Natural Computing
Assignment 1

Tim van Dijk - S4477073
Tristan de Boer - S1007313

February 2018

1

A The total fitness of the population is equal to $100 * 76$. Every time a member is selected, the best member therefore has a chance of $\frac{157}{100*76}$ of being selected. A member is picked 100 times, so the expected value is:

$$100 * \frac{157}{100 * 76} = \frac{157}{76} = 2,065789474 \tag{1}$$

B The chance of the best member not being picked is $1 - \frac{157}{100*76)}$. The chance of this happening 100 times in a row (i.e. the best member is not picked at all is):

$$\left(1 - \frac{157}{100 * 76}\right)^{100} = 0,124005992 \tag{2}$$

2

The probability for selecting the individuals $x = 3$, $x = 5$, $x = 7$, we first calculate the fitness of all individuals, being respectively $f(3) = 9$, $f(5) = 25$, $f(7) = 49$. The probability of calculating the probability of picking a individual is calculated by the fitness of the individual by the sum of fitnesses which, in this case, is 83. Then, we calculate the probability: $P(x = 3) = \frac{9}{81} = \frac{1}{9}$, $P(x = 5) = \frac{25}{81} = 0,\overline{308641975}$, and $P(x = 7) = \frac{49}{81} = 0,\overline{604938271}$.

If we use $f_1(x) = f(x) + 8$ as our fitness function and use the same methodology, we get the following: $f_1(3) = 17$, $f_1(5) = 33$, $f_1(7) = 57$. Sum: 107
$P(x = 3) = \frac{17}{107} = 0,158878505...$, $P(x = 5) = \frac{33}{107} = 0,308411215...$, and $P(x = 7) = \frac{57}{107} = 0,53271028....$

There is a lower selection pressure when we use f_1 as our fitness function. This is because of the flat $+8$ bonus each of the individuals gets to their fitness. This bonus has a larger relative effect on individuals with a lower x value than those with a large x value. For example, $f_1(3)/f(3) = 17/9 = 1.889$ whereas $f_1(7)/f(7) = 57/49 = 1.163$.

3

A The Python code:

```
from random import randint
import matplotlib.pyplot as plt

def fitness(individual):
        return sum(individual)

def new_individual(length):
        return [randint(0, 1) for _ in range(length)]

def monte_carlo(fitness, length, max_gen):
        individual = new_individual(length)
        best = individual
        history = [fitness(individual)]

        for _ in range(max_gen-1):
                individual = new_individual(length)
                if fitness(individual) > fitness(best):
                        best = individual
                history += [fitness(best)]

        print("Best␣individual:␣", best)
        print("Fitness:␣", fitness(best))
        plt.plot(range(0, max_gen), history)
        plt.show()

def main():
        monte_carlo(fitness, 100, 1500)

if __name__ == "__main__":
        main()
```

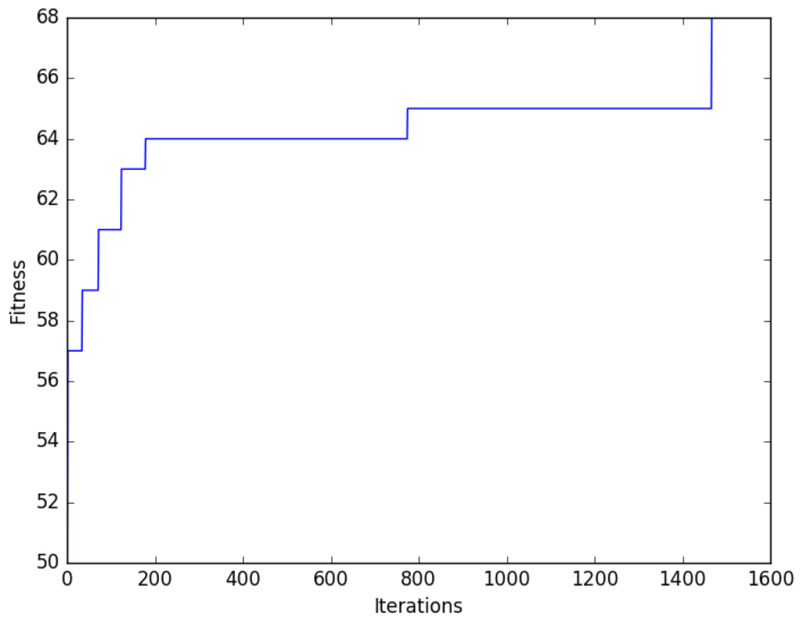

Figure 1: Exercise 3: Counting Ones

B 0 times. This is logical since the chance of finding the optimal solution is: $1-(1-\frac{1}{2^{100}})^{1500} \approx 1-1=0$

4

A The Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from random import randint
from random import random as rand

def new_individual(length):
    return [randint(0, 1) for _ in range(length)]

def fitness(individual):
    return np.sum(individual)

def flip(individual, p):
    return [int(not bit) if rand() < p else bit for bit in individual]

def genetic_algorithm(fitness, length, max_gen):
    p = 1 / length
    individual = new_individual(length)
    history = [fitness(individual)]

    for _ in range(max_gen - 1):
        x_m = flip(individual, p)
        if fitness(x_m) > fitness(individual):
            individual = x_m
        history += [fitness(individual)]
    print("Best individual: ", individual)
    print("Fitness: ", fitness(individual))
    plt.xlabel("Iterations")
    plt.ylabel("Fitness")
    plt.plot(range(0, max_gen), history)
    plt.show()

def main():
    genetic_algorithm(fitness, 100, 1500)

if __name__ == "__main__":
    main()
```

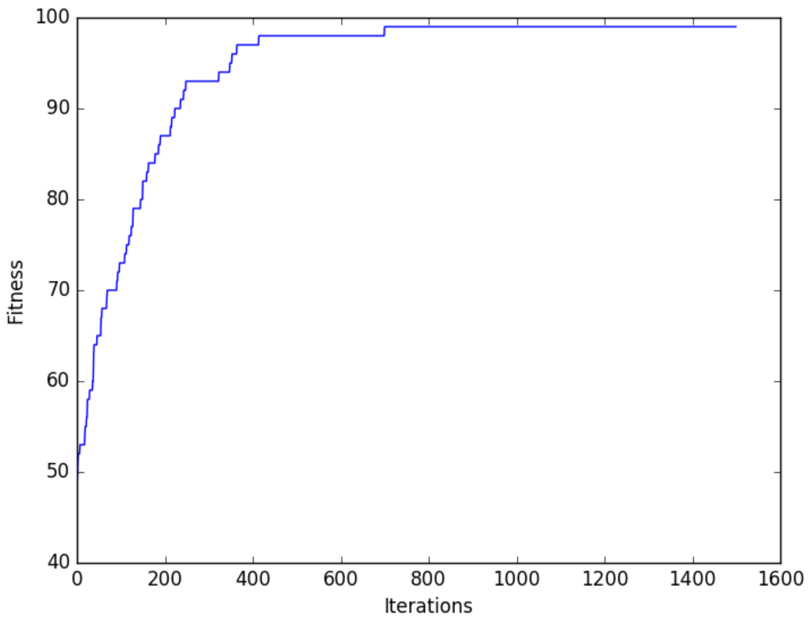

Figure 2: Exercise 4: (1 + 1)-GA

B The algorithm finds the optimal solution every time.

C There is a difference in performance compared to the Monte-Carlo algorithm, as we built upon the progress of the previous generations, whereas with Monte-Carlo, we discard the information previously found.

5

function set: $\{\vee, \wedge, \neg, \rightarrow, \leftrightarrow\}$
terminal set: $\{x, y, z, true, false\}$
s-expression $(\rightarrow\ (\wedge\ x\ true)\ (\vee\ (\vee\ x\ y)\ (\leftrightarrow\ z\ (\wedge\ x\ y)))$

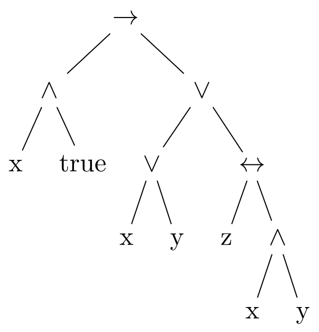

Figure 3: Tree representation of s-expression

6

For implementing the GP framework we've used DEAP (https://github.com/DEAP/deap). The source code can be found in Appendix A. Running the algorithm resulted in Figure 4. As we can see in this figure, we can observe an undesirable phenomenon. As the size of the algorithm increases, the fitness does not improve much, which is called bloat. There are several ways to combat bloat, as mentioned by Smith [2000]:

- Set a hard bound on the size of the program by setting a tree depth or a size limit to the number of nodes.
- Parsimony pressure: assign an explicit size penalty in the fitness measure.
- Restriction of subtree crossover.
- Use Changed Fitness Selection (CFS) or Improved Fitness Selection (IFS) where children are selected if their fitness has changed from that of their parents.

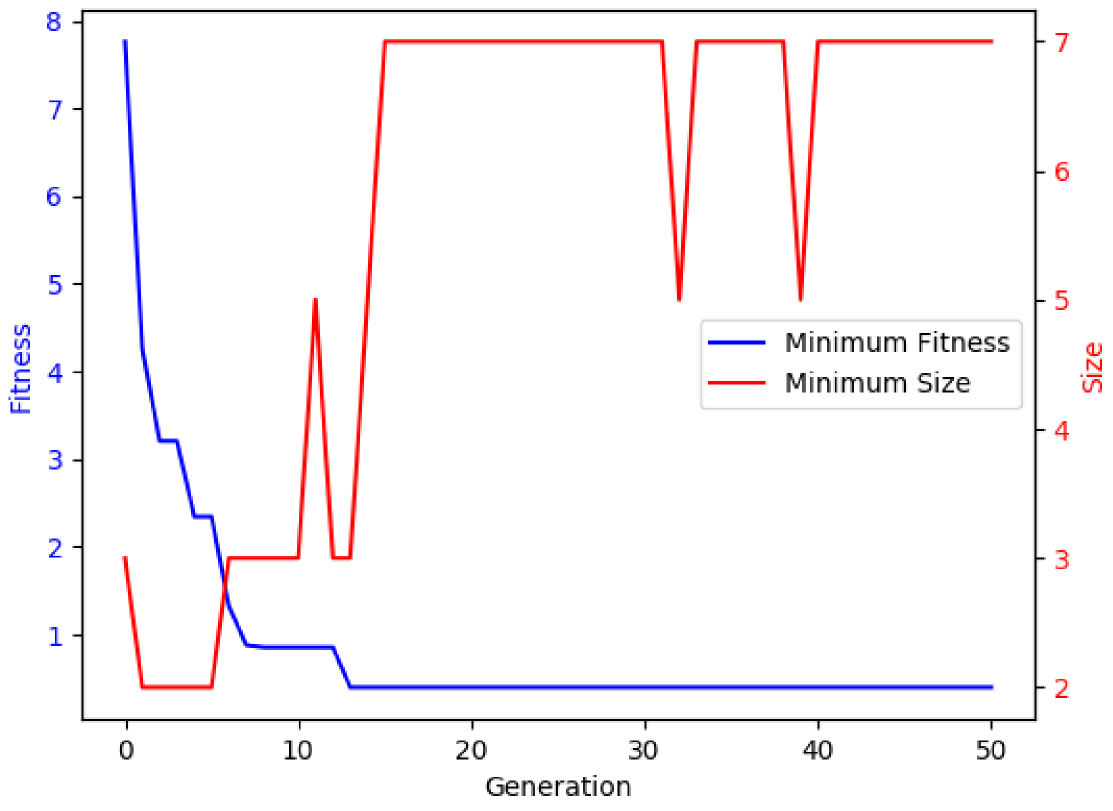

Figure 4: GP Run

References

P. W. H. Smith. Controlling code growth in genetic programming. In Robert John and Ralph Birkenhead, editors, *Advances in Soft Computing*, pages 166–171, De Montfort University, Leicester, UK, 2000. Physica-Verlag. ISBN 3-7908-1257-9. URL `http://www.soi.city.ac.uk/homes/peters/pub/Leicester.ps`.

A GP

```
import random
import operator
import csv
import itertools
import math

import matplotlib.pyplot as plt

import numpy

from deap import algorithms
from deap import base
from deap import creator
from deap import tools
from deap import gp


def evalfitness(individual, verbose=False):
        data = zip([x/10.0 for x in range(-10, 11, 1)], \
                [0.0000, -0.1629, -0.2624, -0.3129, -0.3264, \
                -0.3125, -0.2784, -0.2289,      -0.1664, -0.0909, \
                0.0, 0.1111, 0.2496, 0.4251, 0.6496, 0.9375, \
                1.3056, 1.7731, 2.3616, 3.0951, 4.000])
        if verbose:
                print('Input\tTarget\tActual\tAbsolute Difference')

        func = toolbox.compile(expr=individual)

        score = 0
        for inp, out in data:
                try:
                        if verbose:
                                print('{}\t{}\t{}\t{}'.format(inp, round(func(inp), 3

                        score += abs(func(inp) - out)
                except OverflowError:
                        score += 10**8
        return score,

def safeDiv(left, right):
        try:
                return left / right
        except ZeroDivisionError:
                return 0

def safeLog(x):
        try:
                return math.log(x)
        except ValueError:
                return 0


pset = gp.PrimitiveSet("MAIN", 1)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(operator.mul, 2)
pset.addPrimitive(safeDiv, 2)
pset.addPrimitive(operator.neg, 1)
pset.addPrimitive(math.cos, 1)
pset.addPrimitive(math.sin, 1)
pset.addPrimitive(safeLog, 1)
pset.addPrimitive(math.exp, 1)
pset.addEphemeralConstant('constant', lambda: random.uniform(-1, 1))
pset.renameArguments(ARG0="x")

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.FitnessMin, pset=pset)

toolbox = base.Toolbox()
```

```
toolbox.register("expr", gp.genFull, pset=pset, min_=2, max_=4)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("compile", gp.compile, pset=pset)
toolbox.register("evaluate", evalfitness)
toolbox.register("select", tools.selTournament, tournsize=7)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genGrow, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=pset)


def main():
        pop = toolbox.population(n=1000)
        hof = tools.HallOfFame(1)
        stats_fit = tools.Statistics(key=lambda ind: ind.fitness.values)
        stats_size = tools.Statistics(key=len)
        stats = tools.MultiStatistics(fitness=stats_fit, size=stats_size)
        stats.register("avg", numpy.mean)
        stats.register("std", numpy.std)
        stats.register("min", numpy.min)
        stats.register("max", numpy.max)
        stats.register("size", numpy.size)

        pop, logbook = algorithms.eaSimple(pop, toolbox, 0.7, 0.0, 50, stats, halloffa


        logbook.header = "gen", "evals", "fitness", "size"
        logbook.chapters["fitness"].header = "min", "avg", "max"
        logbook.chapters["size"].header = "min", "avg", "max"

        gen = logbook.select("gen")
        fit_mins = logbook.chapters["fitness"].select("min")
        fit_avgs = logbook.chapters["fitness"].select("avg")
        size_avgs = logbook.chapters["size"].select("avg")
        size_mins = logbook.chapters["size"].select("min")

        fig, ax1 = plt.subplots()
        line1 = ax1.plot(gen, fit_mins, "b-", label="Minimum Fitness")
        ax1.set_xlabel("Generation")
        ax1.set_ylabel("Fitness", color="b")
        for tl in ax1.get_yticklabels():
                tl.set_color("b")

        ax2 = ax1.twinx()
        line2 = ax2.plot(gen, size_mins, "r-", label="Minimum Size")
        ax2.set_ylabel("Size", color="r")
        for tl in ax2.get_yticklabels():
                tl.set_color("r")

        lns = line1 + line2
        labs = [l.get_label() for l in lns]
        ax1.legend(lns, labs, loc="center right")

        plt.show()

        return pop, stats, hof
```

```
if __name__ == "__main__":
        main()
```