Natural computing

Assignment 1

Stijn Voss, s4150511
Kevin Jacobs, s4134621,
Jaap Buurman, s0828122

February 26, 2016

1 Assignment 1

In a generational GA the complete population is replaced by offspring of the previous generation. Since there is no elitism involved there are no individuals which are selected by default. We assume that offspring is generated by selecting two parents(proportional to their fitness) and the population should be constants(100). Parents can be selected multiple times for different offspring. But not twice for the same offspring(it can not pair with itself). All parents are in the mate pool.

The chance of selecting the best member as the first parent is:

$$\frac{157}{7600}$$

the chance selecting it as the second parent is the change of not selecting it as the first parent and as the second parent(on average):

$$\frac{7600-157}{7600} * \frac{157}{98 * \frac{7443}{99} + 157}$$

note that $\frac{7443}{99} = \frac{7600-157}{99}$ the average of all other members than the best member.

(a) Thus the expected value is:

$$100 * \frac{157}{7600} + 100 * \frac{7600-157}{7600} * \frac{157}{98 * \frac{7443}{99} + 157} = 4.11$$

(b) Selecting it 100 times not as first parent and 100 times not as second parent:

$$(1 - \frac{157}{7600})^{100} * (1 - \frac{7600-157}{7600} * \frac{157}{98 * \frac{7443}{99} + 157})^{100} = 0.016$$

2 Assignment 2

Let $Pr(x)$ denote the probability of selection with function f(x) and $Pr1(x)$ denote the the probability of selection with fitness function $f1(x)$.

$f(3) = 9 \quad f(5) = 25 \quad f(7) = 49 \quad f(3) + f(5) + f(7) = 83$
$f1(3) = 17 \quad f1(5) = 33 \quad f1(7) = 57 \quad f1(3) + f1(5) + f1(7) = 107$

$Pr(3) = \frac{9}{83} = .108 \quad Pr(5) = \frac{25}{83} = .301 \quad Pr(7) = \frac{49}{83} = .590$
$Pr1(3) = \frac{17}{107} = .159 \quad Pr1(5) = \frac{33}{107} = .308 \quad Pr1(7) = \frac{57}{107} = .533$

$f1$ has a lower selection pressure than f since the constant added to $f1$ makes the differences between the different individual fitness values relatively smaller.

3 Assignment 3

See the appendix for the used code.

In figure 3, it can be seen that the Monte Carlo search has a hard time discovering better bitstrings. The chance that a bitstring of length n with all 1's is generated, is $\frac{1}{2}^n$. Therefore, the expected time of discovering a bitstring of length n is exponential in n. This can clearly be seen in figure 3.

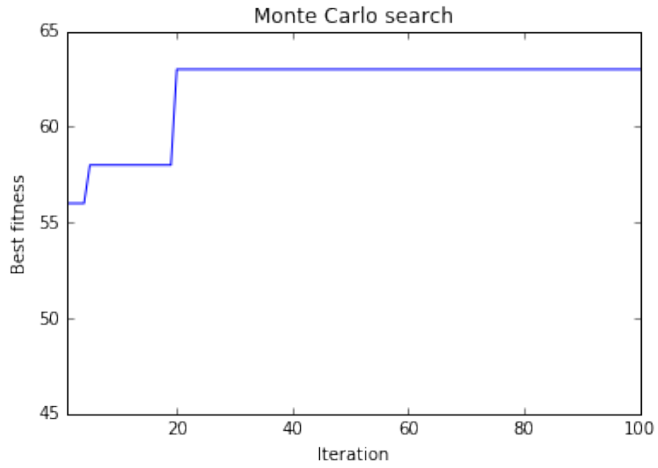

Figure 1: The best fitness for bitstrings of length 100 after a certain number of iterations are shown.

The exponential behaviour can be seen better if 500 iterations are used. These results are shown in figure 3.

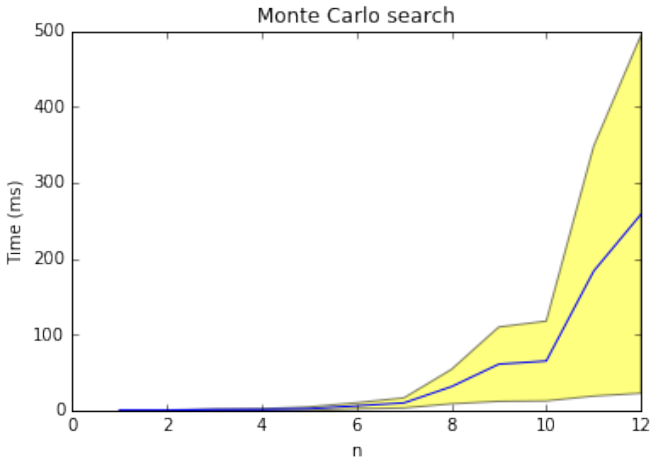

Figure 2: The mean time of finding the optimum for a given n is shown and also one standard deviation above and below the mean are shown in the figure. The numbers are based on 10 iterations for each n.

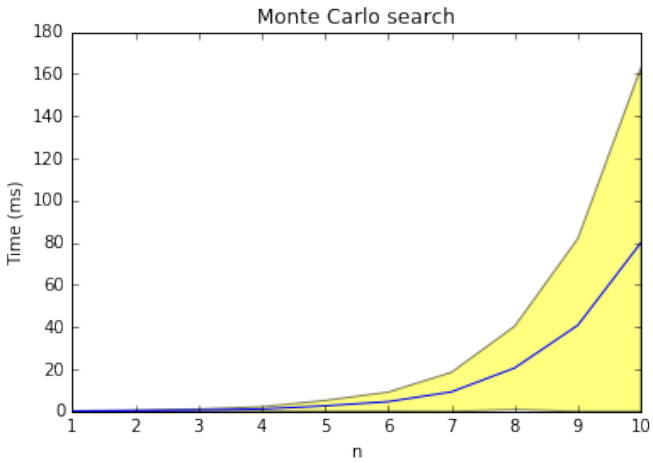

Figure 3: The mean time of finding the optimum for a given n is shown and also one standard deviation above and below the mean are shown in the figure. The smoother results are based on 500 iterations for each n.

4 Assignment 4

We are going to solve the counting ones problem through a genetic algorithm. First we will derive a simplifications for the problem. The fitness function that we have to maximize is given by the sum of the elements of a bitstring. Let n be the length of the bitstring and k the number of ones, then the fitness of a bitstring is given by:

$$\sum_{i=1}^{k} 1 + \sum_{i=1}^{n-k} 0 = k$$

We can also rewrite these bitstrings with ones and minus ones rather than ones and zeros. If we again define the fitness function as the sum of the elements we get:

$$\sum_{i=1}^{k} 1 + \sum_{i=1}^{n-k} -1 = 2k - n$$

The second term of this fitness function depends on n, which is the same for any individual solution in the population. The first term denotes that the value of this fitness function is proportional to the number of ones in the solution. This is also true for the first fitness function. And thus we can safely rewrite our bitstrings of ones and zeros to ones and minus ones and still use the same elementwise sum to compare the fitness of two different solutions. The reason why we want to write our binary strings as ones and minus ones is because this allows us to multiply elements by minus one to flip a bit, which makes things easier.

(a) For the first question we will have a look at a single run to see how the Fitness value of the best individual evolves as a function of the number of iterations. As we can see in figure 4, the genetic algorithm has a much easier time to find new individuals with higher fitness values compared to the Monte-Carlo algorithm, even if the previous individual is already quite good. This is because of the fact that we utilize our previous solution in generating a new individual rather than randomly generating a new one from scratch, as is done in the Monte-Carlo algorithm.

(b) As we can see in figure 5, in contrary to the Monte Carlo algorithm the average time required for convergence to the global optimum depends roughly linearly on the problem size. This is a huge improvement over the exponential relationship between problem size and time required for the Monte Carlo algorithm. For completeness, we have also rerun this experiment with 500 runs for each value of n. These results are shown in the figure 6 below.

(c) There is a huge difference in performance compared to the Monte-Carlo search algorithm. The Monte-Carlo algorithm starts with a new randomly generated solution each iteration. Since the global optimum requires all ones and each element has an equal probability of being a one and zero, there are 2^n unique solutions with equal probability, of which just one is the global optimum. This means that a

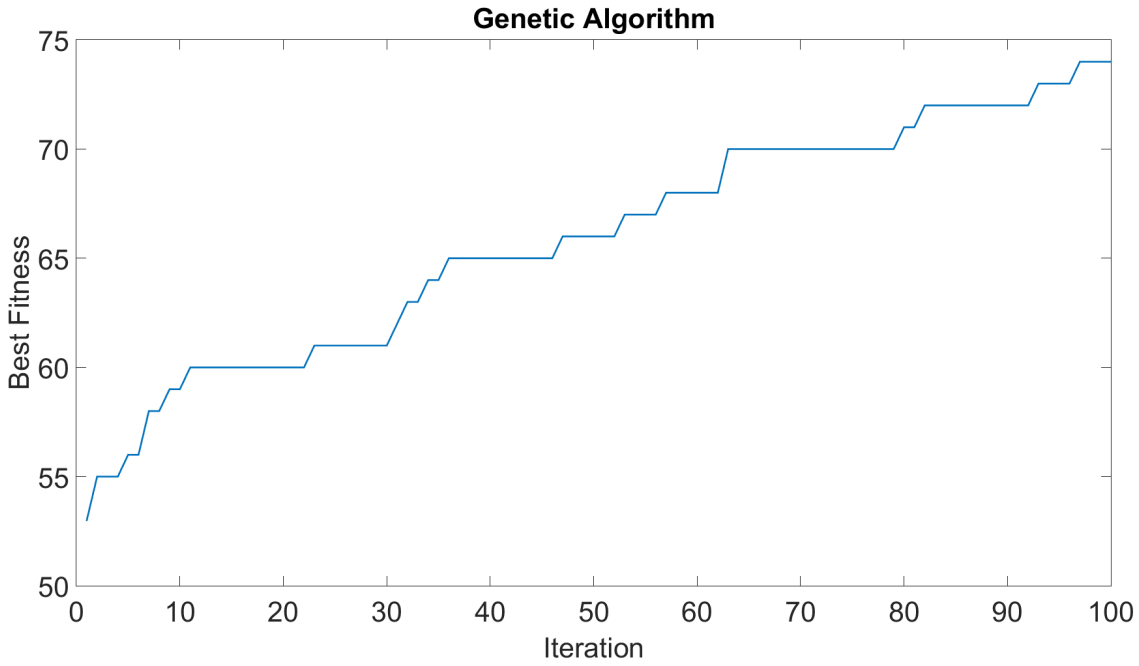

Figure 4: The best fitness for bitstrings of length 100 after a certain number of iterations are shown for the genetic algorithm

Monte-Carlo algorithm would take on average $\frac{2^n}{2}$ iterations to find the global optimum. This exponential dependence on problem size does not plague the genetic algorithm. The data here shows that the computation time needed scales roughly linearly with the problem size.

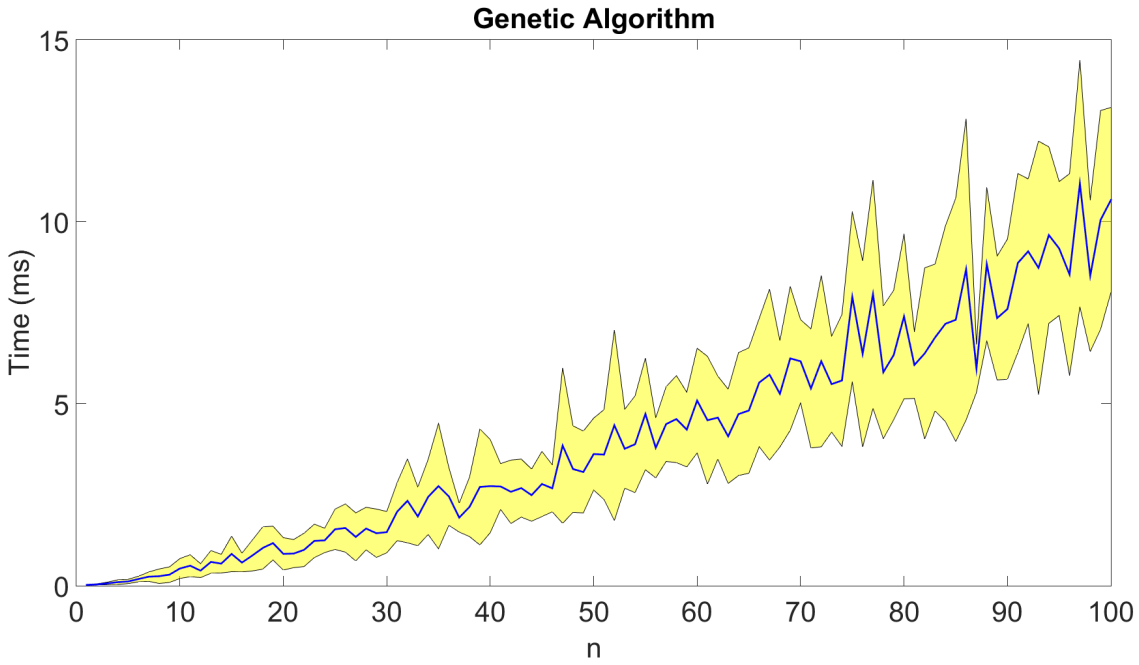

Figure 5: The mean time of finding the optimum for a given n is shown. One standard deviation above and below the curve is shown with the shaded color. There numbers are based on 10 runs for each given value of n.

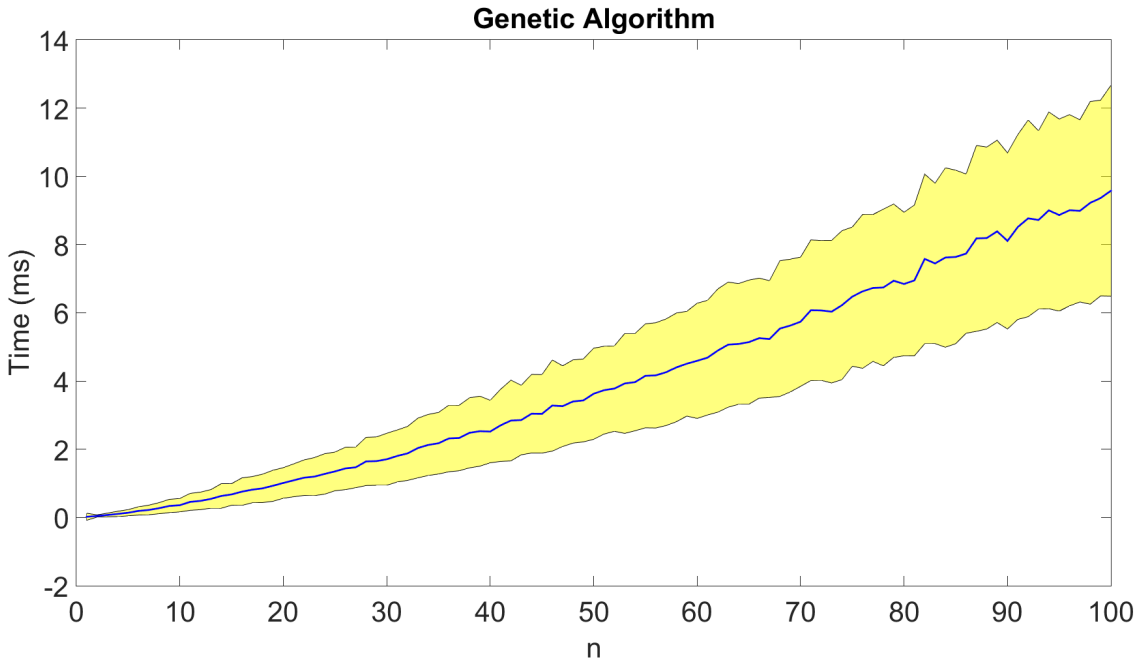

Figure 6: The mean time of finding the optimum for a given n is shown. One standard deviation above and below the curve is shown with the shaded color. There numbers are based on 500 runs for each given value of n.

5 Appendix

5.1 Exercise 3

The following code fragment shows the Monte Carlo search which was implemented in Python (using IPython Notebook).

```
import random
from scipy.stats import bernoulli
import math
import matplotlib.pyplot as plt
import time
import numpy as np
%matplotlib inline

def monte_carlo_search(objective_function, n, max_generations=None):
    """
    Do a Monte Carlo search.

    @param objective_function A function capable of evaluating the fitness of
    @param n The number of bits per bitstring.
    @param max_generations The number of rounds to do.
    """
    logbook = []
    best_score = -1
    generation = 1
    while True:
        bitstring = generate_bitstring(n)
        score = objective_function(bitstring)
        if score > best_score:
            best_score = score
        logbook.append((generation, best_score))
        generation += 1
        if max_generations is None:
            if best_score == n:
                break
        else:
            if generation > max_generations:
                break
    return logbook

def calculate_fitness_countones(bitstring):
    """
    The fitness score is the number of 1's in the bitstring.
    The higher the fitness score, the closer the bitstring is to the goal bits
```

```
    consisting of all 1's.

    @param bitstring A list consisting of bits (0 or 1).
    @return The number of 1's in the bitstring.
    """
    return sum(bitstring)

def generate_bitstring(n, p=0.5):
    """
    Generate a bitstring.

    @param n The number of bits to generate.
    @param p (Optional, default: 0.5) the probability of generating a 1.
    @return A list cosisting of bits (0 or 1).
    """
    return list(bernoulli.rvs(p, size=n))

# Perform Monte Carlo search
n = 100
iterations = 100
logbook = monte_carlo_search(calculate_fitness_countones, n, iterations)

# Plot the results
plt.plot([item[0] for item in logbook], [item[1] for item in logbook], '-b')
plt.xlabel('Iteration')
plt.ylabel('Best fitness')
plt.title('Monte Carlo search')
plt.ylim(45, 65)
plt.xlim(1, iterations)
plt.show()

# Now make the problem easier to detection the mean and variance for the optim
log = []
# Calculate for n=1..12
for n in range(1, 13):
    times = []
    for run in range(1, 11):
        start = time.time()
        # Perform the search
        monte_carlo_search(calculate_fitness_countones, n, None)
        end = time.time()
        # Add the calculate time to the log
        times.append((end - start) * 1000)
    # Convert the times to a matrix
```

```
        times = np.asmatrix(times)
        # Show the intermediate results
        print('n:', n)
        print('-'*70)
        print('Mean time to find optimum:', "\t"*2, times.mean(), 'ms')
        print('Standard deviation to find optimum:', "\t", times.std(), 'ms')
        print()
        # Append the results to the log
        log.append((n, times.mean(), times.std()))

# Make the plot
plt.plot([item[0] for item in log], [item[1] for item in log], '-b')
# Show the standard deviation
plt.fill_between([item[0] for item in log], [max(0, item[1] - item[2]) for ite
                 [item[1] + item[2] for item in log], facecolor='yellow', alph
plt.xlabel('n')
plt.ylabel('Time (ms)')
plt.title('Monte Carlo search')
plt.show()
```

5.2 Exercise 4a

The following code fragment shows the Genetic Algorithm search which was implemented in Matlab (using Matlab 2015a).

```
clear; close all; clc;

n = 100; % the length of the bitstrings
iter = 100; % the number of iterations

results = zeros(iter, n);

parent = sign(randn(n, 1)); % generate a random bitstring with ones and minus
for i = 1:iter
    % generate n random numbers between -1/n and 1-1/n. This gives a
    % probability of 1/n of each element being below zero.
    mutation = rand(n, 1) - 1/n;
    mutation = sign(mutation); % transform the values to plus/minus one
    child = parent .* mutation; % apply element-wise multiplication to randomi
    if sum(child) > sum(parent) % if the mutated child has a higher fitness va
        parent = child; % set the child as the new parent
    end
    results(i, :) = parent; % save the current best
end

results(results == -1) = 0; % transform the problem back into string with ones
results = sum(results, 2); % sum over the second dimension of bitstring length

linewidth = 2;
fontsize = 32;
plot(results, 'linewidth', linewidth)
xlabel('Iteration', 'fontsize', fontsize)
ylabel('Best Fitness', 'fontsize', fontsize)
set(gca, 'fontsize', fontsize)
title('Genetic Algorithm', 'fontsize', fontsize)
```

5.3 Exercise 4b

The following code fragment shows the convergence to the global optimum with a Genetic Algorithm search for different problem sizes n which was implemented in Matlab (using Matlab 2015a).

```
clear; close all; clc;

stringLength = (1:100)';
runs = 500; % the number of runs
time = zeros(length(stringLength), runs);

for s = 1:length(stringLength)
    n = stringLength(s);
    for r = 1:runs
        startTime = tic;
        parent = sign(randn(n, 1)); % generate a random bitstring with ones an
        while sum(parent) ~= n % while we haven't found the optimum
            % generate n random numbers between -1/n and 1-1/n. This gives a
            % probability of 1/n of each element being below zero.
            mutation = rand(n, 1) - 1/n;
            mutation = sign(mutation); % transform the values to plus/minus on
            child = parent .* mutation; % apply element-wise multiplication to
            if sum(child) > sum(parent) % if the mutated child has a higher fi
                parent = child; % set the child as the new parent
            end
        end
        time(s, r) = toc(startTime);
    end
end

average = mean(time, 2) * 1000; % calculate the average time in ms
standardDev = std(time, [], 2) * 1000; % calculate the standard deviation in n
figure
linewidth = 2;
fontsize = 32;
h = fill([stringLength;flipud(stringLength)], [average-standardDev;flipud(aver
set(h,'facealpha',.5)
hold all
plot(average, 'b', 'linewidth', linewidth)
title('Genetic Algorithm', 'fontsize', fontsize)
xlabel('n', 'fontsize', fontsize)
ylabel('Time (ms)', 'fontsize', fontsize)
set(gca, 'fontsize', fontsize)
```