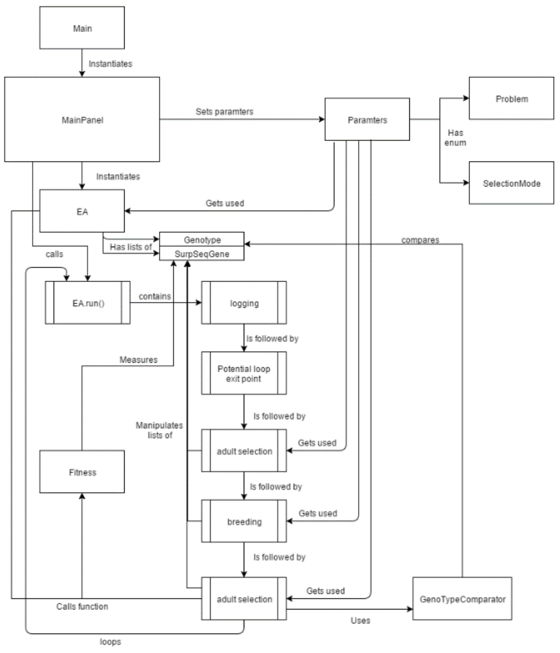
Project 2 IT3708 Torgeir

A)



In my implementation of the EA code, I have chosen to use a Java Swing window to set the parameters and run the program. I do the plotting by pasting the logged data in a spreadsheet and use the spreadsheet tools to visualize the data. See the diagram for a more details on the classes used and the program flow.

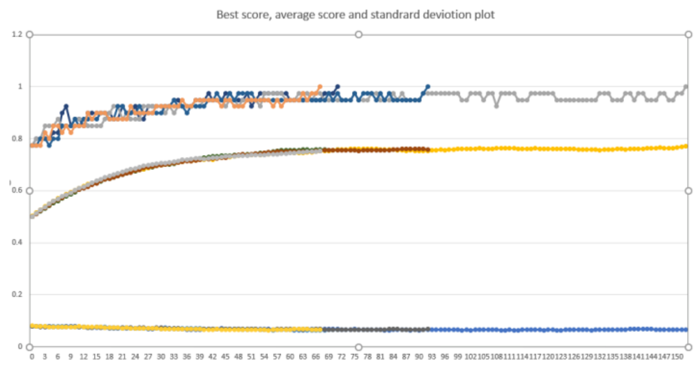
The code is modular in the sense that the problems and parent selection mechanisms each have their own methods and enums to identify which ones to use. For adult selection there is basically only one mechanism, which depends on the numbers of desired children and adults and the Boolean value for whether adults should survive. So creating a new mechanism for adult selection might require more changes. To implement a new parent selection mechanism one only needs to write a method, which has to produce the right amount of children. Then a new enum kind needs to exist, and the option must be added to the GUI. For new problems the process is similar. A new fitness function must be written. Fitness scores are double values where 1.0 is the perfect score. If the problem uses anything else than a bitstring, then a new subclass of Genotype is required. This subclass must implement everything that needs to be different from the superclass.

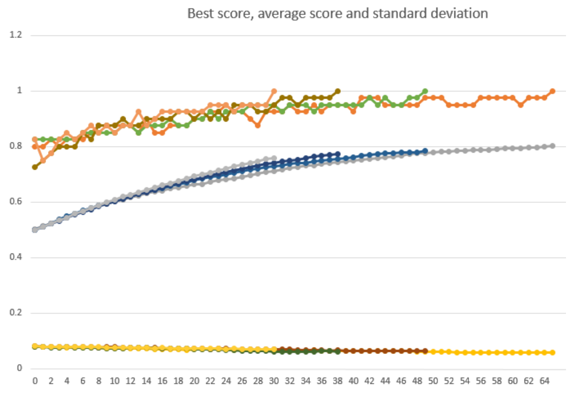
The general code for the fitness function looks like this:

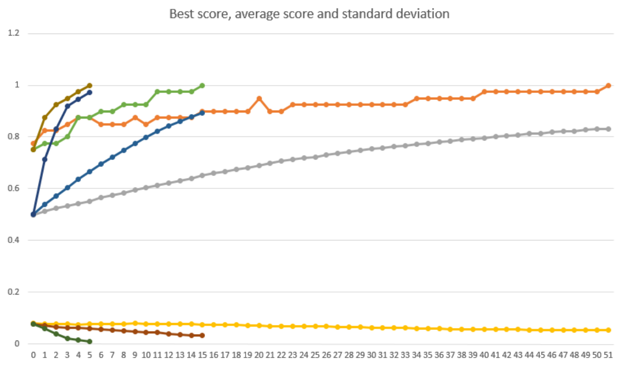
```
public static double function(Genotype genotype){
		double ret = 0.0;
		switch(Parameters.problem){
		case LOLZ:
			ret = lolzScoring(genotype);
			break;
		case ONEMAX:
			ret = oneMaxScoring(genotype);
			break;
		case LOCALSEQ:
			ret = localSurpSeqScoring((SurpSeqGene) genotype);
			break;
		case GLOBALSEQ:
			ret = globalSurpSeqScoring((SurpSeqGene) genotype);
			break;
		default:
			ret = Math.random();

		}
		genotype.fitness = ret;
		return ret;
	}
```

B)

Using a mutation rate of 1%, I reached the conclusion that a population size of 4000 was good for finding the solution to the one max problem within 100 generations. This is with a mutation rate of 1% and a crossover rate of 50%. This plot has four runs with this setting. Occasionally it will take over 100 generations because there's not enough exploration at the end, so works very slowly at that point. Since the curves flatten a good bit before the 100 generations mark, it should be possible to find better parameters to increase the selection pressure. I will do this by changing the mutation rate. Making the mutation rate higher will only introduce more variation, which is bad at this point, because the variance goes both ways. What we ultimately want is to get the parent selection to pick adults with scores below the average. We don't want the mutation rate to be too low, because then improvement will slow down. In the second plot I have been varying the mutation rate down to 0.05%, which took 78 generations. The next rate up was 0.1%, which terminated in a little more than 30 generations on average. I also tried different crossover rates. However, none of these seemed to give consistent results that were better than the ones I got with a crossover rate of 50%

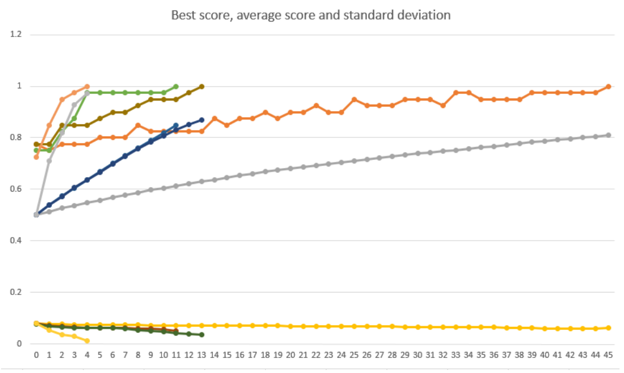




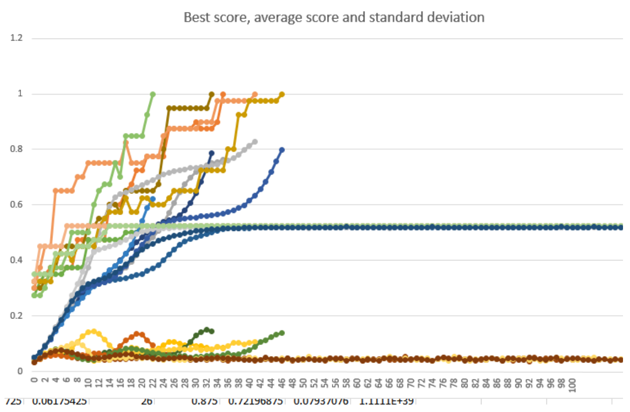


Experimenting with parent selection was easy. The fourth mechanism I implemented was uniform selection, which is interesting because using that means that all the selection pressure has to be applied elsewhere. However, when using full generation replacement, there is no other place to apply selection pressure. Therefore, it made no sense to test the uniform parent selection. The other ones did provide drastically different results. Sigma scaling terminated after 15 generations. Tournament selection with a tournament size of 500 and a probability of having random winners at 5%, made the program terminate after only 5 generations. Clearly tournament selection is the best selection mechanism in this case.

Random.org provided me with a random bitstring, which I used as the target for the program. The result is practically identical to the usual version of the problem. This is no surprise, because the fitness function does the exact same thing just with a different target.

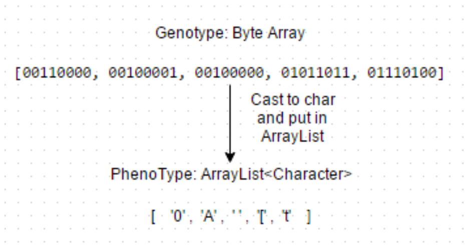


C)



The LOLZ problem is interesting because of the trap. I tried all of the relevant selection mechanisms, and all of them had the property that they would get caught in the trap and never terminate. I plotted eight runs with sigma scaling. It is evident that there is variation among the successful runs, and they find the optimal solution quickly. This is probably due to the fitness function for this problem giving a huge detriment to mutations in an otherwise uniform prefix, so these bad mutations are easy to reject in the selection process. Meanwhile the trapped runs just stay with the leading zeroes followed by a random suffix. The chance of randomly finding a sufficiently long prefix of ones is incredibly unlikely.

D)


Genotype: Byte Array
[00110000, 00100001, 00100000, 01011011, 01110100]
Cast to char and put in ArrayList
PhenoType: ArrayList<Character>
['0', 'A', ' ', '[', 't']

For the surprising sequence problems, I also used a byte array to represent the genome. The phenotype used by the fitness function is different, being an ArrayList of Characters. When printing the phenotypes, I print the byte values.

To evaluate local sequences, I simply scan the sequence. I create a string consisting of the characters representing the symbols that follow each other in the sequence. I put this string in an ArrayList of Strings. For each combination of characters, I check if an equal string is already in the list, if that is the case, then the method terminates, and the returned value is the length of the prefix that did not fail. Effectively all sequences will get a score of at least 2 divided by the length of the sequence, because only starting at the third symbol can the property fail. For global sequences the process is more complicated. I use a double for loop, where the outer loop iterates over the end index of a pair of two symbols in the sequence and the inner loop iterates over all the possible preceding indexes. I calculate the space between these symbols. Then I have a two dimensional ArrayList of Strings, because there is one list of each amount of spacing there is between symbols. So when I calculate the space between symbols, I know which list to use. I use this list the same way as the single list of the local sequence. The point of having the outer loop iterate over the end index of the pair is that for each iteration in this loop, I approve another length step of the sequence. This way I score the sequence according to how much of the prefix satisfies the problem.

Local sequences

S	L	Population	Generations	Sequence
3	10	10	29	1, 2, 0, 2, 2, 1, 1, 0, 0, 1
5	26	100, Adults survive	28	4, 1, 3, 0, 2, 1, 1, 0, 3, 1, 2, 3, 3, 4, 4, 3, 2, 0, 1, 4, 0, 0, 4, 2, 2, 4
10	101	1000, Adults survive	158	2, 3, 7, 9, 8, 6, 4, 2, 0, 5, 3, 9, 1, 9, 0, 3, 1, 5, 1, 3, 3, 2, 5, 2, 7, 3, 0, 2, 4, 6, 6, 1, 6, 2, 1, 4, 7, 1, 7, 5, 8, 8, 7, 7, 2, 2, 9, 3, 6, 8, 4, 9, 2, 8, 9, 6, 0, 0, 9, 9, 5, 9, 4, 5, 4, 4, 3, 4, 0, 7, 6, 9, 7, 0, 6, 3, 5, 6, 7, 8, 0, 1, 0, 8, 3, 8, 5, 5, 0, 4, 1, 1, 8, 1, 2, 6, 5, 7, 4, 8, 2

15	210	1000 adults, 2000 children, Adults survive	262	9, 8, 4, 1, 5, 5, 1, 0, 4, 10, 8, 11, 10, 3, 7, 14, 12, 14, 6, 10, 11, 2, 2, 3, 10, 12, 4, 6, 5, 0, 11, 0, 0, 10, 10, 1, 4, 14, 11, 13, 12, 6, 1, 13, 8, 2, 0, 12, 8, 8, 10, 14, 7, 2, 9, 11, 4, 5, 4, 0, 6, 6, 2, 13, 2, 5, 14, 3, 9, 6, 11, 1, 2, 11, 6, 12, 9, 12, 3, 3, 6, 13, 0, 1, 9, 9, 14, 4, 12, 5, 12, 1, 1, 11, 3, 8, 13, 10, 7, 10, 9, 1, 14, 8, 9, 3, 14, 0, 3, 13, 9, 10, 13, 1, 3, 0, 5, 13, 5, 3, 4, 13, 3, 11, 9, 7, 13, 7, 5, 8, 12, 13, 11, 7, 11, 11, 14, 5, 11, 8, 0, 8, 1, 7, 9, 0, 14, 13, 14, 10, 2, 8, 7, 12, 7, 0, 13, 6, 9, 4, 7, 3, 1, 10, 0, 2, 10, 6, 14, 1, 8, 14, 2, 4, 8, 6, 4, 4, 11, 12, 12, 2, 14, 9, 13, 4, 9, 2, 6, 3, 2, 12, 11, 5, 2, 7, 6, 8, 3, 5, 6, 0, 9, 5, 10, 4, 2, 1, 12, 0
20	350	1000 adults, 2000 children, adults survive	433	11, 7, 3, 8, 3, 14, 7, 12, 1, 7, 19, 11, 0, 12, 19, 4, 17, 19, 7, 6, 1, 2, 6, 10, 11, 4, 3, 0, 10, 13, 7, 7, 14, 2, 7, 2, 12, 4, 6, 16, 15, 19, 2, 1, 6, 4, 8, 1, 18, 15, 8, 15, 1, 0, 11, 6, 18, 5, 17, 13, 19, 14, 17, 10, 1, 19, 12, 0, 6, 17, 1, 1, 4, 4, 13, 4, 11, 10, 2, 19, 10, 8, 16, 2, 13, 15, 10, 19, 17, 12, 18, 10, 16, 5, 6, 6, 5, 16, 12, 11, 1, 14, 1, 5, 2, 18, 16, 17, 4, 1, 15, 9, 13, 0, 18, 11, 11, 2, 5, 12, 6, 8, 4, 14, 16, 14, 11, 5, 4, 12, 10, 14, 15, 7, 4, 15, 2, 4, 19, 1, 16, 10, 7, 17, 15, 18, 13, 12, 8, 11, 15, 4, 9, 11, 12, 12, 17, 3, 1, 12, 14, 18, 0, 8, 14, 12, 7, 15, 16, 19, 6, 3, 10, 0, 19, 9, 17, 18, 2, 16, 3, 19, 3, 15, 5, 1, 13, 14, 19, 16, 4, 7, 11, 14, 6, 2, 2, 3, 9, 6, 19, 8, 2, 0, 3, 4, 16, 13, 11, 16, 6, 7, 8, 7, 18, 3, 3, 16, 18, 4, 10, 15, 14, 13, 5, 7, 9, 2, 11, 18, 14, 9, 1, 8, 0, 15, 11, 3, 5, 9, 19, 19, 5, 11, 17, 11, 13, 13, 10, 17, 16, 0, 16, 1, 10, 12, 9, 8, 5, 13, 18, 19, 13, 1, 9, 15, 17, 6, 14, 14, 5, 3, 18, 1, 3, 12, 2, 8, 18, 9, 18, 17, 17, 2, 10, 6, 9, 0, 17, 8, 13, 16, 7, 10, 9, 3, 2, 17, 9, 4, 18, 8, 10, 3, 13, 3, 11, 8, 12, 5, 19, 15, 6, 0, 2, 9, 9, 14, 3, 6, 15, 0, 4, 5, 15, 3, 17, 14, 0, 13, 9, 16, 9, 5, 10, 5, 5, 0, 0, 9, 7, 0, 5, 14, 10, 10, 18, 18, 7, 13

Global sequences:

S	L	Population	Generations	Sequence
3	7	10	6	1, 2, 0, 0, 1, 0, 2
5	12	100, Adults survive	5	1, 2, 4, 2, 3, 0, 1, 3, 2, 0, 4, 4
10	25	1000, Adults survive	30	8, 1, 2, 0, 7, 9, 5, 9, 4, 3, 2, 6, 6, 5, 3, 6, 0, 8, 4, 7, 1, 9, 2, 8, 5
15	35	1000, Adults survive	40	2, 0, 8, 2, 13, 1, 4, 6, 13, 11, 9, 3, 5, 7, 11, 14, 14, 1, 9, 7, 8, 5, 6, 10, 6, 12, 0, 2, 11, 4, 13, 3, 12, 8, 10
20	49	1000 adults, 2000 children, Adults survive	247	10, 5, 17, 0, 1, 7, 11, 2, 6, 15, 3, 18, 19, 9, 12, 16, 4, 16, 12, 18, 7, 8, 3, 2, 14, 17, 7, 10, 11, 14, 13, 19, 8, 13, 0, 3, 5, 2, 15, 8, 1, 9, 4, 18, 10, 17, 5, 15, 6

E)

My ranking for the difficulty of these problems goes like this: OneMax, LOLZ, Global Surprising Sequences and finally Local Surprising Sequences.

The OneMax problem is obviously the easiest one, since the EA doesn't have to make mutations in a specific point to find better solutions. The scoring is also based on the full solution. Meanwhile the LOLZ problem has a trap, which can be extremely hard for the EA to get out of. It also needs the mutations to happen at the end of the already found prefix, since mutations earlier will make the score worse, and mutations later will not impact the score. My implementation of the surprising sequences uses the longest legal prefix for the score, so this is similar to the LOLZ problem. The difference being that mutations in the prefix might not be bad. Since mutations are not binary, the mutations need to be to the right value as well as being in the right positions. So beneficial mutations become rarer since the symbol set also matters. Local seeming harder because there are much longer solutions to the local version than the global one for the same symbol set. The difficulty of finding long solutions seems harder than finding shorter ones that are rarer.