Go and Genetic Programming
Playing Go with Filter Functions

S.F. da Silva

November 21, 1996

Contents

1 Introduction

The purpose of this project has been to see if there is a future in looking for an evaluation function for an Alpha-Beta algorithm for the game go using genetic programming.

The reason I wanted to look at go is because the current go playing programs aren't really very good as compared to current chess playing programs for instance and also because I like the game.

I have been primarily looking at the 7×7 variant of go so as to be able to compare my results with other efforts in this direction, especially those of C.D. Rosin and R.K. Belew whose studies on co-evolution in genetic algorithms[1] included the game go. Another reason I used such a small board is because it makes the runs of the genetic program faster.

The largest part of this research has been to find the optimal parameters for finding an evaluation function of the type I called *filters* (see section refeval).

2 Go and genetic programming

There are at least two ways to use genetic programming (GP) in go or in any other two-player game. If you have an evaluation function you can use a GP to find the best search strategy. Or if you have a search strategy you can use a GP to find an evaluation function.

I chose the latter, mainly because I don't have an evaluation function for go. There are however many search strategies and getting one to test the evaluation functions on isn't so hard.

On the other hand, because there are so many moves to choose from in go (361 on a 19×19 board at the start of the game), most "simple" search strategies aren't that good. They are just too slow. So looking for a new search strategy would be a good idea for later research.

3 Description of the go board evaluation function

I will describe the type of function (S-expression) I will be looking at: it has as an argument a board configuration (i.e. a board with some black and white stones on it); the return value will be the evaluation of the board configuration. The higher the return value, the better it is for black.

The way this return value is calculated is by creating, from the given board, a new board which will represent, in a way, the expected configuration after the game has ended — that is the configuration in which the program expects the game to end given the present configuration. Thus by counting the territories on this board we will know the return value:

"return value" = "black territory" - "white territory"

Finally I will explain how the new board configuration is calculated: I have a function, f which will be the major component of the evaluation function; it has 25 arguments, or rather one 5×5 matrix. This is explained below. The return value of the function is the

value [1] of one point on the board. The new board is calculated from the old-one with the function *trans* which uses the function f to find the new value for each coordinate on the board. For each point (x, y) on the original board b I calculate $f(g)$ where g is a 5×5 matrix containing the values of 25 points in a square around (x, y) on b. On the new board, b', I set point (x, y) to $f(g)$. In more mathematical terms:

I make use of the following types and functions:

- V is a set of values. This is the set $\{< \text{EMPTY} >, < \text{EDGE} >, < \text{BLACK}_l >, < \text{WHITE}_l >\}$. Here l is the number of liberties of a stone; if that number exceeds 15 then $l = 15$.
- $LB,\ UB \in \mathbf{N}$, $LB < UB$. LB and UB are a lower bound and an upper bound for the coordinates of a go board.
- $C = [LB,\ UB]^2$. C is the set of coordinates on a go board.
- $board = V^C$. This is the type for a board configuration.
- $trans : board \rightarrow board$. The function *trans* converts a board configuration to a new board configuration.
- $g : board \times C \rightarrow V^5 \times V^5$. This function is only used to provide the next function, f, with the right data.
- $f : V^5 \times V^5 \rightarrow V$. This function is used for each point on the board to give it a new value.

The functions are defined like this:

$$trans(b) = b',\ \text{so that}$$

$$\forall (x, y) \in C : (b'(x, y) = f(g(b, (x, y))))$$

and

$$g(b, (x, y)) = \begin{pmatrix} b(x-2, y-2) & b(x-1, y-2) & b(x, y-2) & b(x+1, y-2) & b(x+2, y-2) \\ b(x-2, y-1) & b(x-1, y-1) & b(x, y-1) & b(x+1, y-1) & b(x+2, y-1) \\ b(x-2, y) & b(x-1, y) & b(x, y) & b(x+1, y) & b(x+2, y) \\ b(x-2, y+1) & b(x-1, y+1) & b(x, y+1) & b(x+1, y+1) & b(x+2, y+1) \\ b(x-2, y+2) & b(x-1, y+2) & b(x, y+2) & b(x+1, y+2) & b(x+2, y+2) \end{pmatrix}$$

We get the final board b_{final} like this: $b_{final} = trans^n(b)$. This b_{final} is used to evaluate the board configuration b. Together with an alpha-beta algorithm the function b_{final} can be viewed as a go program. Here n is a constant. This constant can either be set by hand or be determined by the GP. b_{final} is the evaluation function but, effectively, the function f is the S-expression I will be optimizing.

I got the idea for the function f from the bitmap-editor Paint Shop Pro [2] in which you can create one bitmap from another by applying a so-called filter. Such filters can be used for things like blur-effects or highlighting the edges of a drawing. I think this type

[1] This is not just one of the values *black stone*, *white stone*, *empty*, it can also be *edge*, meaning the edge of the board, and the liberties of a stone are also encoded (to a maximum of 15 liberties).

[2] Paint Shop Pro version SHAREWARE 3.12 · 32 is copyrighted ©1990-1995 to JASC, Inc. .

of function might do well as an evaluation function because what you are doing when you are playing go is looking at the board and estimating the number of points you are likely to get by, in your mind, finishing the teritories. This is what this type of function does. The only problem is that most of these functions finish the territory incorrectly.

To find out to whom a certain coordinate belongs you need to look at the entire board. An example of why this is necessary is the shicho or ladder (see Appendix A). That is why I wanted the board to be filtered more than once; in other words, this is the reason for n in b_{final}.

4 Fitness

4.1 Criteria for fitness

There are a couple of ways of looking at the fitness of a go program.

You could say a program is good if it makes "good shape" — that is, if it makes shapes that are considered efficient or strong. This is the way a go player might look at a program.

The definition of shape is a bit vague. It is a combination of stones that can occur at different places on a board.

You could also say that a program is good if it can beat a certain predetermined opponent; or, to extend this idea, if it finishes a game against that opponent with a certain minimum score. (The program loses with at most, say, twenty points or wins with at least five points.)

The ideal fitness would be a combination of both "good shape" and winning. In any case, "good shape" on its own is probably not enough for a good program. Winning would be a good criterium, but it is not that obvious that perfect play implies good shape. It is probable though.

4.2 Fitness and training the GP

Fitness is not only used to determine if the S-expression that rolls out of the GP is any good, but also to steer the GP in the direction of the optimal S-expression – to train the GP, so to say. In fact this is the main purpose of the fitness. In any case, the same ways of looking at fitness apply.

In the GP I have tried a number of ways of assigning fitness.

4.2.1 The tournament method

The first and easiest way of assigning fitness I tried, worked like this: In each generation the pool of S-expressions played a simple tournament. In the first round individual 1 played individual 2, 3 played 4, 5 played 6, and so on. In the second round the winner of the first game played the winner of the second game and so on. The losers were out of the tournament. I repeated this until I had only one individual left.

Each time an S-expression won a game it got a point. This way the one with the most points was the one who had won all its games and was most probably the best individual. The number of points an S-expression got I used as a measure for its fitness.

This method isn't very fair, because S-expressions losing in the first round might have won against other S-expressions, but they don't get a chance to prove themselves. I used this method anyway because it is fast. It takes about as many games as there are individuals in the pool.

4.2.2 The Wally method

Another training method I used was letting each individual in the pool play the program Wally[7]. The score against Wally was used as the fitness.

The advantage of this method is that, in the beginning, the pressure from the fitness is strong. For an individual to get a fitness value of more than the the minimum it would have to be able to make a living group at the very least.

The disadvantage is that the S-expressions can really only become as strong as Wally. After that there is no more pressure from the fitness. This was not the case with tournament evaluation.

It's like riding a bike in the first gear: you can gain speed relatively fast but after a while the pedals rotate too fast to speedup any more.

Another disadvantage is that the only information about go-strategies the S-expressions get is the information Wally gives them; another program, which might be just as weak, but which plays by a different strategy, might beat an S-expression that normally beats Wally.

4.2.3 The best-so-far method

This method combines the ideas of tournament evaluation and the Wally method. The individuals have a so called champion to beat but when he is beaten another, better-one, takes his place.

The way it works is, all S-expressions play the champion – the best individual so far – and their score constitutes their fitness. If there is a better individual, it will have a better fitness so it will be the champion for the next generation. This way the fitness pressure won't drop until a good S-expression has been found. This method doesn't have the advantage of having a strong champion to begin with though, so it might take a long time before a good S-expression is found.

4.2.4 Another Wally method – Wally-min

Through the course of the experiments I thought of another way to assign fitness to the S-expressions. It is an extension of the Wally method. In stead of playing just one game I let the individual play Wally a fixed number of times. The worst score for the individual is taken as the fitness.

This way the fitness is more reliable: in a big population there is always a chance that Wally "screws up" a game and when that occurs the individual that happens to be the opponent at that time gets an undeserved high fitness. By playing more games the chance of a wrong fitness is lower.

On the down side is the fact that it takes longer to evaluate the S-expressions.

Actually, this didn't work very well so, instead of just the worst score, I took a weighted sum of the worst score, the average score and the (squared) standard deviation of the

scores:

$$fitness = w_1 * worst\_score + w_2 * average\_score + w_3 * standard\_deviation$$

4.2.5 The Test-set method

This method I never used. Somewhere on WWW I found a collection of test-sets [4]. These were SGF-files [3] with special notations for good and bad moves.

I wanted to used these files for calculating the fitness by having each S-expression start with the configurations described in these files; then the S-expression would make a move and according to if the move was marked good or bad it would get points or lose them.

This would have been a "good shape" kind of fitness, the emphasis being on the moves, not on winning.

I expected this way of assigning fitness to be faster than playing entire games because the S-expressions would only have to make a few moves for each fitness evaluation. Unfortunately all test files were on 19×19 boards and the functions turned out to be much too slow on a board of this size. So, in the end, I didn't even try this method.

5 The Genetic Program

The GP consists of a function set, a selection and reproduction part (the kernel) and an evaluation part (the fitness function). I used lil-gp[9] for the kernel and my own fitness function made partly from wally.c[7].

5.1 The function set

The function set that I used to construct the S-expressions is:

f_if_black_ponnuki,
f_if_white_ponnuki,
f_if_eq,
f_if_weaker,
f_near_edge,
f_is_type,
f_add,
f_sub,
f_invert,

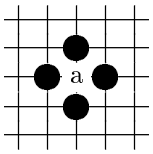
I will explain each function.

f_if_black_ponnuki

This function has two arguments. If the 5×5 matrix has a black *ponnuki* at the center, the first argument is returned, otherwise the return value is the second argument.

The following diagram shows what a (black) ponnuki looks like. It consists of the four black stones and the point at *a* should be empty. All other points can contain any value.

[3] See Appendix B.



Black Ponnuki.

In the function, the black stones can also be an edge for it to recognize the shape as a ponnuki. So the following diagram is also a ponnuki.

A ponnuki at the edge, in the corner.

f_if_white_ponnuki
: This function is the same as *f_if_black_ponnuki* except, it looks for a white *ponnuki*.

f_if_eq
: This function takes four arguments. It compares the first two and if they are the same the third argument is returned, otherwise the fourth is returned.

f_if_weaker
: This function takes four arguments. If the number of liberties of the first argument is less than the number of liberties of the second, the return value is the third argument, otherwise it is the fourth.

f_near_edge
: This function again takes two arguments; the first as a return value for if one of the points of the 5×5 matrix contains the value *edge*, the second for if none do.

f_is_type
: This function is like a *switch*-statement in *C* or a *CASE*-statement in *Pascal*: it looks at the first argument; if it is *edge*, the second argument is returned, if it is *empty*, the third, if it is a black stone, the fourth and finally, if it is a white stone the return value is the fifth argument.

f_add
: This function adds up its two arguments. If the sum exceeds the maximum number of liberties or if one of the arguments is *edge*, the sum is *edge*.

f_sub
: This function subtracts its first argument from its second. If the maximum number of liberties is exceeded or if one of the arguments is *edge*, the return value is *edge*.

f_invert
: This function "inverts" its one argument: a black stone becomes a white stone and a white stone becomes a black stone, both with the same number of liberties. All other values stay the same.

5.2 The terminal set

Of course with a function set you need a terminal set too. The terminal set is rather short:

f_erc_gen,

f_erc_gen
: This is the terminal symbol for the function set. The function generates two types of constants: *values* and *stones*.

  Values can be a black stone with l liberties, *<BLACK(l)>*, a white stone with l liberties, *<WHITE(l)>*, an empty space, *<EMPTY>* or the edge of the board, *<EDGE>*. *Stones* are pointers to values in the 5×5 matrix, $stone[x][y]$.

5.3 The fitness functions

As described in section 4.2 I have used two fundamentally different ways of assigning fitness. One produces absolute fitness values whereas the other only yields relative fitness values. I will give a short description of each.

5.3.1 An absolute fitness function

The fitness functions that produce absolute fitness values are the Wally method of section 4.2.2 and the Wally-min method of section 4.2.4. On whichever S-expression you use them, they always assign the same fitness to the same S-expression (on average, because Wally is not deterministic).

With the Wally method the fitness was calculated like this:

$$\mathit{fitness} = \mathit{result}(\mathit{game})$$

with

$$\mathit{result}(\mathit{game}) = \begin{cases} \textit{out-of-time score}, & \text{if the game was not ended in time.} \\ \textit{board-size}^2 + \textit{Wally's score} - \textit{S-expression's score}, & \text{otherwise} \end{cases}$$

The fitness of Wally-min was calculated as follows:

$$\begin{aligned} \mathit{fitness} &= w_1 * \mathbf{max}(r_1, \ldots, r_n) \\ &+ w_2 * \mu \\ &+ w_3 * \sigma^2 \end{aligned}$$

with

n the numer of games played.

$r_i = \mathit{result}(\mathit{game}_i), \forall i \in \{1, \ldots, n\}$ (This is the same *result*() as above)

$\mu = \frac{\sum_{i=1}^{n} r_i}{n}$

$\sigma^2 = \frac{\sum_{i=1}^{n} (\mu - r_i)^2}{n}$

5.3.2 A relative fitness function

For a relative fitness function I used the methods described in section 4.2.1 and 4.2.3. These functions do not produce values that make the S-expressions comparable with the rest of the world, as the Wally method does, but they do have the property that the fitness pressure doesn't drop before one of the best S-expressions is found. They will keep on assigning better fitness values as long as the S-expressions get better, even after the S-expressions win against Wally with a maximum score. The Wally method doesn't distinguish between two S-expressions beating Wally with the maximum score, nor, for that matter, between two S-expressions losing from Wally with the maximum score against. (Maybe I should point out here that Wally is a computer go program and the Wally method is a way of assigning fitness values to S-expressions by letting them play Wally.)

5.4 Explanation of the control parameters

Most parameters are described thoroughly in Koza's book Genetic Programming[3] and are standard so I won't go into these here, however some less conventional parameters may need some explaining.

In the first runs I used two subpopulations in stead of just one big population and every ten generations I exchanged a certain number of individuals between the populations: every tenth generation the five worst individuals from population two were replaced by the five best from population one and the five worst from population one with the five best from population two.

I used one selection method provided by lil-gp[9] but not mentioned in [3] and another I made myself by changing that selection method of lil-gp:

- method "best" which takes the best individual of that generation the first time one is selected for reproduction or crossover or mutation; the second time it takes the second best, etc.
- method "nth" which takes the n-th best individual of that generation every time an individual is selected for reproduction etc. . This selection method has one parameter, namely the number for n.

There are some application-specific parameters which don't have much to do with genetic programming. They apply to the go-playing part of the GP.

- The search depth; this is the number of plies the Alpha-Beta algorithm looks ahead.
- The board size; this is usually one of the sizes 9, 13 and 19, but it can be anything from 2 to 19. In this research I have used mainly size 7.
- The maximum number of moves, in case two players don't know when to stop. If this number is reached the game is ended and the out-of-time score is returned.
- The number of *sweeps*; this is the number of times the function *eval* is repeated the way it is explained at the end of section 3.
- The out-of-time score; this is the value that is taken as the score (for black) when the game isn't ended within the maximum number of moves.

5.5 Unchanging parameters

The following tables list the parameters I never changed. There are some other parameters that never changed but I found it better to leave them in the separate tables of each experiment.

Parameters that stayed the same for all (sub)populations.

Parameter	Value
Generative method for initial random population	ramped half-and-half
Maximum size for S-expressions created during the run	$D_c = 256$ nodes
Maximum tree depth for initial random S-expressions	$D_i = 4$
Probability of permutation	$p_p = 0\%$
Frequency of editing	$f_{ed} = 0$
Probability of encapsulation	$p_{en} = 0\%$
Condition for decimation	*NIL*

Application specific parameters.

Parameter	Value
Search depth	1ply
Board size	7
Maximum number of moves per game	147ply
Number of *sweeps*	2

6 The experiments

I have run the GP with all ways of assigning fitness described in section 5.3. I ran each experiment five times with the same parameters but with different random-seeds. (I would have liked to do more runs for each experiment, but time didn't allow that). After I found which fitness function worked best I ran some more experiments and varied the parameters to find optimal values.

6.1 Finding the best fitness function

The first runs I used to find the fitness function that worked best. I took a set of arbitrary control parameters and varied the fitness function for each experiment.

Fitness functions.

Run	Fitness function and parameters	
BSF	best-so-far method	
	Out-of-time score	0
TRN	tournament method	
	Out-of-time score	0
WAL	Wally method	
	Out-of-time score	0

The major numerical parameters.

Parameter	Value
Number of subpopulations	$S = 2$
Population size for each subpopulation i ($i \in \{1, 2\}$)	$M_i = 50$
Maximum number of generations to be run	$G = 100$

The minor numerical parameters for subpopulation 1.

Parameter	Value
Probability of crossover	$p_c = 80\%$
Probability of reproduction	$p_r = 5\%$
Probability of choosing internal points for crossover	$p_{ip} = 90\%$
Probability of mutation	$p_m = 15\%$
Probability of choosing internal points for mutation	$p_{ip} = 50\%$

The minor numerical parameters for subpopulation 2.

Parameter	Value
Probability of crossover	$p_c = 0\%$
Probability of reproduction	$p_r = 5\%$
Probability of mutation	$p_m = 95\%$
Probability of choosing internal points for mutation	$p_{ip} = 20\%$

The qualitative variables for subpopulation 1.

Parameter	**Value**
Selection method for reproduction	"best"
Selection method for crossover	fitness proportionate (roulette wheel selection)
Selection method for mutation	fitness proportionate (roulette wheel selection)
Type of fitness used for selection	Adjusted fitness

The qualitative variables for subpopulation 2.

Parameter	**Value**
Selection method for reproduction	"best"
Selection method for mutation	fitness proportionate (roulette wheel selection)

The results.

Fitness function	Average fitness
Best-so-far	98.00
Tournament	97.52
Wally	93.94

The best fitness function turned out to be the Wally method. To my disappointment the other two didn't do at all well. They didn't even get off the mark (except for one run with the tournament method but that was probably just a fluke).

6.2 The Wally method

Going on with just the Wally method I tried some different population dynamics. The two subpopulations was just a wild try. I went on with the more usual one population. Later (see section 6.4.1) the two subpopulations turned out to be not so bad.

6.2.1 Crossover and mutation, fifty-fifty

Because in the previous experiments I noticed that the populations got very uniform soon, I tried my next experiment with a higher chance for mutation. I also changed the selection methods for crossover and mutation from roulette wheel to tournament selection because I was told this was better.

The major numerical parameters.

Parameter	**Value**
Population size	$M = 100$
Maximum number of generations to be run	$G = 100$

The minor numerical parameters.

Parameter	**Value**
Probability of reproduction	$p_r = 4\%$
Probability of crossover	$p_c = 48\%$
Probability of choosing internal points for crossover	$p_{ip} = 90\%$
Probability of mutation	$p_m = 48\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for reproduction	"best"
Selection method for crossover	tournament, size 2
Selection method for mutation	tournament, size 2
Type of fitness used for selection	Adjusted fitness

Fitness function parameters.

Parameter	**Value**
Name	Wally method
Out-of-time score	0

The results

Experiment	Fitness value
2 pops	93.94
1 pop	86.82

The results were on average about seven points of territory better than the experiment with two subpopulations using the Wally method, so that was a big improvement.

6.2.2 Mutation only

Taking things one step further I tried an experiment with mutation as the only operator. No more crossover and also no more reproduction, so this experiment, unlike all the previous experiments, was non-elitist.

The major numerical parameters.

Parameter	Value
Population size	$M = 100$
Maximum number of generations to be run	$G = 100$

The minor numerical parameters.

Parameter	Value
Probability of reproduction	$p_r = 0\%$
Probability of crossover	$p_c = 0\%$
Probability of mutation	$p_m = 100\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	Value
Selection method for mutation	tournament, size 2
Type of fitness used for selection	Adjusted fitness

Fitness function parameters.

Parameter	Value
Name	Wally method
Out-of-time score	0

The results

Experiment	Fitness value
Crossover + mutation	86.82
Only mutation	85.20

This experiment did one or two points better than the previous-one. Not such a big improvement.

6.2.3 $(1+1)$

Hearing J.vd Hauw's good results with $(1+1)$[2] I decided to try that on go.

$(1+1)$ is the method used in Evolutionary Strategies[4] in which you have one parent from which you create one child with mutation. You then take the best of those two as the parent for the new generation.

A generalization of this method is $(\mu + \lambda)$ in which you have μ parents and create λ children and select the best μ individuals from the parents and the children for the new generation.

Because lilgp v1.0 had no option for (1+1) I simulated it by taking a population of size 2 and each generation created one individual by reproduction (the parent) and another-one by mutation (the child). I used selection method "best" for the parent and roulette wheel selection for the child. This way most of the time the parent would be selected for mutation (as should be in (1+1)) — I hadn't thought of the selection method "nth" yet.

The major numerical parameters.

Parameter	**Value**
Population size	$M = 2$
Maximum number of generations to be run	$G = 10000$

Fitness function parameters.

Parameter	**Value**
Name	Wally method
Out-of-time score	0 later 49

The minor numerical parameters.

Parameter	**Value**
Frequency of reproduction	$p_r = 50\%$
Frequency of mutation	$p_m = 50\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for reproduction	"best"
Selection method for mutation	fitness proportionate (roulette wheel selection)
Type of fitness used for selection	Adjusted fitness

The results

Experiment	Fitness value
Mutation only	85.20
$(1+1)$ with out-of-time score 0	87.30
$(1+1)$ with out-of-time score 49	84.06

[4] For more information on Evolutionary Strategies see [8].

Looking at some of the games in this experiment I noticed that those that ended in a tie were somewhat strange: the individual had found a flaw in Wally so that the same sequence was repeated infinitely. Tough this is an interesting result it is not what I was looking for so I ran a similar run with the out-of-time score set to 49 which meant that such a "tie" would now count as a (maximum) loss for the S-expression. Actually such a situation means that the game isn't over — and never will be over — so you can't say who has played best (both players have played lousy), but I didn't want it to be counted as a good result, and a tie didn't work, so I just set it to maximum loss.

Because the results of $(1 + 1)$ with out-of-time score 49 were better than those with a population of 100 and because in those runs I hadn't seen any out-of-time games, I concluded, wrongly perhaps, that $(1 + 1)$ was the best option so far and I went on with that.

6.2.4 One crossover + one mutation

I also tried a variation on $(1 + 1)$; I took a population of size 2 and created the new generations by taking the two individuals and creating one child by crossover and taking one individual by roulette wheel selection and creating a child by mutation.

The major numerical parameters.

Parameter	**Value**
Population size	$M = 2$
Maximum number of generations to be run	$G = 10000$

The minor numerical parameters.

Parameter	**Value**
Frequency of crossover	$p_c = 50\%$
Probability of choosing internal points for crossover	$p_{ip} = 90\%$
Frequency of mutation	$p_m = 50\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for crossover	"best"
Selection method for mutation	fitness proportionate (roulette wheel selection)
Type of fitness used for selection	Adjusted fitness

Fitness function parameters.

Parameter	**Value**
Name	Wally method
Out-of-time score	49

The results

Experiment	Fitness value
$(1 + 1)$ with out-of-time score 49	84.06
1 cross + 1 mut	87.75

This didn't work very well, probably because there was no elitism. To give you an idea of how it worked, figure 1 shows the best fitness value of each generation. This figure shows the average fitness of five runs (the lower the value the better the fitness). In the individual runs the fitness went a bit lower, but it is clear that there is no gradual improvement of the fitness.

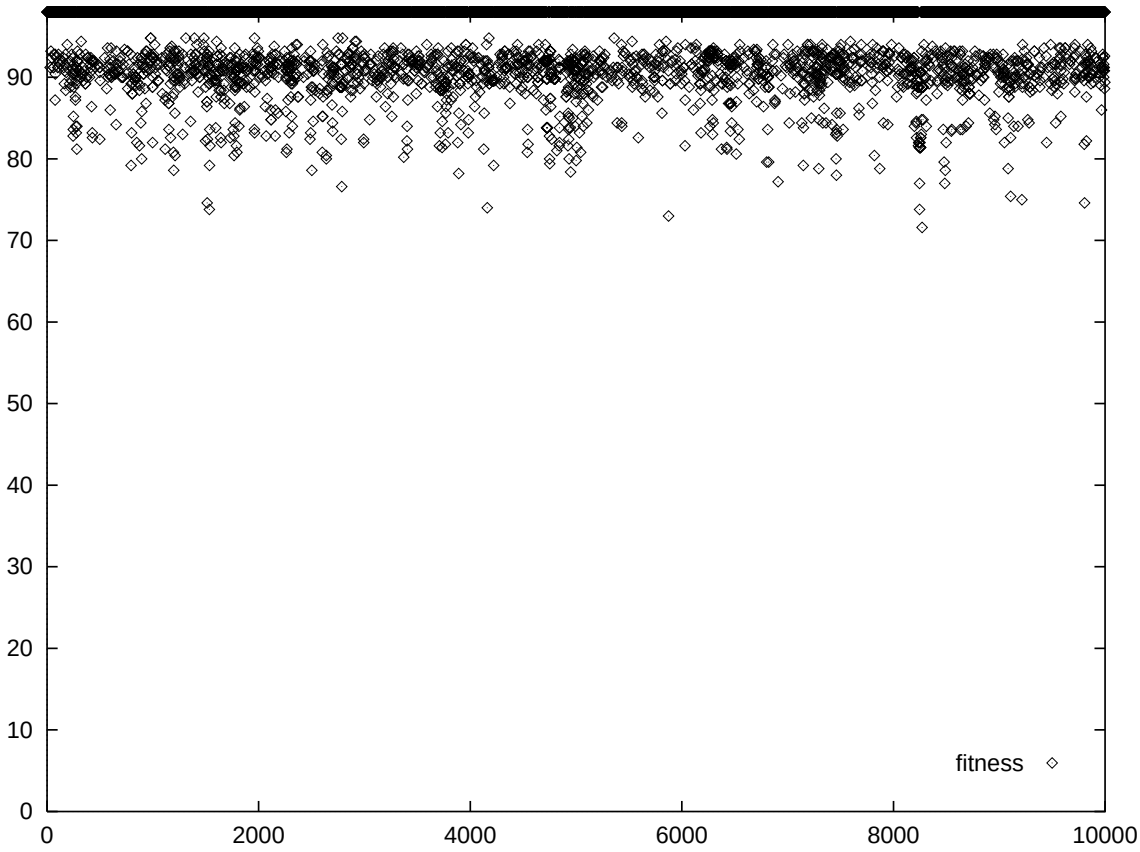


Figure 1: fitnesses of the best individuals of "one crossover + one mutation"

6.3 Another improvement on the fitness function: Wally-min

When I looked at each run to see how many games the best individual of each run would win the results weren't that good. Of a hundred games the individuals would usually win four or five by a relatively large difference, but the rest of the games they didn't even get a point. I used Wally-min on the next runs to get more consistent individuals. The other parameters were those of $(1 + 1)$ of section 6.2.3.

First I used only the worst score of five games as the fitness. To get the fitness up a bit faster I tried 98 times the worst score plus once the average score. I used 98 times the worst score because the average could take a value between 0 and 98. This way I could extract the average and the worst score from the fitness value.

Fitness function parameters.

Parameter	**Value**
Name	Wally-min method
Out-of-time score	49
Number of games for one fitness evaluation	5
Weight for minimum score	1, later 98
Weight for average score	0, later 1
Weight for σ^2 of scores	0

The major numerical parameters.

Parameter	Value
Population size	$M = 2$
Maximum number of generations to be run	$G = 10000$

The minor numerical parameters.

Parameter	Value
Frequency of reproduction	$p_r = 50\%$
Frequency of mutation	$p_m = 50\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	Value
Selection method for reproduction	"best"
Selection method for mutation	fitness proportionate (roulette wheel selection)
Type of fitness used for selection	Adjusted fitness

The results

Experiment	Fitness value
$(1 + 1)$ with Wally method	84.06
$(1 + 1)$ with Wally-min using worst score	73.37
$(1 + 1)$ with Wally-min using worst + avg	70.86

Wally-min worked well, but the first increase in fitness using only the worst score wasn't until the 4000^{th} generation. Using a combination of worst score and average score worked better.

6.4 The Wally-min method

I ran the experiments done with the Wally method again but now with Wally-min. So, all the experiments in this section 6.4 have the following fitness function parameters.

Fitness function parameters.

Parameter	Value
Name	Wally-min method
Out-of-time score	49
Number of games for one fitness evaluation	5
Weight for minimum score	98
Weight for average score	1
Weight for σ^2 of scores	0

The following experiments will all be compared with their Wally method counterparts. All of them are compared with the Wally method with out-of-time score 0, except "one crossover + one mutation" (section 6.4.4); this-one is compared with the Wally method with out-of-time score 49, because I didn't do the experiment with out-of-time score 0.

6.4.1 Two subpopulations

The major numerical parameters.

Parameter	Value
Number of subpopulations	$S = 2$
Population size for each subpopulation i ($i \in \{1, 2\}$)	$M_i = 50$
Maximum number of generations to be run	$G = 100$

The minor numerical parameters for subpopulation 1.

Parameter	Value
Probability of crossover	$p_c = 80\%$
Probability of reproduction	$p_r = 5\%$
Probability of choosing internal points for crossover	$p_{ip} = 90\%$
Probability of mutation	$p_m = 15\%$
Probability of choosing internal points for mutation	$p_{ip} = 50\%$

The minor numerical parameters for subpopulation 2.

Parameter	Value
Probability of crossover	$p_c = 0\%$
Probability of reproduction	$p_r = 5\%$
Probability of mutation	$p_m = 95\%$
Probability of choosing internal points for mutation	$p_{ip} = 20\%$

The qualitative variables for subpopulation 1.

Parameter	Value
Selection method for reproduction	"best"
Selection method for crossover	fitness proportionate (roulette wheel selection)
Selection method for mutation	fitness proportionate (roulette wheel selection)
Type of fitness used for selection	Adjusted fitness

The qualitative variables for subpopulation 2.

Parameter	Value
Selection method for reproduction	"best"
Selection method for mutation	fitness proportionate (roulette wheel selection)

The results

Experiment	Fitness value
2 pops with Wally	93.94
2 pops with Wally-min	64.16

I was surprised by the results of this experiment. They were not at all what I had expected. The results were about 6 points better than the best of the other runs I had done so far.

6.4.2 Crossover and mutation, fifty-fifty

The major numerical parameters

Parameter	**Value**
Population size	$M = 100$
Maximum number of generations to be run	$G = 100$

The minor numerical parameters.

Parameter	**Value**
Probability of reproduction	$p_r = 4\%$
Probability of crossover	$p_c = 48\%$
Probability of choosing internal points for crossover	$p_{ip} = 90\%$
Probability of mutation	$p_m = 48\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for reproduction	"best"
Selection method for crossover	tournament, size 2
Selection method for mutation	tournament, size 2
Type of fitness used for selection	Adjusted fitness

The results

Experiment	Fitness value
crossover and mutation 50-50 with Wally	86.82
crossover and mutation 50-50 with Wally-min	70.92

This experiment produced about the same results as $(1 + 1)$, just as it did with the Wally method as a fitness function. For reference:

Results of $(1 + 1)$

Experiment	Fitness value
$(1 + 1)$ with Wally	87.30
$(1 + 1)$ with Wally-min	70.86

6.4.3 Mutation only

The major numerical parameters.

Parameter	**Value**
Population size	$M = 100$
Maximum number of generations to be run	$G = 100$

The minor numerical parameters.

Parameter	**Value**
Probability of reproduction	$p_r = 0\%$
Probability of crossover	$p_c = 0\%$
Probability of mutation	$p_m = 100\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for mutation	tournament, size 2
Type of fitness used for selection	Adjusted fitness

The results

Experiment	Fitness value
Mutation only with Wally	85.20
Mutation only with Wally-min	74.20

The results were better than with the Wally method as a fitness function, but there wasn't as much an improvement as there was with the previous experiments.

6.4.4 One crossover + one mutation

The major numerical parameters.

Parameter	**Value**
Population size	$M = 2$
Maximum number of generations to be run	$G = 10000$

The minor numerical parameters.

Parameter	**Value**
Frequency of crossover	$p_c = 50\%$
Probability of choosing internal points for crossover	$p_{ip} = 90\%$
Frequency of mutation	$p_m = 50\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for crossover	"best"
Selection method for mutation	fitness proportionate (roulette wheel selection)
Type of fitness used for selection	Adjusted fitness

The results

Experiment	Fitness value
1 crossover + 1 mutation with Wally	87.75
1 crossover + 1 mutation with Wally-min	61.46

Just as in section 6.4.1 the improvement here was great and unexpected.

6.5 $(\mu + \lambda)$

From this point on I have been mainly occupied with optimizing the parameters for the generalization of $(1 + 1)$, namely $(\mu + \lambda)$. Because of the time each run takes I haven't got very far with this.

6.5.1 $(1 + \lambda)$

The next experiments are all $(1 + \lambda)$ with the same parameters as the last $(1 + 1)$, that is, with 98 times the worst score plus the average score. For completeness I have also included $(1 + 1)$.

The purpose of these experiments was to find the optimal value of λ for $\mu = 1$.

The varying parameters.

	$(1 + 1)$	$(1 + 5)$	$(1 + 10)$	$(1 + 15)$
Population size	$M = 2$	$M = 6$	$M = 11$	$M = 16$
Maximum nr. of generations	$G = 10000$	$G = 2000$	$G = 1000$	$G = 667$
Frequency of reproduction	$p_r = 50\%$	$p_r = 17\%$	$p_r = 9\%$	$p_r = 6\%$
Frequency of mutation	$p_m = 50\%$	$p_m = 83\%$	$p_m = 91\%$	$p_m = 94\%$

The minor numerical parameters.

Parameter	**Value**
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for reproduction	"nth", $n = 1$
Selection method for mutation	"nth", $n = 1$
Type of fitness used for selection	Adjusted fitness

Fitness function parameters.

Parameter	**Value**
Name	Wally-min method
Out-of-time score	49
Number of games for one fitness evaluation	5
Weight for minimum score	98
Weight for average score	1
Weight for σ^2 of scores	0

The results

Experiment	Fitness value
$(1 + 1)$	70.86
$(1 + 5)$	84.55
$(1 + 10)$	72.96
$(1 + 15)$	86.13

Of the values for λ that I have tried, $\lambda = 1$ got the best results. And $\lambda = 10$ did almost as good. However $\lambda = 5$ and $\lambda = 15$ got worse results by an average of about 10 points. That is, on average they got about 10 points less in a game against Wally than $(1 + 1)$ and $(1 + 10)$. This leads me to conclude that I haven't done enough tests yet to tell which value for λ is best or even if there is a best value.

6.5.2 $(2 + \lambda)$

I also tried four experiments with two parents in stead of one ($\mu = 2$). I wanted these runs to be comparable with the runs of $(1 + \lambda)$, and the best way to make them comparable was, I thought, to take for each $(1 + \lambda)$ a population $(2 + 2\lambda)$.

The purpose of these experiments was similar to those of $(1 + \lambda)$: to see which value of λ worked best for $\mu = 2$. The second goal was to see if $\mu = 1$ was better or $\mu = 2$.

The varying parameters.

	$(2 + 2)$	$(2 + 10)$	$(2 + 20)$	$(2 + 30)$
Population size	$M = 4$	$M = 12$	$M = 22$	$M = 32$
Maximum nr. of generations	$G = 5000$	$G = 1000$	$G = 500$	$G = 333$
Frequency of reproduction	$p_r = 50\%$	$p_r = 17\%$	$p_r = 9\%$	$p_r = 6\%$
Frequency of mutation	$p_m = 50\%$	$p_m = 83\%$	$p_m = 91\%$	$p_m = 94\%$

The minor numerical parameters.

Parameter	**Value**
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for reproduction	"nth", $n = 1$ for first individual, $n = 2$ for second
Selection method for mutation	"nth", $n = 1$ for first half, $n = 2$ for second half
Type of fitness used for selection	Adjusted fitness

Fitness function parameters.

Parameter	**Value**
Name	Wally-min method
Out-of-time score	49
Number of games for one fitness evaluation	5
Weight for minimum score	98
Weight for average score	1
Weight for σ^2 of scores	0

The results

Experiment	Fitness value
$(2 + 2)$	79.93
$(2 + 10)$	75.51
$(2 + 20)$	80.68
$(2 + 30)$	80.48

The best value for λ of the four I have tried was $\lambda = 10$. This corresponds with $\lambda = 5$ in the $(1 + \lambda)$ runs which was the second worst there.

The over all results of $(2 + \lambda_2)$ were worse than those of $(1 + \lambda_1)$.

6.5.3 $(2 + \lambda + \kappa)$

I tried crossover on $(2 + \lambda)$. I called this $(2 + \lambda + \kappa)$, or in general, $(\mu + \lambda + \kappa)$, with μ the number of parents, λ the number of children by mutation, and κ the number of children by crossover.

This notation leaves the questions of which parents to take for crossover, but in my experiments with $(2 + \lambda + \kappa)$ I always took parent one as the first parent and parent two as the second.

Again I wanted these experiments to be comparable with $(1 + \lambda)$ or actually with $(2 + \lambda)$, so I took the population sizes to be (almost) the same in both experiments: for $(2 + \lambda)$ I took $(2 + (\lambda - 2) + 2)$. However, I didn't think using only crossover made any sense. That is why I haven't done $(2 + 0 + 2)$. In stead I did $(2 + 2 + 1)$: one child by crossover and one for each parent by mutation.

The varying parameters.

	$(2+2+1)$	$(2+8+2)$	$(2+18+2)$	$(2+28+2)$
Population size	$M = 5$	$M = 12$	$M = 22$	$M = 32$
Maximum nr. of generations	$G = 3333$	$G = 1000$	$G = 500$	$G = 333$
Frequency of reproduction	$p_r = 40\%$	$p_r = 16\%$	$p_r = 9\%$	$p_r = 6\%$
Frequency of crossover	$p_c = 40\%$	$p_c = 16\%$	$p_c = 9\%$	$p_c = 6\%$
Frequency of mutation	$p_m = 20\%$	$p_m = 68\%$	$p_m = 82\%$	$p_m = 88\%$

The minor numerical parameters.

Parameter	**Value**
Probability of choosing internal points for crossover	$p_{ip} = 90\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Selection method for reproduction	"nth", $n = 1$ for first individual, $n = 2$ for second
Selection method for crossover	"nth", $n = 1$ for first individual, $n = 2$ for second
Selection method for mutation	"nth", $n = 1$ for first half, $n = 2$ for second half
Type of fitness used for selection	Adjusted fitness

Fitness function parameters.

Parameter	**Value**
Name	Wally-min method
Out-of-time score	49
Number of games for one fitness evaluation	5
Weight for minimum score	98
Weight for average score	1
Weight for σ^2 of scores	0

The results

Experiment	Fitness value
$(2+2+1)$	70.01
$(2+8+2)$	76.92
$(2+18+2)$	74.73
$(2+28+2)$	82.29

The results of $(2+\lambda+2)$ were similar to those of $(1+\lambda)$; just as $(1+1)$, $(2+2+1)$ worked best and $(2+18+2)$ after that. In three of the four experiments $(2+\lambda_2+2)$ did slightly better than the corresponding $(1+\lambda_1)$. Only $(2+18+2)$ scored below $(1+10)$.

Since $(2+\lambda_2+2)$ did better than $(1+\lambda_1)$, the over all results of $(2+\lambda_2+2)$ were better than those of $(2+\lambda_2')$. It seems crossover does help.

6.6 Continuing with best-so-far

I wanted to see how best-so-far did with an initial population of already fairly good individuals. I took the best runs I had so far — this was $(2+2+1)$ — and let them continue for another 10000 evaluations with best-so-far as a fitness function. In this case that was 3333 generations.

The major numerical parameters.

Parameter	**Value**
Population size	$M = 5$
Maximum number of generations to be run	$G = 3333$

The minor numerical parameters.

Parameter	**Value**
Frequency of reproduction	$p_r = 40\%$
Frequency of crossover	$p_c = 40\%$
Probability of choosing internal points for crossover	$p_{ip} = 90\%$
Frequency of mutation	$p_m = 20\%$
Probability of choosing internal points for mutation	$p_{ip} = NIL$ (uniform selection over all points)

The qualitative variables.

Parameter	**Value**
Initial population	last population of (2+2+1)
Selection method for reproduction	"nth", $n = 1$ for first individual, $n = 2$ for second
Selection method for crossover	"nth", $n = 1$ for first individual, $n = 2$ for second
Selection method for mutation	"nth", $n = 1$ for first half, $n = 2$ for second half
Type of fitness used for selection	Adjusted fitness

Fitness function parameters.

Parameter	**Value**
Name	Best-so-far
Out-of-time score	49

The results

Experiment	Fitness value
$(2+2+1)$ with Wally-min	70.01
$(2+2+1)$ continued with best-so-far	95.72

The results were very bad. The individuals had got worse.

7 Conclusion

7.1 Approach

In my research I have tried to find the best parameters for arriving fast at a good evaluation function for an alpha-beta algorithm applied to go. As you can see by studying the results I have done this by first optimizing the fitness function and then trying to find the best value for the other parameters.

Halfway through optimizing the population parameters I found that some things went wrong with the fitness function I was using. The individuals seemed to get stuck at tie games – the fitness got better until a tie was reached, then it froze. This was because I counted a game that ran out of time as a tie.

I also found that a good fitness didn't imply a good result against Wally all the time; usually it meant that in about ten games of a hundred an S-expression with a good fitness got a good result, the other times the result was a maximum loss.

After tuning the fitness function by setting the out-of-time score to 49 and having the individuals play five games in stead of just one, I continued optimizing the population parameters.

Finally I filled up some gaps and retried the best-so-far fitness function.

7.2 The results

The following table contains the average of the results of 100 games against Wally for best individual of each experiment, averaged over five runs and the standard deviation of those five runs. The first number in a cell is the average fitness, the second is the standard deviation. The results here are all of fitness values returned by the Wally method. A value greater than 49 corresponds with a loss; a value below 49 is a win.

If a cell contains an **X** it means that I haven't done that experiment because I thought it wouldn't give a good result. For instance, I have only done two experiments with fitness function best-so-far because it was clear to me this function would always be worse than the Wally method. For a table of section references to the results see appendix C.

"Goodness" of the best individuals.

	2 pops	1 pop	only mut	1 cross + 1 mut
Best-so-far	98.00, 0.00	**X**	**X**	**X**
Tournament	97.52, 0.97	**X**	**X**	**X**
Wally, oot 0	93.94, 5.65	86.82, 4.19	85.20, 6.78	**X**
Wally, oot 49	**X**	**X**	**X**	87.75, 6.92
Wallymin, worst	**X**	**X**	**X**	**X**
Wallymin, worst + avg	64.16, 8.61	70.92, 10.77	74.20, 9.47	61.46, 5.77

"Goodness" of the best individuals.

	$(1+1)$	$(1+5)$	$(1+10)$	$(1+15)$
Best-so-far	**X**	**X**	**X**	**X**
Tournament	**X**	**X**	**X**	**X**
Wally, oot 0	87.30, 6.64	**X**	**X**	**X**
Wally, oot 49	84.06, 7.60	**X**	**X**	**X**
Wallymin, worst	73.37, 16.35	**X**	**X**	**X**
Wallymin, worst + avg	70.86, 13.47	84.55, 3.89	72.96, 12.58	86.13, 4.17

"Goodness" of the best individuals.

	$(2+2)$	$(2+10)$	$(2+20)$	$(2+30)$
Wallymin, worst + avg	79.93, 7.74	75.51, 7.37	80.68, 10.50	80.48, 8.36

"Goodness" of the best individuals.

	$(2+2+1)$	$(2+8+2)$	$(2+18+2)$	$(2+28+2)$
Wallymin, worst + avg	70.01, 12.10	76.92, 14.58	74.73, 14.53	82.29, 12.25

"Goodness" of the best individuals.

	$(2+2+1)$ continued
Best-so-far	95.72, 2.14

As you can see there are no wins. But these values are all averages. I had one run, with the parameters of section 6.5.1, $(1+10)$, in which the best individual won *all* its games by one point (a fitness value of 48).

7.3 About the results

The best results were with one crossover and one mutation (that is a population of size 2!) using Wally-min (section 6.4.4) and the initial two subpopulations (of size 50 each) with Wally-min (section 6.4.1). The parameters of these two experiments have nothing to do with each-other so it is hard to say why these two gave the best results.

Two experiments that worked well too were $(1+1)$ and $(2+2+1)$ of sections 6.5.1 and 6.5.3. $(2+2+1)$ was the better of these two. In fact in three of the four cases $(2+\lambda_2+\mu_2)$ worked better than the corresponding $(1+\lambda_1)$.

It was hard to compare the individual results of $(1+\lambda_1)$ and $(2+\lambda_2+2)$ with those of the corresponding $(2+\lambda_2')$, but on the whole $(2+\lambda_2')$ did worse.

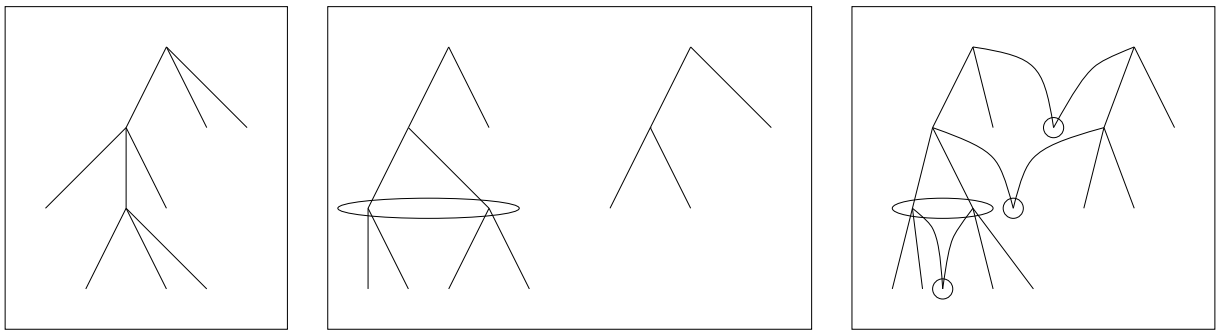
All this seems to imply that crossover is important. I will try to give an explanation for these results.

I think $(1+\lambda)$ isn't that good because it touches only a small part of the search space (see figure 2a). After each generation a few possible solutions (S-expressions) of the search space, near the parent, are examined; the best of these solutions is taken as the parent for the next generation.

$(2+\lambda)$ should have worked better because more ground is covered. In stead of walking along one path, you are searching near two paths. However, it is likely that one parent will produce two children that are better than the children of the other parent, and then you are just searching along one path again, albeit on two sides of it. (See figure 2b; the oval indicates the point where you jump from two paths to one.) This is probably

why $(2 + \lambda)$ did worse; almost the same space was covered, but with the use of more individuals.

The way $(2 + \lambda + \kappa)$ should work, is that you are not just searching along two paths, but also in-between. There is still a big chance that one parent will produce the two best individuals of a generation, which means that, in the next generation, your search is less broad again. Figure 2c shows this. The oval shows the point where the search narrows. The small circles mark the crossover-children. This does not explain why $(2 + \lambda_2 + 2)$ did better than $(1 + \lambda)$.



a b c

Figure 2: search paths

Actually the pictures in figure 2 are not entirely correct: a parent of one generation can also be chosen as a parent for the next generation, which would make the chance that the search path is narrowed too soon even higher.

What I think would do best is a $(2 + 2 + 1)$ search in which each parent for crossover must be chosen from the children of different parents, or from the crossover-child. I will say more about this in section 8.

I think the reason the S-expression got worse when I used best-so-far on a pre-generated population (section 6.6) is that they had been trained to beat Wally which always played black, so they always played white, and with best-so-far the best individual had to take Wally's place as the "champion", playing black. Because they hadn't been trained for playing black the champions weren't good at it, so they were easily beaten by a weaker S-expression.

7.4 Wally-min vs. Best-so-far

The fitness function that worked best was Wally-min. It was to be expected that a fitness function that was similar to the function that expressed the "goodness" of the resulting S-expressions would work better than a fitness function, like best-so-far, that had no knowledge at all about the valuation criterium.

I think, however, that best-so-far or a similar fitness function produces go-functions that play better against many opponents than the individuals trained with Wally-like fitness functions.

Using Wally to find opponents that can beat Wally works, but if you do this the opponents get over-trained; they can only beat Wally and not other opponents of the same strength.

I have found that using Wally as an opponent works remarkably well for finding faults in Wally itself. An example of this is the tied games I got when using an out-of-time score of 0, as I mentioned in section 6.2.3. I imagine that if you use another hand-made go-program as a fitness function you could use genetic programming to find faults in that program.

7.5 About the evaluation function itself

The speed of the evaluation function was a bit disappointing to me. Of course I knew it would be slow, but not a slow as it turned out to be. In combination with the alpha-beta algorithm with a search depth of one ply it took the program almost a minute, on a board of size 19×19, to make one move. For genetic programing this is just too slow.

As for the power of the function, the results show that it is possible to find a function that plays go better than random; I even found one that was better than Wally or, at least, it could beat Wally. This of course is not a big accomplishment but it does show that the function can get better, maybe with a more sophisticated fitness function.

8 Future research

There are a few things I have come across that I thought deserve a look into, but which didn't fit in this project.

- While writing section 7 I realized that the population dynamics of $(\mu + \lambda)$ and $(\mu + \lambda + \kappa)$ I had been using might not have been that good. As I mentioned there, those dynamics could make the search too narrow too soon. For example in $(2 + 2 + 1)$, where each generation the new parents are simply the two best of the five individuals (two parents and three children), there is a big chance that those two best individuals are (too) closely related.

  Continuing the example of $(2 + 2 + 1)$, I think it would be better to choose the parents as follows: lets say parent a creates child a_1 and parent b creates child b_1. Using crossover parents a and b could create child c. Now the new parents a' and b' should be chosen from the sets respectively $\{a, a_1, c\}$ and $\{b, b_1, c\}$, making sure c is not chosen twice. This example can easily be generalized to $(\mu + \lambda + \kappa)$.

  I think it would be interesting to compare the performance of the method I have described here and the normal $(\mu + \lambda)$ or $(\mu + \lambda + \kappa)$.

- Something that bothered me was that I couldn't really use the Alpha-Beta algorithm. Because of the enormous number of board configurations that have to be generated for each move I could only search to a depth of one ply. On a 7×7 board the number of configurations that have to be searched increases by a factor of 7^2 with each ply.

  To solve this problem you can design a search strategy better adapted to go. When I am playing go and I want to figure out which move I should make, I don't try every possible move in my mind, only the ones I "feel" could be good moves for me or for my opponent. It could be interesting to design a search strategy that does something like this.

I had in mind a function like my filter function which takes the current board configuration but which, in stead of giving a new board configuration, assigns a value to each possible move. Then the best, say, ten of these or the ones with a high enough value can be tried. You can use genetic programming to find such a function.

One problem is that you need an evaluation function to use with the search strategy. For this you could use one of my solutions, but I am not sure they would be good enough.

- Because the S-expressions get over trained by using Wally as a trainer, it could be better to use a number of different opponents in stead; even using Wally in different ways – once as usual, once by giving Wally the go boards rotated by 90°, etc. – would probably be better.

- My fitness functions, best-so-far and tournament evaluation, didn't work, but it would be worth a try using co-evolution as described by Rosin and Belew[1] because they used a scheme in which two strategies, one for black and one for white, are co-evolved. Another advantage of this method is the way fitness is assigned: the S-expressions get better fitnesses if they can beat many different opponents or opponents that no-one else can beat.

A Go terminology

A.1 Liberties

The liberties of a string of stones are the unoccupied adjacent points of that string. In figure 3 the points marked m are all the liberties of the black string. Point n is not a liberty of the string.

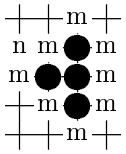


Figure 3: liberties

A.2 Atari

A string of stones is said to be *in atari* when it has only one liberty left. Figure 4 shows a group of black stones in atari.

A.3 Live groups

A group is alive when it has at least two free spaces that can never be filled by the opponent. Figure 5 shows an example of a black group that is alive. Points a and b can never be filled by white because a move at either point would be suicide unless the other point is filled in first.

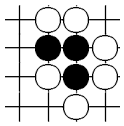


Figure 4: black stones in atari

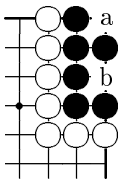


Figure 5: a live group.

A.4 Ponnuki

The pattern of figure 6 of the four stones and the empty space at *a* is a ponnuki.

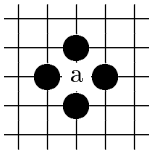


Figure 6: a ponnuki.

A.5 Shicho

The shicho, or ladder, is a configuration in which a player can catch a group by a series of ataris. Figure 7 shows an example of a shicho. Black can capture the two white stones by atari-ing at the left of the two stones. If there is a white stone at or near *a* black can't capture the white stones that easily. A white stone at *a* would be called a *ladder breaker* or a *shicho-atari*.

The shicho with its shicho-atari is one of the reasons why go is so much more complex than games like chess.

A.6 Komi

The komi is the number of points given to the white player in advance to compensate for black's having the first move. This is usually 5.5 points. The half point is to prevent ties.

B SGF-files

SGF stands for Smart Game File Format. It is meant to be a standard file format to exchange machine-readable games, problems, and opening libraries. A more complete

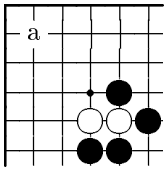


Figure 7: a shicho.

description is given by Martin Müller[5]. I will give the syntax and semantics of the subset of SGF that I have used.

I have used SGF in two ways. One to save the games played by the GP and the other to read test configurations for the GP to play. I have used the SGF format because there are many programs available that support it (for go at least), one of which is MGT[6], so I didn't have to write an application to view the games.

Below is the syntax of the SGF files in BNF form. First I will give a brief summary of the conventions I have used:

"..."	:	terminal symbols.
[...]	:	option; occurs at most once.
{...}	:	repetition; zero or more times.
|	:	exclusive or.

Bold-font characters should be read as terminals. The symbols letter, digit and character mean the obvious, except that "[" and "]" should be written as "\[" and "\]" respectively and "\" as "\\".

Collection	::=	{GameTree}		
GameTree	::=	"(" Sequence {GameTree} ")"		
Sequence	::=	Node {Node}		
Node	::=	";" {Property}		
Property	::=	**AB** {"["Move"]"}	| **AE** {"["Move"]"}	| **AW** {"["Move"]"}
	|	**B** "["Move"]"	| **BM** "["Triple"]"	| **C** "["Text"]"
	|	**CR** {"["Move"]"}	| **GM** "["Number"]"	| **KM** "["Real"]"
	|	**MA** {"["Move"]"}	| **PB** "["Text"]"	| **PW** "["Text"]"
	|	**RG** {"["Move"]"}	| **TE** "["Move"]"	| **W** "["Move"]"
Move	::=	letter letter		
Number	::=	["+" | "-"] digit {digit}		
Real	::=	Number ["." {digit}]		
Text	::=	{character}		
Triple	::=	"1" | "2"		

Not all strings derivable from this grammar are valid SGF-files. For instance, a node can not contain both a **B** attribute and an **AB** attribute, and each attribute can only appear once per node, some even only once per game tree. Furthermore between properties you can put as many white-characters as you like.

I will explain the semantics of the grammar with the help of an example:

(;**PB**[Wally] **PW**[individual 1] **GM**[1] **SZ**[19] **KM**[5.5]
;**B**[pd] ; **W**[dp] ; **B**[dd]
;**W**[pp] **C**[a standard opening])

The **PB** and **PW** fields contain the names of the black player and the white player respectively. The **GM** field represents the game type; 1 means go, 2 is othello, 3 is chess. **SZ** is the size of the board. **KM** is the komi. All these fields should be in a game tree only once.

The fields **B** and **W** contain the moves for black and white. The first letter is the first coordinate of the board, the second is the second coordinate. Unlike on most (real, wooden) go boards the letter 'j' is not left out and the second coordinate is not a number but a letter. **C** contains the comments for the node.

The next example shows how good and bad moves can be marked in a game:

(;**SZ**[13] **GM**[1] **AB**[jd] **AW**[dd][jj]
;**B**[dj] **TE**[2]
CR[ek][gc]
MA[aa][ma][mg]
RG[ek][ma])

Attributes **AB** and **AW** tell you to add black and white stones respectively to the board at the coordinates given in the lists, no matter whether there are already stones there or not. An attribute **AE** would mean that the stones at the coordinates in the following list should be removed from the board (if there are any).

An attribute **TE** in a node with a move means that this is a good move. If the value of the attribute is 1, it is a fairly good move; if it is 2, it is an even better move. The attribute **BM** means the opposite: 1 is a bad move, 2 is an even worse move.

The attributes **CR**, **MA** and **RG** are used here in a different manner than they were intended in the original SGF-format. **CR** marks good moves for whose ever turn it is in the node, so in the example it marks two good moves for black. **MA** marks the bad moves. **RG** works like the 2 in **TE** and **BM**: if a move is marked both **CR** and **RG** it is very good and if it is marked both **MA** and **RG** it is very bad.

C Section references to the results

These tables contains the section numbers of the results from the table in section results.

"Goodness" of the best individuals.

	2 pops	1 pop	only mut	1 cross + 1 mut
Best-so-far	6.1	**X**	**X**	**X**
Tournament	6.1	**X**	**X**	**X**
Wally, oot 0	6.1	6.2.1	6.2.2	**X**
Wally, oot 49	**X**	**X**	**X**	6.2.4
Wallymin, worst	**X**	**X**	**X**	**X**
Wallymin, worst + avg	6.4.1	6.4.2	6.4.3	6.4.4

"Goodness" of the best individuals.

	$(1+1)$	$(1+5)$	$(1+10)$	$(1+15)$
Best-so-far	**X**	**X**	**X**	**X**
Tournament	**X**	**X**	**X**	**X**
Wally, oot 0	6.2.3	**X**	**X**	**X**
Wally, oot 49	6.2.3	**X**	**X**	**X**
Wallymin, worst	6.3	**X**	**X**	**X**
Wallymin, worst + avg	6.3, 6.5.1	6.5.1	6.5.1	6.5.1

"Goodness" of the best individuals.

	$(2+2)$	$(2+10)$	$(2+20)$	$(2+30)$
Wallymin, worst + avg	6.5.2	6.5.2	6.5.2	6.5.2

"Goodness" of the best individuals.

	$(2+2+1)$	$(2+8+2)$	$(2+18+2)$	$(2+28+2)$
Wallymin, worst + avg	6.5.3	6.5.3	6.5.3	6.5.3

"Goodness" of the best individuals.

	$(2+2+1)$ continued
Best-so-far	6.6

References

[1] Christopher D. Rosin and Richard K. Belew,*Methods for Competitive Co-evolution: Finding Opponents Worth Beating*, in Proceedings of the Sixth International Conference on Genetic Algorithms. L.J. Eshelman (ed.), pp. 373-380, 1995

[2] Koen van der Hauw, *Evaluating and Improving Steady State Evolutionary Algorithms on Constraint Satisfaction Problems*, Master Thesis, IR-96-21, July 1996.

[3] John Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, The MIT Press, 1992.

[4] Martin Müller, *Computer Go Test Collection*, http://nobi.ethz.ch/martin/special.html

[5] Martin Müller, *Smart Game File Format*, http://nobi.ethz.ch/martin/sgfspec.html

[6] *My Go Teacher*,ftp://ftp.pasteur.fr/pub/Go/mgt/*

[7] Bill Newman (newman@tcgould.tn.cornell.edu), *wally.c*, ftp://ftp.pasteur.fr/pub/Go/comp/wally.sh.Z

[8] Hans-Paul Schwefel, *Evolution and Optimum Seeking*, John Wiley & Sons,inc., 1995

[9] Doug Zongker (zongker@isl.cps.msu.edu), *lil-gp 1.0*, http://isl.cps.msu.edu/GA/software/lil-gp/