined for a GA to function. Generally the crossover operation swaps subsections of the binary representation of two candidate solutions giving two new solutions that components of their predecessors. The mutation operator randomly modifies solutions (at a set rate) in order to introduce genetic diversity into a converging population of solutions.

1.3 Minimax Algorithm

The minimax algorithm is an adversarial state space search algorithm that is used by many game AIs to exhaustively search a certain number of moves advanced from a starting state [3]. The minimax algorithm is deterministic in that given the same initial state and search depth limit, the same best move will be chosen each time. The minimax algorithm is often represented as a tree of board states with the initial state at the root. Each branch of the tree represents a possible move from the given board state. All possible moves from the initial board state are enumerated as turns for the current player "max". Minimax then takes each of the resulting states of these moves by the "max" player and enumerates all the possible moves the opposing player "min" can make. This rotation continues up to a specified depth at which point the state is assigned an value by a state evaluation heuristic. The heuristic gives greater values for board states beneficial to the max player. With this in mind, the algorithm takes advantage of the fact that the max player will want to make the moves that result in the highest heuristic value while the min player will want the minimum heuristic value for the max player's turn. To search the entire space of most games would take an inordinate amount of time so a depth limit is implemented. The algorithm proceeds downward until it reaches this depth limit at which point it considers the current node a leaf node. In this manner, the minimax algorithm is considered an approximation since it cannot moves past the horizon of its depth limit. The algorithm proceeds from the leaves to each ascending level. At each level where min was the moving player, the child node with the minimal heuristic value is chosen and its value is assigned to the current node. For each max player, the maximum heuristic value of its children is assigned. This pattern continues up the tree until the max player finds the subtree with the highest heuristic value and selects that branch as the correct move to make.

The Minimax algorithm has an exponential space and time complexity as described. A few optimizations are described that improve the space complexity to d, the tree depth, and improve the time complexity from $O(b^d)$ to $O(b^{\lceil 3d/4 \rceil})$ where b is the branching factor. By searching the tree in a depth first manner, only one copy of the board is stored (and modified at each link traversal) along with the best heuristic value found so far at each depth. A technique called alpha beta pruning attempts to further optmize the minimax algorithm by pruning away branches that cannot lead to an optimal solution. As the al-

Figure 1: Minimax Tree



gorithm proceeds, it keeps a window of the best and worst move found thus far. When a move is found that falls outside the window, the resulting subtree will never yield the optimal value and the current branch is aborted. The alpha beta pruning generally reduces the branching by a factor of two.

Many minimax algorithms also include a mechanism to prevent repeated moves. A history of previous moves is kept and each move is checked against that list to prevent cycles of the same move.

In Figure 1, an example minimax tree is given. The arrows represent move candidates and each leaf contains the value generated by the state heuristic.

1.4 Research

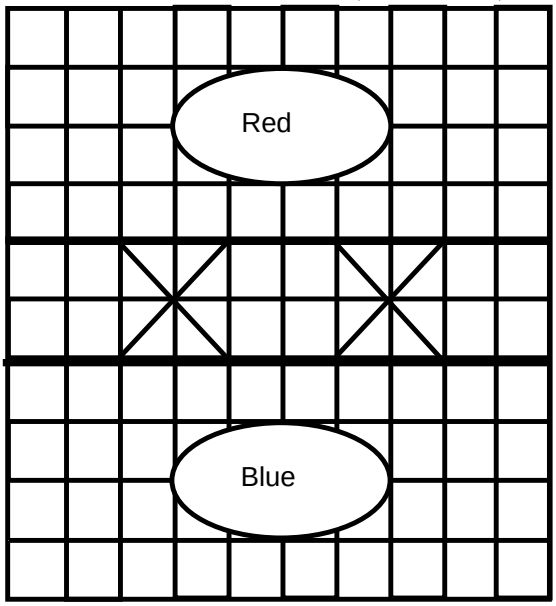
The research questions this paper addresses are as follows:

- Can genetic programming be successfully applied to Stratego?
- If successful, is the result applicable to other board games?

1.5 Stratego

Stratego is a two player turn based strategy game. The board has a 10 by 10 grid layout with each player using the 4 rows nearest their respective sides as starting blocks. Each player begins the game with 40 pieces of varying rank and mobility and may initially place those pieces in the starting block. Each player receives 6 immobile bomb pieces and 1 immobile flag piece. The mobile pieces are ranked from 0 to 9 with pieces ranked pieces having the ability to attack and destroy pieces of lower rank. Mobile pieces that attack enemy bomb pieces are destroyed with the exception of the miner (rank2). When a piece

of equal rank is attacked, both pieces are lost. The objective of the game is to capture the enemy flag (attacking it) or destroy all of his mobile units.



2 Methodology

2.1 Base Heuristic

Most minimax heuristics are linear weighted sums of various aspects of a given game state. In this sense, the heuristic shell is a static weighted summation of each individual aspect of a given board state. The base heuristic is given by the formula $\sum_{i=0}^{n}(w_i * a_i)$ Each a_i is a static representation a different aspect of the board state. Examples of aspects include number of enemy pieces, number of friendly generals etc. Since the Although the value of the a_i will change from board state to board state, the idea it is representing will remain static. The set of w_i represent the weights of each board aspect. Even though the aspects vary in range and meaning, the optimization of the weights has a simultaneous effect of scaling the aspects relative to each other. The goal of the genetic algorithm is to optimize the set of weights, w_i. The base heuristic also includes a "tip" factor that tips the heuristic value up to 3% to introduce variation in gameplay for a given heuristic.

2.2 Solution Representation

A candidate solution is the set of w_i that are applied to the Stratego base heuristic to aspects. Each w_i is stored as a signed 8 bit integer allowing for weights in the range $[-128, 127]$. Each candidate solution is represented as a

binary string, b, defined as the concatenation of the binary representation of the w_i for the particular solution. A binary representation was chosen over an integer representation to allow an existing mutation operator to fluctuate the weights in a variable manner. The GA class used is generic in the sense that it will work with any length binary string. An individual must be converted from the binary string format to a series of 8bit integer weights before being applied to the base heuristic function.

2.3 Genetic Process

The GA class performs several initialization tasks on the population. The GA class performs a one time creation of the entire population during its initialization phase to avoid costly operating system allocation calls later on. The trial solutions are stored in an array of length equal to the maximum possible population size. There are two indexes into this array, the current population capacity, and the competition position demarcating surviving parent solutions from solutions not surviving the competition phase. The position of the demarcating index is a variable parameter in the GA and affects what percentage of the solutions is kept from generation to generation. This parameter could be varied from one extreme to the other to obtain a generational or steady-state EA algorithm.

The executable takes one command line parameter, the name of the file that contains all of the operational parameters for the GA. This parameter file contains one parameter per line in the format "parametername, parametervalue". Here is a list of the parameter names and their functionality.

2.4 Parameters and Usage

See Table 1

3 Evolutionary Process

3.1 Generations

After the initialization, the user may call the TimeStep function to cause one generational step in the GA algorithm. Each timestep consists of 5 stages: Evaluation, Adjustment, Competition, Selection, and Mutation. During Evaluation, the GA computes the value of each trial solution soley as the profit of that solution. The Adjustment phase consists of potentially moving the populcation cap closer to the maximum population to increase diversity. To simulate competition, the array of candidates is quick-sorted leaving the best candidates on the surviving side of the competition line as described in section 2.3. During the Selection phase, a ranked selection process moves down the list of candidates and chooses two parents to combine into into a new child. This process is repeated for each child space on the poor side of the competition line. Each child is produced by a random selection of genetic material from each parent based

Parameter	*Description*	*Range*
initial_population	Initial population size	10-population_cap
population_cap	Maximum size the population can ever reach	10-100000
quietlimit	The number of intervals between pop increase	1-1000
popincr	Quiet period increase factor	1.1-1000
percent_new	The % of the pop replaced each generation	.8-.1
select_chance	The % chance for selection	1.0-.001
num_pt_cross	The N-point crossover parameter	1-stringlength
mutrate	Mutation rate (Pm)	.000001-1.0
generations	Stop the optimization after X generations	0-1000000
timelimit	Stop the optimization after this many seconds	0-1000
statcount	Analysis on this many iterations	0-100
loginterval	Log every interval generations	0-1000
logfile	Log to this file	-

Table 1: GA Parameter List

on the N-point crossover scheme (where N is a parameter to the system). The mutation phase consists of appyling the Pm (chance of mutation) to each bit in each individual. If the chance occurs, the bit is flipped. In this way, variation will still occur to progress the algorithm along in the case of convergence towards a non-optimal solution.

3.2 Evaluation

The fitness/objective function is one of the key pieces of a working genetic algorithm. In order to evolve towards better Stratego heuristic weights, there must be some judge of how well a particular set of weights performs. Here are three basic metrics for which a set of weights is evaluated.

- victory
- pieces remaining
- moves required

Victory is the obvious choice as the most important metric of an heuristic's performance. Victorious heuristics are given a large fitness bonus over those which lose. The number of pieces remaining for a particular heuristic is a good measure of how well it was doing at the end of the game. A heuristic that lost its flag but managed to capture many key enemy pieces might have merely been unlucky. It is also desired that the game win in a shortest number of moves. For any particular matchup, the winning heuristic's fitness is decreased by each move it requires to win the game. The losing heuristic is given a slight fitness boost for every move it stays alive. In this manner, the more efficient weights will stay in the gene pool given equivalent numbers of victories. The fitness

function for a winning and losing heuristic is given by the following formulas respectively.

- $400 + (-1) * moves + 10 * (pieces - pieces_o)$
- $moves/2 + 10 * (pieces - pieces_o)$

The fitness evaluation process is hampered by the persistent randomness inherent to any gaming environment. This randomness is affected by three factors:

- tip factor
- opponent skill
- starting positions

A heuristic may perform well against one particular opponent but fail against a more commonly skilled opponent. While the tip factor of the base heuristic prevents an opponent from guessing the move heuristic, it also prevents the same heuristic from performing the same each time. Starting positions can also drastically affect gameplay. If the flag is placed too close to the front lines it may be captured unintentionally by the opponent. If critical pieces are placed too far back, they may become trapped when needed leading to an increased number of moves required or loss of the game. To combat these problems, a second pool of opponent candidates is kept. This weights of the individuals in this pool are initialized to random values at the beginning of system execution. After each generation selection, a random member of the opponent pool is overwritten by a random candidate from the top 10% of the general population pool. For a smoother fitness value, each member of the general population pool is required to play all members of the opponent pool once per evaluation. The final fitness value for an individual is the sum of the fitness against each opponent pool member.

4 Experiments

4.1 Restrictions

In order to execute the genetic algorithm in a reasonable amount of time, certain restrictions were placed on the gameplay. The maximum number of turns per game was limited to 1000. Although the minimax algorithm can easily reach depths of 10 for Stratego, the depth of the minimax algorithm was limited to 4 to reduce the amount of time required to run the massive number of games required by the GA.

Parameter	*Value*
initial_population	10
population_cap	100
quietlimit	5
popincr	1.4
percent_new	.8
select_chance	.7
num_pt_cross	24
mutrate	.04

Table 2: GA Parameter Settings

4.2 Setup

The genetic algorithm was configured to the parameters shown in Table 2. The maximum generations was ranged from 30 to 1000 in an effort to determine how often the GA would break out of a local minima. In one experiment, the generational restriction was released and a maximum time was set at 8 hour in order to attain the highest fitness individual possible. The crossover parameter was set equal to the number of integer values being represented by the binary GA. This forces crossover to always occur at integer boundaries, preventing integers from being crossed together to give odd results.

4.3 Parameter Variances

The mutation parameter was varied from $1/(2m)$ to $3/m$ to find the effectiveness of mutation in the experiement. The population size was increased from 10 up to 100 to see if larger populations produced a faster convergence rate than smaller ones. The size of the opponent pool was varied from 3 to 10 to determine the effect of more competition among the heuristics for fitness.

4.4 Statistical Analysis

Each test was executed at least 10 times in order to gain smoother average results. Since the average and overall fitness are both relative to the skill of heuristics in the opponent pool, any type of statistical analysis between subsequent executions would not be comparable. The consistency between subsequent runs is secondary to the real purpose of the experiment: to develop an intelligent heuristic. With this in mind, it is expected that the lack of stasticial analysis will be tolerable.

Figure 2: Maximum fitness over time

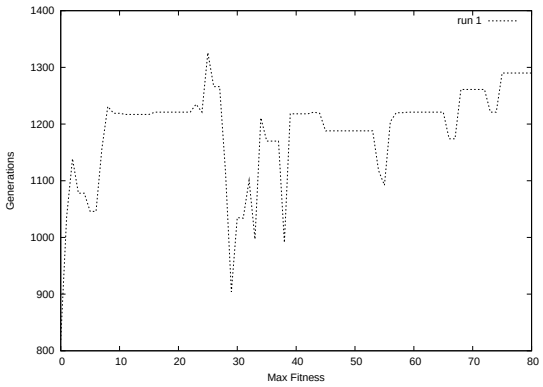


Figure 3: Average fitness over time

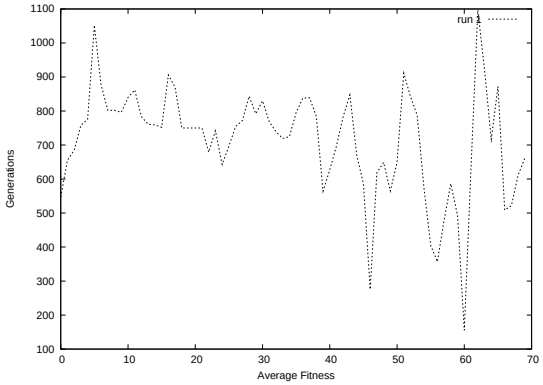
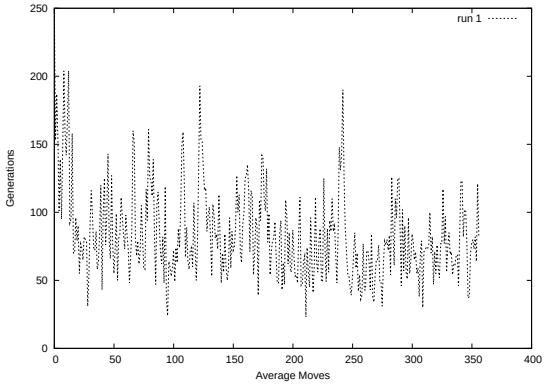


Figure 4: Average number of moves per game over time



5 Results

5.1 Variance of Parameters

As the mutation rate was lowered, the individuals in the population became less diverse and the opponent pool quickly became homogenous. Since the fitness is tied to skill of the heuristics against the opponent pool, the fitness level decreased and stagnated as the heuristics played their "cousins" that resided the opponent pool. The rate of convergence grew inversely proportional to the mutation rate.

In a general sense, the average fitness of the population quickly rises as the GA begins to develop its away from the random initial heuristics as seen in Figure 4. After this initial rise, the fitness tends to remain remain on a line with varying fluctuations. The fitness doesn't tend to rise any further because as the heuristics improves, the opponent pool also improves. This is also what leads to the fluctuations in average fitness value. When a good heuristic is placed in the opponent pool, the fitness of the lesser heuristics will go down as they are forced to compete against better opponents.

In Figure fig:plot4, the maximum fitness is seen to generally increase, but also experience dips at intervals. These dips are explained by the stochastic nature of gaming. Sometimes a heuristic is lucky and does well, and other times it is unlucky and loses quickly. When the best solution so far happens to lose a game, its fitness will drop below the second best heuristic for a generation. Just as in real games, the better opponents manage to win the most games, even when luck is involved.

The average number of moves, as seen in Figure 4, fluctuates wildly through most of the lifespan of the GA. The important aspect to notice is that in the very early generations, the number of moves consistently takes a lengthy amount of time. After approximately 20 generations, the average number of moves being used by the heuristics to win games becomes approximately half of the early values (on average). Being that the number of moves required is in the fitness evaluation, this result seems to support the notion that the heuristics are getting better with each passing generation.

6 Conclusion

Based on the results shown above, the GA appears to produce better heuristics with time. In an absolute sense, it is difficult to conclude whether the heuristics produced by the GA are effective against human players. The mere fact that the GA improves the heuristics suggests that even if the resulting heuristic is not real-world viable, the GA could be effective at generating better opponents given the right input opponents to the fitness function (i.e. fitness by playing against human players). As it stands now, the GA can only get better against what it generates itself, which might not be a viable solution anyways.

7 Future Work

This research has the potential to open the door to further study into developing genetic algorithm techniques for other board games that use the minimax algorithms. Starting board layouts could be optimized simultaenously or independently of the board heuristic using a similar genetic algorithm. An integer representation EA could be developed to replace the binary representation GA currently being used. Since the integer representation matches the actual values needed, better or faster results may result. In addition, an EA might be applied to optimize the weights in the fitness/evaluation function. Instead of the two pool opponent approach taken in this implementation, a match play tournament style of evaluation could be setup whereby each heuristic plays many of the other heuristics and gains a point score.

References

[1] Jean-Marc Alliot and Nicolas Durand. A genetic algorithm to improve an othello program. In *Artificial Evolution*, pages 307–319, 1995.

[2] Jeffrey W. Herrmann. A genetic algorithm for minimax optimization problems. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1099–1103, Mayflower Hotel, Washington D.C., USA, 6-9 1999. IEEE Press.

[3] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. A minimax algorithm better than alpha-beta? no and yes. Technical Report 95-15, none, Edmonton, Canada, 1995.