Feasibility of Applying a Genetic Algorithm to Playing Stratego

Vincent Tunru

Supervisor: prof. dr. Vincent van Oostrom
7.5 ECTS
February 23, 2012

Contents

Introduction

The field of AI has concerned itself with making computers play games for almost as long as it exists (Schaeffer and Van den Herik, 2002). Since the initial artificial checkers player, many games have seen their top-class human players fail to beat an artificial opponent. Initially, many approaches to making a computer play games focused on "brute forcing" a solution: by simply computing long enough, one could find out all the possible ways a game could end, and pick the move that is part of a path that ends in a win.

This approach, however, proved to be insufficient for many complicated games. More sophisticated search and learning algorithms were then developed, leading to more and more breakthroughs. Some techniques achieved great success for two-player perfect-information games. Games with imperfect-information add an additional difficulty, though: one cannot see the opponent's cards, or the course of the game is dependent on the outcome of dice rolls, or some other piece of information is hidden. This means that the size of the search space increases vastly, needing to take into account many different scenario's of possible courses of the game. As a result, search techniques that are successful in perfect-information games cannot simply be applied to imperfect-information games. Nevertheless, even in the area of games with incomplete information a lot of progress was made. For example, it is believed that the strongest Scrabble player is now non-human (Sheppard, 2002).

One such game with imperfect information is Stratego. There has been some research on crafting an artificial Stratego player of a reasonable skill level, however, all research I could find hinges on a skilled player inserting his/her knowledge of the game into a computer. In this thesis, I will try to find out the following:

Is it feasible to apply a genetic algorithm to playing Stratego?

This research is conducted in tandem with that of Roseline de Boer, who investigated how results generated by our artificial Stratego player can be used by human Stratego players and vice versa: how knowledge from an expert human Stratego player (Roseline was Stratego world champion from 2008 to 2010) can be incorporated in the design of an artificial player.

The thesis is structured as follows. First, I will investigate the game of Stratego, and what efforts have been undertaken before to approach the problem of making a computer play Stratego at a reasonable level. I will then outline the approach we took in designing our genetic algorithm Vicki and the design decisions we took. Finally, I will evaluate the results and analyse the feasibility of this approach to an artificial Stratego player, pinpoint weaknesses in our approach and make suggestions on what kind of research would be logical next steps.

1 Background

1.1 Stratego

Stratego is a game for two players played on a 10x10 board. Each player has 40 pieces with ranks varying from Marshal (0, strongest) to Spy (9, weakest).

Table 1: Stratego board

R	R	R	R	R	R	R	R	R	R
R	R	R	R	R	R	R	R	R	R
R	R	R	R	R	R	R	R	R	R
R	R	R	R	R	R	R	R	R	R
		∼	∼			∼	∼		
		∼	∼			∼	∼		
B	B	B	B	B	B	B	B	B	B
B	B	B	B	B	B	B	B	B	B
B	B	B	B	B	B	B	B	B	B
B	B	B	B	B	B	B	B	B	B

At the start of the game, each player places those pieces on the four rows at his/her own side, such that the opponent is unable to see the ranks of the pieces. Table 1 displays a Stratego board and the positions that the pieces of the red player (marked with R) and the blue player (marked with B) should be placed on.

Who gets to make the first move (and play the red pieces) is decided by chance. Once all pieces have been placed on the board, each turn a piece can be moved either forward, backward, left or right, moving a single square at a time. Exception to this last rule is the Scout (8), which can move as many squares as possible provided all those squares are in the same direction and unblocked. Pieces cannot be moved onto squares inhabited by other pieces of the same player, or onto any of the eight squares of water in the middle two rows of the board (marked with ∼ in table 1).

When a piece is moved to square that is already occupied by one of the opponent's pieces this is called an attack. When the player's piece is stronger than the opponent's piece (i.e. has a lower rank), the opponent's piece is removed from the board and the player gets to finish the move as he/she would have if there had been no piece. If the opponent's piece is stronger than the player's piece, the latter is removed from the board and the opponent's piece retains its position. When both pieces are of equal strength, both pieces are removed from the board.

The goal of the game is to find and attack the opponent's Flag. Every piece that attacks the Flag beats it, and once it is caught, the game is over and the player that captured the Flag wins. Alternatively, one can win when

the opponent is unable to move, The Flag itself cannot move — it has to be protected by other pieces.

Another piece that cannot move is the Bomb. In exchange for this immovability, every piece except the Miner (7) is beaten when it attacks a bomb.

The last exceptional piece is the Spy (9), which is beaten by every other piece but wins when it initiates an attack on the opponent's Marshal (0). This makes it and the the bomb the only pieces that can beat the marshal.

It is undesirable to continuously move the same piece back and forth on the same two squares; hence, the *two squares-rule* was instated: it is forbidden to move back and forth on the same two squares for more than three consecutive turns[1].

1.2 Computer Stratego

In 2007, 2008, 2009 and 2010, the USA Stratego Federation organized Computer Stratego World Championships that have spurred research into various approaches to tackling the problem of making a computer play the game of Stratego. Most of these approaches involve hardcoding specific strategies. For example, Vincent de Boer's bot *Invincible* (De Boer, 2007) revolves around a number of "Plans" that assign values to possible moves. Some Plans are more important than others, where more important plans have greater influence in which move is finally decided upon.

This approach resulted in an automated player that was anecdotally better than average players, however, constructing it required a skilled Stratego player (De Boer was world champion in 2003, 2004 and 2007). Other attempts to produce a skilled artificial Stratego player include a multi-agent approach where each agent followed a set of "rules" that were activated when certain predefined conditions were met (Treijtel and Rothkrantz, 2001), and a strategy revolving around predefining a number of valuations of different pieces (Arts, 2010). A more comprehensive overview of scientific approaches to computer Stratego is also available in the latter article.

1.3 Machine learning

Since skilled Stratego players are a scarce resource, it would be more useful if a computer could find out what strategies work well by himself. In fact, this could even be instructive in turn to the skilled Stratego players: the computer might find out strategies that work well but are not widely known. Likewise, it might show strategies thought to work well to be relatively ineffective.

For example, in Sheppard (2002), it is explained that whereas experts in the game of Scrabble believed playing extra tiles was beneficial because it allows a player to draw new, potentially useful, tiles, the genetic algorithm did not in fact evolve such behaviour, because it stimulates preserving bad tiles. Thus, even though human Scrabble players are significantly outclassed by computers, they can improve their play by drawing lessons from the computer's behaviour.

[1] As per the Stratego Original rules by Jumbo: http://www.jumbo.eu

1.4 Parameter-based approach to genetic algorithms

There are various ways to implement genetic algorithms, which for a large part distinguish themselves in the way the problem is represented. For example, as described in Koza (1990), complete programs can be generated using operators to manipulate if-then clauses.

In our research, we opted to adopt the relatively simple approach of evolving a set of parameters. Although this method is still highly dependant on knowledge of the actual game and strategies that work well, as noted by De Jong (1988) "significant behavioral changes can be achieved within this simple framework".

Solutions generated using this method sport a set of parameters, along which all possible moves at a certain turn are evaluated. When a move satisfies the criteria of such a parameter, the value of that parameter influences the final valuation of that move by a certain amount. The amount of influence each parameter has (the parameter values) is decided by our genetic algorithm.

Initially, we generate a population of potential solutions whose parameter values are initialized to a random value. We then calculate the fitness of all potential solutions with a fitness function (see section 2.1). Using this fitness we can generate a new population. For each solution in the new population, we select two "parents": solutions from the previous population, where solutions with higher fitness are more likely to be selected as parent (see section 2.2). Each parameter for the new solution is copied randomly from one of the parents. This way, parameters that play a part in making a solution work well (have a higher fitness value) are more likely to be preserved in a new generation, leading to the solutions converging to sets of parameters that work well.

2 The Algorithm

A high-level overview of the algorithm used to generate Vicki is as follows:

for $generation = 1 \rightarrow 20$ **do**
  if $generation == 1$ **then**
    Initialize 200 solutions with random parameter values $-0.5 < x < 0.5$
  else
    Use the solutions generated at the end of a previous generation
  end if

  for each solution **do**
    Play five matches against a randomly playing opponent
    Calculate the average fitness over those matches ▷ See section 2.1
  end for

  for $newSolution = 1 \rightarrow 200$ **do**
    Pick two solutions to use as parents ▷ See section 2.2
    for each parameter of the new solution **do**
      Pick this parameter at random from one of the parents
    end for
  end for
end for

Section 4.1 outlines the reasoning behind the specific values used.

2.1 Fitness function

Our approach hinges on being able to determine how well a solution would perform. We chose to base this valuation on performance of a solution when playing a number of matches against a player making random moves, due to its being a stable opponent in that it will not suddenly get a lot better, and can be used as early as the first generation. To calculate the performance we give the solution a number of points depending on whether he wins the match, the amount of pieces he has left at the end of the game, the amount of pieces the opponent has left and the amount of moves needed to end the game.

The amount of points given defines the importance of each of these factors. The most important factor is whether the solution actually wins. Consequently, it provides the highest reward. Next, each owned piece left gives additional points. This stimulates convincing wins (where the opponent's flag is quickly found without sacrificing many of one's own pieces) and makes sure losses that were the result of bad luck, such as the opponent coincidentally locating one's flag at the start of the game, aren't punished as harshly as convincing losses. Likewise, each piece the opponent has left will result in points being subtracted, making convincing wins rank higher than accidental wins. However, we chose to limit the effects of this factor because it might stimulate inefficient search strategies that result in VICKI trying to beat each piece that is not the flag before turning to winning the game. Finally, a couple of points are deducted for each move made to punish solutions that repeatedly make the same move and use stalling tactics — mostly to speed up the process of evolution and prevent looping from happening.

Because an opponent making random moves could lose a game simply by coincidentally picking the wrong moves, we pit each potential solution against the randomly playing opponent five times, and take the average of the fitness values of each match to determine the final fitness of the solution.

The fitness function we settled on was as follows:

$$f(s) = (\sum_{i=0}^{5} -0.25 * plys(s,i) + 100 * p_s(s,i) - 20 * p_o(s,i) + 1000 * w(s,i))/5 \quad (1)$$

Here, $plys(s,i)$ is the number of plys solution s played in match i (i.e. the amount of moves made by the opponent and the player combined), $p_s(s,i)$ is the amount of pieces solution s had left at the end of game i, $p_o(s,i)$ are the amount of pieces the opponent of game i had left at the end of the game, and $w(s,i)$ is 1 when s won game i and 0 on loss.

2.2 Breeding new generations

Having determined the fitness of each solution in a generation, a new generation needs to be bred. The way this is done is another important factor in determining the quality of a genetic algorithm. It is especially important to balance the influence well-performing solutions have with the desire to keep the population diverse in the beginning. If we converge to a set of parameters that

work well prematurely we might miss out on completely different sets that might have worked even better (Beasley et al., 1993b). Additionally, solutions that happen to perform well accidentally (e.g. by being pitted against opponents whose random moves are particularly ill-chosen) might end up influencing the final solutions far more than desired. On the other hand, stimulating diversity too much will not result in strong solutions emerging, as they might not have enough chance to breed offspring.

We took all this into account when coming up with a way of breeding a new population that is best described by a simplified example (illustrated in figure 1). Take an initial population of three solutions with fitness values 2, 2 and 3, respectively. We now want to generate a new population of three. For each solution in the new population, we perform the following steps to select each parent:

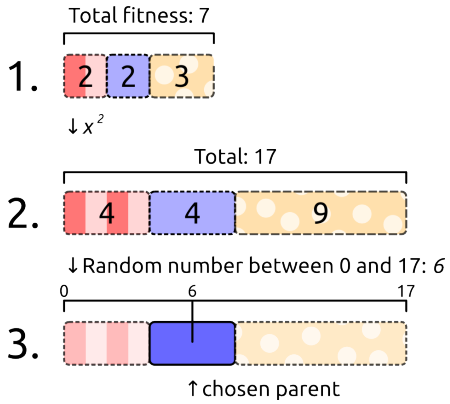

Figure 1: Parent selection

1. Preserving the (arbitrary) order in the solutions, we multiply each fitness value with itself, to exponentially increase the percentage of the cumulative fitness being provided by better solutions (see also section 4.1.5). The fitness values are now 2^2, 2^2 and 3^2, or 4, 4 and 9, adding up to 17.

2. We now take a random value between 0 and the sum of all solutions, i.e. between 0 and 17, in this case. For this example, assume the random value to be six.

3. The solution that is at this point in the scale is selected as parent. In this case, the value of the first solution is 4, which is less than 6. Moving on to the second solution though, we see another 4 which, added to the previous 4, results in a cumulative value of 8, which is higher than 6. Hence, the second solution is the one to be chosen.

4. Repeat these steps for choosing the other parent.

For each of the parameters required for the new solution, we randomly copy the value of either of its parents. This means we do not combine parts of parameters of the parents, and parameters that are "next to each other" do not have a higher chance of being propagated to the new generation than parameters that are "far away".

3 Parameters

Parameters to use were determined with Roseline de Boer, Stratego world champion from 2008 to 2010. For a detailed analysis of the effects of the parameters, see de Boer (2012).

FIRST_MOVE When a piece has not moved yet, this parameter will be added to the evaluation. As long as a piece has not moved, as far as the opponent is concerned, it could still be a bomb. If a piece wants to trick the opponent into thinking it is a bomb, this parameter would be a negative value so that moves that result in moving a piece for the first time will be less likely to be chosen.

STILL_OPPONENT When a move would result in attacking an opponent, and that opponent has not moved yet, this parameter will be added to the evaluation. If a piece has not moved yet, it is more likely to be a bomb (because bombs cannot move), so pieces that would prefer not to attack bombs are expected to develop low values for this parameter.

BEAT_OPPONENT When a move would result in attacking an opponent whose rank is known, and whose rank is lower than that of the piece that is to be moved, this parameter will take effect. It is different from the other parameters in the sense that this value will be multiplied by a higher value when the opponent is a stronger piece.

BIAS_<direction> For each possible direction (left, right, towards the opponent and towards one's own side), a parameter is included that specifies a bias in a certain direction, i.e. a positive value for a certain direction indicates a preference to move towards that direction, a negative value indicates a preference to stay away from that direction.

EXPLORATION_RATE When a move would result in attacking a piece of which one does not know the value yet, but that has already moved (i.e. is not a bomb and not the flag), this parameter will be added to the evaluation. This means that a brave piece will have chosen a high value for this parameter, whereas more conservative pieces would rather leave the attacking to the braver pieces and assign this parameter a low or even negative value.

WATER_CORNER The squares at the opponent's side that are diagonally next to the water are supposedly tactical places for offensive pieces.

VALLEY_EDGE The outermost squares of the two rows in the middle of the board are supposedly good positions for defensive pieces.

RANDOM_INFLUENCE The higher the value of this parameter, the more likely a piece is to make a random move. In other words, if this value is higher, a random valuation that is added to the valuation is more likely to be higher/lower, thus influencing a move's valuation more. This makes a strategy less predictable.

4 Results

Figure 2 shows a plot of the data in table 2: the average fitness of each generation of a run. For an elaboration on the process of obtaining these results, see section 4.1.

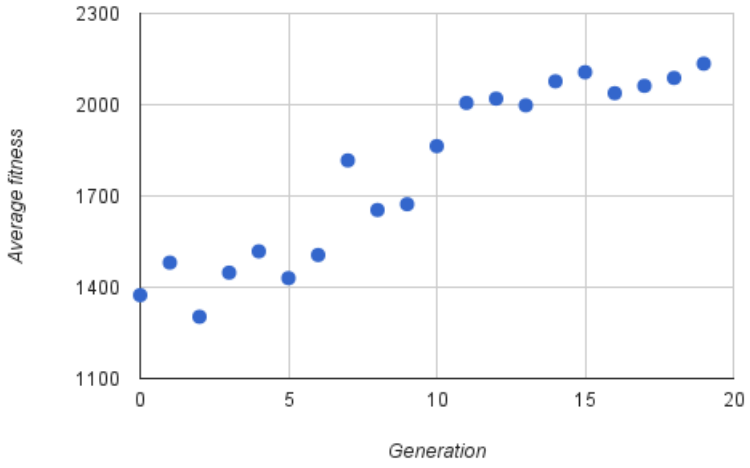

Figure 2: Average fitness by generation

A few things are worth noting:

- There is a clear upward trend. Apparently the parameters do manage to influence behaviour enough to better achieve the goals embedded in the fitness function.
- At around generation 12, the rate of improvement appears to flatten out. This might seem fairly quick but, as noted by De Jong (1988), it is common for GAs that "typically, even for large search spaces (e.g. 10^{30} points), acceptable combinations are found after only ten simulated generations". Stratego has an upper bound of 10^{115} possible states, and of a game-tree complexity of 10^{535} (Arts, 2010) (compare to chess's upper bound of 10^{50} states and a game-tree complexity of 10^{123} (Allis, 1994)).
- Most of the solutions already defeat a randomly playing opponent in the first generation; the fitness improvement is mostly the result of performing well at the other characteristics defined in our fitness function (preserving one's own pieces, defeating as many opponent pieces as possible and ending the game in as few moves as possible).
- Our implementation was incomplete: the genetic algorithm had no way to evolve the ability to move the Scout more than one square at a time.

Table 2: Average fitness by generation

Generation	**Average fitness**
1	1374
2	1481
3	1303
4	1448
5	1518
6	1430
7	1506
8	1817
9	1654
10	1673
11	1864
12	2006
13	2020
14	1998
15	2077
16	2107
17	2038
18	2062
19	2088
20	2135

Table 3: Algorithm options

Factor	**Value**
Population size	200
Amount of generations	20
Number of parents to breed offspring	2
Number of opponents for calculating fitness	5
Likelihood of fit solutions to reproduce	$x^2/\sum$

4.1 Methodology

When developing a genetic algorithm, there are several choices to be made that might or might not affect performance to a certain extent. An overview of the choices made can be seen in table 3; for a description of each, see below.

4.1.1 Population size

De Jong (1988) suggests that often "GAs can rapidly locate structures with high fitness ratings using a database of 50-100 structures". However, in our tests, a population of 100 solutions appeared to converge rather rapidly to an average fitness sigiicantly lower than obtained with a larger population (see figure 3).

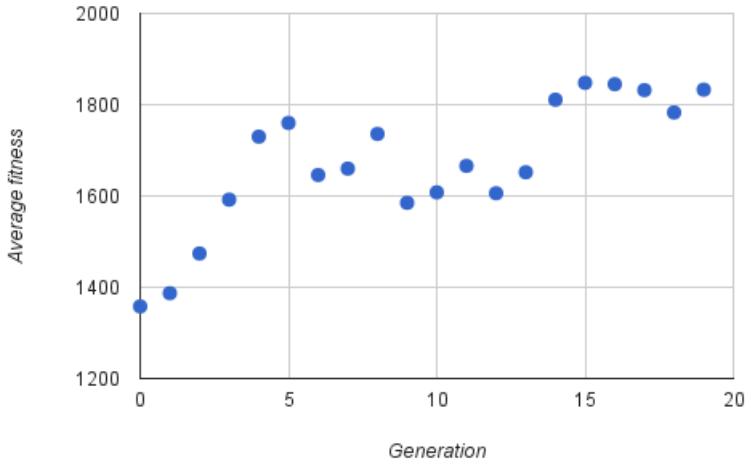

Figure 3: Average fitness by generation (population 100)

A larger population size means increased diversity and thus allows for more specific strategies to be evolved. Due to our algorithm not applying mutation (see section 6.1), the variety of our population is limited when compared to other GA implementations that do, which might be the reason a larger population size

is needed for ours. However, increasing the population size comes with a cost: the time is takes to run the algorithm grows proportionally. Increasing the population to 200 did provide enough diversity to produce the results shown in figure 2, while achieving acceptable performance on our test machine.

4.1.2 Amount of generations

The results shown in figure 2 show the average fitness improving until about population 12, so ending the simulation after 15 generations could well be sufficient in most cases to ensure the maximum fitness growth has been achieved while decreasing the amount of time required for a simulation to run. This is a rather simple approach; a more sophisticated approach could have been chosen, such as terminating evolution when the algorithm is likely to have converged maximally. Greenhalgh and Marshall (2000)

As a compromise, we programmed the algorithm to write each generation to disk, being able to resume a session later for as many generations as desired. This allows us to take a rather conservative estimate with regards to the required number of generations, and to continue evolving a number of extra generations when deemed helpful.

4.1.3 Number of parents to breed offspring

In nature, offspring usually have either one or two parents. In a genetic algorithm, however, it could be interesting to investigate an increased number of parents. According to Eiben et al. (1995), while not having been extensively researched, one could hold an increase in performance when using more parents to be possible, though it might also lead to premature convergence.

As shown in figure 4, simply changing the number of parents of which parameters are randomly selected to three completely wiped away the general improvement in fitness observed when using two parents.

This might be a case of premature convergence, with "super-chromosomes" not being able to pass on enough of its characteristics to its offspring (Eiben et al., 1995). Further research on the effect of changing the amount of parents might be interesting.

4.1.4 Number of opponents for calculating fitness

When calculating the fitness, we let a solution play a number of matches against a randomly playing opponent to see how well it does. The more matches it plays, the more reliable it is (since a chance win will only partly affect the averaged fitness of that solution). However, a higher number of matches means it takes longer to calculate the fitness and hence run the genetic algorithm.

In figure 5 a typical progression of average fitness is shown for a run where each solution would battle two opponents each time their fitness was calculated. Though in the end the average fitness reaches a level comparable to that of our results in figure 2, development leading towards it is a lot less stable and knows more setbacks.

To err on the safe side, to generate our results we had each solution fight five opponents.

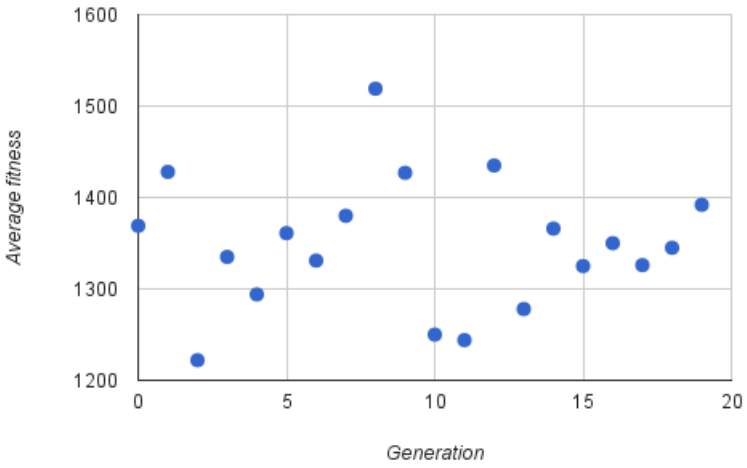

Figure 4: Average fitness by generation (three parents per solution)

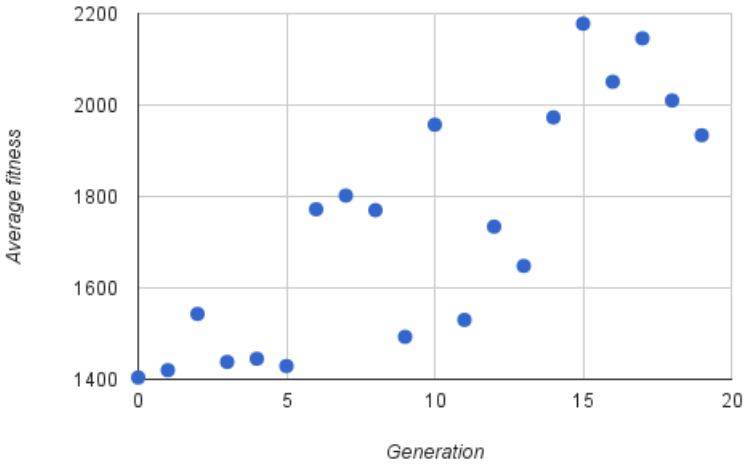

Figure 5: Average fitness by generation (two opponents per solution)

4.1.5 Likelihood of fit solutions to reproduce

Figure 1 shows how the parents to breed a new generation are chosen: the fitness of each solution is squared, meaning that fitter solutions have a larger chance to be selected for breeding. This very much improved fitness; omitting the squaring step could even lead to newer generations not really improving overall fitness, modifying it to take the power of e.g. five instead of two increases the speed at which the fitness improves over generations (i.e. it takes less generations to reach the same fitness). Care should however be taken to prevent over-fitting (Beasley et al., 1993b). As can be seen in figure 6 the quick increase in fitness can come at the cost of a radical drop in the average fitness once the algorithm has stabilized.

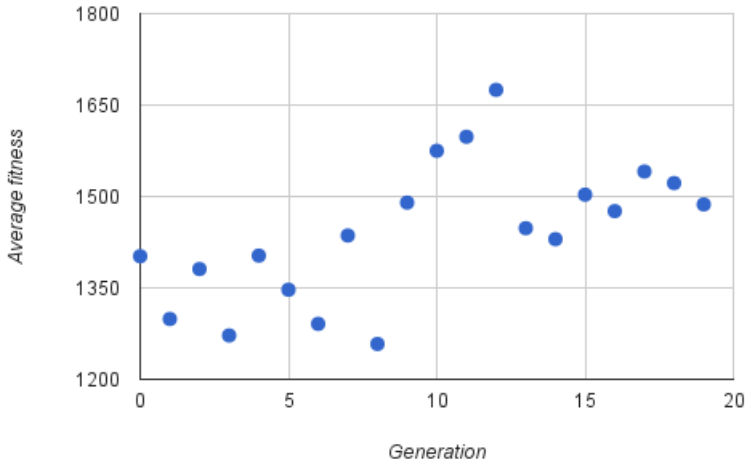

Figure 6: Average fitness per generation (fitness to the power of five)

4.2 Comparison to alternative approaches

Although other Stratego bots were hard to come by, we managed to get the code from Arts (2010). As far as Stratego bots go, this one ("StarBot") uses a relatively complex evaluation function — still, this function was hand-crafted using strategies as formulated by skilled human Stratego players.

This evaluation was coupled with a *-minimax search to allow the bot to try to look a few turns ahead, making a guess as to the value of unknown pieces of his opponent. We tested the performance of the bot we had evolved after playing tens of thousands of games against a randomly playing opponent against this bot, using different search depths. This way, even though Vicki could only consider the next move without taking into account long-term or even short-term consequences, we would have a certain measure of the quality of our bot.

When StarBot was configured with a search depth of one, the game was played from start to finish in negligible time — and our bot won. Increasing the search depth to two resulted in the game taking significantly longer than a depth of one, but still very fast. Yet again, StarBot could not best our evolved player. Subsequently, we tried a search depth of three. Waiting for the game to finish now took some patience, though it was still a lot faster than a game played by two human players would be. Nonetheless, even though Vicki could think ahead two turns less than StarBot, its strategy once more proved superior.

We then increased the search depth to four. On our machine, it now took StarBot on average longer than it would take a human to come up with a turn, and finishing the game required leaving the machine running overnight. At the Computer Stratego World Championship, the maximum allowed average time it could take a bot to contemplate a move was 5 seconds, and the maximum allowed time 15 seconds — both values StarBot running with a search depth of four would exceed, running on our machine. That said, it is a fairly weak machine, so running it on a modern desktop computer would probably be adequate for a search depth of four.

Apparently, slow and steady does indeed win the race, as this time StarBot did manage to beat Vicki. Leaving StarBot running for two days with search depth five only managed to complete a handful of moves, but without a doubt this would have crushed our current bot had we had the resources to finish the game.

It should however be noted that in between the production of StarBot and our algorithm, the rules of Stratego changed: in our implementation, players were not allowed to move back and forth on the same two squares more than three consecutive times. When StarBot was written, the maximum allowed number of consecutive movements on the same squares was five. Therefore, when pitting the two against each other, enforcement of the two-squares rule, as it is called, was disabled. The effect of this was that our algorithm assumed it could not move back and forth on the same squares more than three times in a row, whereas StarBot could happily continue repeating the same move. This way, our algorithm could be forced into moves it would rather not making, even when starting from an advantageous position. How much of an effect this has on the outcome of the duels is unknown, but it is something to be kept in mind. However, seeing as this is a disadvantage to our algorithm, it is still possible to draw conclusions from the matches it *did* win.

5 Conclusion

Due its large number of possible states and the fact that players have incomplete information, Stratego is a game that cannot simply be bruteforced. There has been some research trying to tackle the problem of making computers play Stratego, but all of those solutions were dependent on incorporating heuristics formulated by expert human Stratego players. This research tried to investigate the feasibility of approaching this problem with a genetic algorithm.

The results show that, even with a simple parameter-based approach to the genetic algorithm, a clear improvement can be observed. Solutions judging possible moves using a set of valuations evolved over as few as twelve generations

managed to achieve more and more convincing victories over an opponent playing randomly. Moreover, the resulting valuations provided insight in strategies that work well, as noted in de Boer (2012).

Whereas other attempts were dependent on the skill of their programmers and thus are unable to exceed the level of play of their creator, our approach could theoretically outperform the best human players. However, to really get a sense of whether this approach might eventually produce better results, it should compete with other artificial players. Luckily, even though obtaining the code of other research on artificial Stratego players proved troublesome, we finally managed to lay our hands on StarBot. Despite a small handicap due to a change in Stratego's rules, our evolved strategy was advanced enough to be able to beat an artificial player with a manually constructed evaluation function that looked three moves ahead, even though Vicki could only reason about the current move.

This indicates that the genetic approach is, indeed, a viable one to pursue to construct an artificial, well performing Stratego player. Section 7 outlines several paths that can be taken to improve upon these results to produce a high-quality artificial Stratego player. Possibly being able to develop a skilled artificial player for a game with imperfect information such as Stratego is a step forward for artificial game playing and, with it being one of its most-researched areas, artificial intelligence.

6 Discussion

Despite the clear improvement realized in fitness, there were several areas in which our implementation could have been better.

6.1 Mutation

When generating a new population, the only genetic operator used to initialize the offspring's parameter values is crossover. This means all parameter values the final solution can consist of have to be present in the initial population. Once a value has (possibly prematurely) "died out", there is no way it can be reintroduced.

To introduce more variety to the population, the genetic operator of mutation could have been introduced. Every generation each parameter will also have a chance to be modified a little bit, so new tactics can emerge. An interesting approach here could be to see whether the algorithm could dynamically adapt to the evolutionary phase it is in, balancing between crossover and mutation depending on the distribution of fitnesses (Beasley et al., 1993a).

Possibly, introducing mutation could speed up the evolutionary process, because it allows for a smaller population size. Since it is less important when parameter values completely vanish from a population (they can be reproduced or at least approximated by mutations of other values), less solutions could suffice.

6.2 Scout mobility

Although the game implementation did allow for the Scout to move as many squares in the same direction as possible without being blocked (which StarBot — see section 4.2 — profited from), our genetic algorithm did not have the possibility to evolve that behaviour. With this, the Scout became an ordinary piece, and one of the weakest at that. Moreover, one of the most important elements of many a Stratego strategy — exploring unknown pieces far away, sacrificing Scouts — could now play no role in the evolved strategy. This, too, might have skewed the results against StarBot to the disadvantage of our algorithm.

6.3 Dependence on human player

Even though the actual parameter values were evolved by the computer, determining which parameters we used and when they kicked in still required help of a skilled human player. Yet, even for a skilled human player it can be troublesome to come up with useful parameters, as has been observed by Samuel (2000): even though experts can have extensive knowledge of the game, it may prove difficult to articulate this knowledge to the programmer.

The advantage of using a parameter-based approach is that it is immediately clear what the evolved strategy entails. Simply looking at the evolved values gives an indication of the type of actions each piece has a preference for. It can also be used to test hypotheses about strategies that might or might not work well. If the algorithm is given the chance to develop this specific strategy, whether it actually does can be interpreted as an indicator of whether that strategy does or does not work well. Finally, it is relatively simple to implement.

That said, it would be a stretch to claim that the computer is actually learning to play Stratego by itself. Because the possible effects of each parameter need to be clearly predefined, they have to be rather generic, so most strategies are necessarily superficial.

7 Future Research

There are a variety of ways the genetic algorithm approach to writing an artificial Stratego player could be improved so as to achieve a higher level of play.

7.1 Evolve board positions

To be a successful Stratego player, a very important element of playing is arranging one's 40 pieces at the start of the game (De Boer, 2007). In our research we used the same predefined board setup every time. This is extremely predictable and only allows for very shallow strategies. Furthermore, it was a rather simplistic setup.

The approach to generating a board setup in De Boer (2007) involves generating a number of setups based on a large corpus of setups used in real games and selecting an actual setup according to some predefined heuristics. Another approach is to use a setup chosen randomly from a number of predefined manually selected setups Arts (2010) Schadd et al. (2009).

An obvious improvement for our approach would be to evolve the initial setups along with the strategies. Playing the same setup every time makes for a bad player (since his opponent can know the position of the flag at the start of the game) so it would be advisable to allow solutions to be able to use multiple setups (and introduce a certain measure of randomness to each) and to make the strategy be setup-dependent. In fact, board setup should be considered part of the strategy.

Alternatively, several good solutions from a population can be kept, and the final player could switch between any of those strategies so as to be more unpredictable in their strategy. During evolution using the same strategy every time does not influence the results, since their opponents do not incorporate previous playing habits in their decisions. Since both defensive and offensive strategies could be successful, the final player would then be able to play both types of strategies.

7.2 Piece-based parameter values

In our algorithm, we evolved a set of parameters for each available moveable rank. This allowed us to draw interesting conclusions (de Boer, 2012) about the sort of actions certain ranks should perform to achieve good performance (e.g. attack unmoved opponent pieces — which are more likely to be a Bomb — with relatively worthless pieces, i.e. a low rank and no special abilities such as the Miner's to defuse bombs). However, this makes for a relatively unsophisticated strategy compared to evolving a strategy per piece. Whereas we currently have ten different piece strategies (one for each moveable rank), we could have evolved 33 (one for each moveable piece, so even pieces with the same rank could evolve different strategies). These strategies would be bound to a certain board setup, as each piece could adapt its strategy to its specific board position (it would be superfluous to evolve separate tactics for pieces that might be of the same rank and start on the same position, or in other words, whose situation is exactly the same regardless of strategy).

Using piece-based parameter values in combination with evolving board setups, the algorithm could evolve more advanced tactics, such as a strategy with an aggressive left side of the board where strong pieces make some easy kills after exploration by explorers and a more defensive right side, where pieces of the same rank as pieces of the left side might rather want to flee than to attack unknown opponent pieces. Combine this with tactical placement of among others flag and bombs, and we have a strategy far more sophisticated than what our current algorithm has evolved.

The piece-based approach could also be used to evaluate board setups used in existing tactics by human players. If a person wants to utilise a defensive strategy against a known aggressive opponent, and considers using a certain setup, a strategy could be evolved building on that setup. If the pieces this person expected to act defensively appear to work better acting offensively with the considered initial setup, another setup can be considered and re-evaluated until a suitable defensive setup to start from has been devised.

7.3 Look ahead

Our algorithm considers all possible moves of a turn, evaluates the merits of each move based on the evolved values of each parameter, and performs the move with the highest valuation. Looking only at the next move VICKI can make itself is a pretty limited approach though, which does not take into account future implications of the move, and does not allow for the construction of longer-term tactics. In fact, the the opponent's way of playing is hardly taken into account.

The problem with Stratego is that it is a game of imperfect information. When evaluating a move, one cannot tell with which move the opponent would react best, since that depends on the ranks of the opponent's pieces which are unknown. A way to deal with this lack of information is to use one of the variants of the *-MINIMAX algorithms, which involves adding 'chance nodes' to the game search tree (W. and Ballard, 1983). This approach has been applied to Stratego by Arts (2010), albeit with a simple, manually constructed evaluation function. In fact, our algorithm could beat this algorithm when it was set to look no more than three moves in the future (the highest acceptable depth on our test machine), by only looking a single move ahead. Replacing the hand-crafted evaluation function with an evolved one could drastically improve performance. An interesting endeavour to undertake would be to try and define a set of parameters that actually incorporate information from multiple turns.

It should be noted, however, that this will significantly impact the speed of execution. Informal tests indicate that a search depth of 2 could be used during evolution on a reasonably powerful modern PC — perhaps even a depth of 3. Once evolution has finished, search depth can be increased for regular games to achieve similar speed to that achieved with the manually compiled evaluation function. Evolved strategies cannot explicitly make use of the capability to look ahead further, but performance will still be boosted by the ability to make more informed decisions.

7.4 Opponent modeling

The options our algorithm can contemplate are pretty limited with regard to the information it takes into account when determining the values of its parameters. Some of the most advanced reasoning it can do is base its options on whether a piece has moved yet; a piece having stood still the entire game having a greater chance of being a bomb or a flag.

The evolved strategy could be more sophisticated if it also had an internal model of the opponent's pieces, assigning probabilities to each piece about the rank it is likely to be. Such a model could incorporate knowledge such as that pieces that have already moved are definitely not Bombs or Flags, or how many pieces of a certain rank the opponent still has available. Incorporating these and other facts into a probability distribution for Stratego has been found to significantly improve performance by Stankiewicz and Schadd (2009).

Combining this opponent model with the genetic algorithm approach could potentially allow for far more specific strategies. For the parameter-based approach specifically this would enable parameters that are activated after a threshold is crossed in the probability that a certain piece is of a certain rank. This threshold, in turn, could also be evolved. Alternatively, a parameter's value could be dependent on the likelihood of a piece being of a certain rank,

with a higher probability resulting in more influence of that parameter.

Examples of parameters that could be added/improved with a probability distribution of the opponents pieces are the urge to attack an opponent (if the opponent is likely to be an opponent we can beat, attack; currently, the attack only proceeds when we are sure of the opponent's rank, i.e. when it has already attacked one of our pieces), or incorporating how likely it is that the opponent still believes a piece to be a bomb in deciding whether to move it for the first time.

7.5 Analyze gameplay

We looked primarily at the effects of the evolved player – does this player's strategy result in many wins, does he have many pieces left at the end of a game, etc. A thorough analysis of actual games played by the algorithm could provide more meaningful information on the effect of certain parameters in the fitness function. It could show whether a punishment for each piece the opponent has left does indeed lead to the algorithm avoiding capturing the flag early, whether we can tweak the rewards in the fitness function to generate a more offensive or defensive player, etc. Additionally, obvious flaws in the strategy could be detected with more detailed analysis to construct new parameters that could be used in the player's evaluation function.

References

Victor L. Allis. Searching for Solutions in Games and Artificial Intelligence. PhD thesis, University of Limburg, 1994. URL http://fragrieu.free.fr/SearchingForSolutions.pdf.

Sander Arts. Competitive play in stratego. Master's thesis, Maastricht University, March 2010. URL http://www.unimaas.nl/games/files/msc/Arts_thesis.pdf.

D. Beasley, D.R. Bull, and R.R. Martin. An overview of genetic algorithms: Part 2, research topics. University computing, 15(4):170–181, 1993a.

D. Beasley, RR Martin, and DR Bull. An overview of genetic algorithms: Part 1. fundamentals. University computing, 15:58–58, 1993b.

Roseline de Boer. Reachable level of stratego using genetic algorithms, 2012.

V. De Boer. Invincible. Master's thesis, TU Delft, 2007.

Kenneth De Jong. Learning with genetic algorithms: An overview. Machine Learning, 3(2-3):121–138, 1988. doi:10.1007/BF00113894.

A. Eiben, C. van Kemenade, and J. Kok. Orgy in the computer: Multi-parent reproduction in genetic algorithms. In Federico Morn, Alvaro Moreno, Juan Merelo, and Pablo Chacn, editors, Advances in Artificial Life, Lecture Notes in Computer Science, pages 934–945. Springer Berlin / Heidelberg, 1995. ISBN 978-3-540-59496-3. doi:10.1007/3-540-59496-5_354.

David Greenhalgh and Stephen Marshall. Convergence criteria for genetic algorithms. SIAM Journal on Computing, 30(1):269–282, 2000. doi:10.1137/S009753979732565X.

Stratego Original rules. Jumbo, 2006. URL http://www.jumbo.eu.

J.R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Stanford University, Department of Computer Science, 1990.

A. L. Samuel. Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 44(1.2):206 –226, jan. 2000. ISSN 0018-8646. doi:10.1147/rd.441.0206.

M.P.D. Schadd, M.H.M. Winands, and J.W.H.M. Uiterwijk. Chanceprobcut: Forward pruning in chance nodes. In Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on, pages 178 –185, sept. 2009. doi:10.1109/CIG.2009.5286476.

J. Schaeffer and H.J. Van den Herik. Games, computers, and artificial intelligence. Artificial Intelligence, 134(1-2):1–8, 2002.

Brian Sheppard. World-championship-caliber scrabble. Artificial Intelligence, 134(12):241 – 275, 2002. ISSN 0004-3702. doi:10.1016/S0004-3702(01)00166-7.

J.A. Stankiewicz and M.P.D. Schadd. Opponent modeling in stratego. In Proceedings of the 21st BeNeLux Conference on Artificial Intelligence (BNAIC09)(eds. T. Calders, K. Tuyls, and M. Pechenizkiy), pages 233–240, 2009.

C. Treijtel and L.J.M. Rothkrantz. Stratego expert system shell. In GAME-ON, pages 17–21, 2001.

Bruce W. and Ballard. The *-minimax search procedure for trees containing chance nodes. Artificial Intelligence, 21(3):327 – 350, 1983. ISSN 0004-3702. doi:10.1016/S0004-3702(83)80015-0.

A Value added to my bachelor

Performing the research taught me a number of things. First and foremost, I developed the commitment to actually performing research, enjoying the process moving from a vague idea on what to research, to obtaining actual results, to clearly writing it down in this thesis. Secondly, I of course learned a lot on the techniques used in this and related work, and about the current state of research in this area. Finally, I learned a lot on working rigidly, in a structured way gather information so as to also be able to write it down later so it can actually be useful.

B The code

At the time of writing, the code used to generate the results is located at https://gitorious.org/stratego and is licensed under the GNU General Public License version 2.