Metaheuristics TP7

Ning Tientso

Introduction

Genetic Programming is the application of the idea of the Genetic Algorithm (seen from the previous assignment) to individuals which are computer programs. The challenge is to represent computer programs in a way that can still utilize concepts of crossover and mutation, such that these concepts do not syntactically break the execution of the programs. This can be achieved by representing variables and operations in a stack.

Methodology

Individuals are created by randomly selecting from the two available sets, of terminals and of functions, where terminals are variables and functions are operations (such as AND, OR, NOT). A population is generated by randomly creating individuals. Selection is done by comparing two individuals chosen by random from the population, and evaluating their fitness. The higher fitness individual is then put into the next population, and this selection is done until the next population is of the desired size (the same as the current population). Crossover of computer programs is possible due to the implementation of the programs as a stack. The stack contains variables (terminals) and operations (functions) in a stack. The stack executes each item, and the final result is the top item of the stack. This structure allows crossover to occur, since there is not a concern for whether or not the value at the end of the stack is valid, just whether or not we have a value in the stack at all. Thus, crossover is done the same as previously outlined, with a single point crossover, swapping the halves of two individuals in a population and producing the children that are put into the next population, with a given probability. Mutation follows the same logic as crossover in that it is not a concern that swapping values in the stack will break the syntax of the program. Thus, mutation is just a random swap of a stack value with a value from either the terminal or function set, with a given probability.

Results and Discussion

For parameters Probability of Crossover (Pc) and Probability of Mutation (Pm), 0.8 and 0.5 was chosen. The values are averaged over 25 runs.

Pc	Pm	Average Fitness
0.0	0.0	9.48
0.0	0.1	8.72
0.0	0.2	7.48
0.0	0.3	7.88

Pc	Pm	Average Fitness
0.0	0.4	7.48
0.0	0.5	7.08
0.0	0.6	6.64
0.0	0.7	7.2
0.0	0.8	7.04
0.0	0.9	7.64
0.1	0.0	10.4
0.1	0.1	12.2
0.1	0.2	12.08
0.1	0.3	12.36
0.1	0.4	11.92
0.1	0.5	12.64
0.1	0.6	12.08
0.1	0.7	12.08
0.1	0.8	12.32
0.1	0.9	12.2
0.2	0.0	9.12
0.2	0.1	12.88
0.2	0.2	12.6
0.2	0.3	12.76
0.2	0.4	12.68
0.2	0.5	12.44
0.2	0.6	12.4
0.2	0.7	12.56
0.2	0.8	12.72
0.2	0.9	12.68
0.3	0.0	8.96
0.3	0.1	12.44
0.3	0.2	12.76
0.3	0.3	12.68
0.3	0.4	12.72
0.3	0.5	12.52
0.3	0.6	12.6
0.3	0.7	12.64
0.3	0.8	12.88
0.3	0.9	12.72
0.4	0.0	10.84
0.4	0.1	12.84
0.4	0.2	12.96
0.4	0.3	12.64
0.4	0.4	12.92
0.4	0.5	12.84
0.4	0.6	12.64
0.4	0.7	12.32
0.4	0.8	12.96
0.4	0.9	12.72

Pc	Pm	Average Fitness
0.5	0.0	9.72
0.5	0.1	13.04
0.5	0.2	13.04
0.5	0.3	12.88
0.5	0.4	13.04
0.5	0.5	12.56
0.5	0.6	12.76
0.5	0.7	12.96
0.5	0.8	12.68
0.5	0.9	12.84
0.6	0.0	10.08
0.6	0.1	12.68
0.6	0.2	13.04
0.6	0.3	12.8
0.6	0.4	12.88
0.6	0.5	12.88
0.6	0.6	12.84
0.6	0.7	12.88
0.6	0.8	12.88
0.6	0.9	12.72
0.7	0.0	10.28
0.7	0.1	13.08
0.7	0.2	12.76
0.7	0.3	12.76
0.7	0.4	12.72
0.7	0.5	12.96
0.7	0.6	12.92
0.7	0.7	12.92
0.7	0.8	12.8
0.7	0.9	12.72
0.8	0.0	9.24
0.8	0.1	13.12
0.8	0.2	12.92
0.8	0.3	12.92
0.8	0.4	12.96
0.8	0.5	13.04
0.8	0.6	12.44
0.8	0.7	12.76
0.8	0.8	12.68
0.8	0.9	12.92
0.9	0.0	11.36
0.9	0.1	13.04
0.9	0.2	13.0
0.9	0.3	12.44
0.9	0.4	12.84
0.9	0.5	12.8

Pc	Pm	Average Fitness
0.9	0.6	12.88
0.9	0.7	12.88
0.9	0.8	13.0
0.9	0.9	12.84

Program	Fitness
[‘NOT’, ‘XOR’, ‘AND’, ‘X3’, ‘AND’]	13
[‘XOR’, ‘X2’, ‘X2’, ‘X2’, ‘XOR’]	13
[‘NOT’, ‘X3’, ‘AND’, ‘AND’, ‘AND’]	13
[‘X4’, ‘OR’, ‘NOT’, ‘X4’, ‘AND’]	13
[‘OR’, ‘X2’, ‘NOT’, ‘X4’, ‘AND’]	13
[‘X4’, ‘X4’, ‘XOR’, ‘X2’, ‘AND’]	13 **
[‘NOT’, ‘X2’, ‘X2’, ‘X2’, ‘XOR’]	13
[‘X3’, ‘NOT’, ‘NOT’, ‘X3’, ‘XOR’]	13
[‘X2’, ‘X4’, ‘OR’, ‘OR’, ‘X4’]	11
[‘NOT’, ‘NOT’, ‘AND’, ‘X4’, ‘AND’]	13
[‘X2’, ‘X3’, ‘X4’, ‘X4’, ‘XOR’]	13
[‘NOT’, ‘X2’, ‘OR’, ‘OR’, ‘NOT’]	14 **
[‘X1’, ‘NOT’, ‘XOR’, ‘AND’, ‘AND’]	13
[‘X2’, ‘X2’, ‘X2’, ‘AND’, ‘XOR’]	13
[‘X1’, ‘X4’, ‘X4’, ‘X4’, ‘XOR’]	13
[‘X2’, ‘X2’, ‘NOT’, ‘AND’, ‘AND’]	13
[‘X1’, ‘XOR’, ‘X2’, ‘OR’, ‘NOT’]	13
[‘AND’, ‘X2’, ‘X2’, ‘NOT’, ‘AND’]	13
[‘XOR’, ‘X4’, ‘X1’, ‘NOT’, ‘AND’]	13
[‘OR’, ‘X2’, ‘XOR’, ‘X4’, ‘AND’]	15 **
[‘X4’, ‘AND’, ‘X1’, ‘X1’, ‘XOR’]	13
[‘X4’, ‘X2’, ‘X1’, ‘XOR’, ‘AND’]	13
[‘X4’, ‘X1’, ‘X3’, ‘XOR’, ‘AND’]	13 **
[‘X4’, ‘NOT’, ‘OR’, ‘NOT’, ‘AND’]	13
[‘AND’, ‘OR’, ‘XOR’, ‘X4’, ‘AND’]	12

We highlight the interesting solutions with double asterisks (**), and note that although these programs don’t reproduce the table completely, they do a decent job at optimizing. We notice that our algorithm tends to favor having an element from the function set in the last slot of the program, which shows that our program is trending towards optimizing, since with the number of instructions, we are optimally looking for the setup “terminal, terminal, function, terminal, function.” It is also interesting to note that our algorithm was able to get an optimization in the right format, but the variables happen to yield a lower fitness compared to something with a seemingly erroneous structure, but high fitness. ([‘X4’, ‘X4’, ‘XOR’, ‘X2’, ‘AND’] compared to [‘NOT’, ‘X2’, ‘OR’, ‘OR’, ‘NOT’])

In the case that the program has variable sizes, we could aim to limit the size of the program during evolution via introducing a penalty for programs that are too long, causing a trend towards preferring shorter programs. Additionally, with the program sizes being variable, we would have to alter the crossover function to use the correct lengths when referencing the respective segments, but otherwise would not have to change the implementation since the stack syntax would still hold even if the output isn't always a valid value, meaning that we can still evaluate fitness and penalize accordingly.

Conclusion

Our algorithm was able to provide good results and optimize the program given.