University of Queensland

COSC7502 Major Project - Milestone 1

Genetic Algorithm

Name Tristan Burns

Student Number 42648493

November 3, 2021

1 Summary

Abstract

This report details the serial implementation, verification and performance optimisation of solution to the OneMax problem using a Genetic Algorithm implemented in C++.

This report details the serial implementation, verification and performance optimisation of solution to the OneMax problem using a Genetic Algorithm implemented in C++.

2 Serial Implementation

2.1 Algorithm

The genetic algorithm is a type of metaheuristic optimisation algorithm which has biologically inspired iterative crossover and mutation of candidate solutions.

Pseudo-code for the algorithm implemented in this project is presented in Algorithm 1.

Algorithm 1 The Genetic Algorithm

```
$n \leftarrow$ Chromosome Size
$m \leftarrow$ Population Size
$p \leftarrow \{\}$                                    ▷ Population of $m$ individuals with chromosomes of length $n$
$q \leftarrow \{\}$                                    ▷ Children of $m$ individuals with chromosomes of length $n$
$maxgenerations \leftarrow$ Max generations.
$best \leftarrow$ null
$generation \leftarrow 0$
$fitness \leftarrow \{\}$
for $m$ times do
    $p \leftarrow$ random individual                   ▷ Define a random seed to permit deterministic behaviour
end for
while $best \neq n$ and $generations < maxgenerations$ do
    for each $i$ in $p$ do
        $fitness[i] \leftarrow IndividualFitness(i)$
    end for
    $best = \max(fitness)$
    for $m/2$ times do
        Parents $p_1, p_2 \leftarrow select(p)$         ▷ Use tournament selection in this implementation
        Children $c_1, c_2 \leftarrow crossover(p)$     ▷ Use single point crossover in this implementation
        $q \leftarrow \{mutate(c_1), mutate(c_1)\}$     ▷ Use bitflip mutation in this implementation
    end for
    $p \leftarrow q$
    $generation++$
end while
return $best$
```

There are many possible variations on the genetic algorithm. Key features of this implementation are summarised as comments in Algorithm 1.

2.2 OneMax Problem

The OneMax problem is defined as the maximum number of value 1 bits in a bit string. Solution of the OneMax problem is effectively searching for a bitsting of exclusively ones.

For the Genetic Algorithm this is encoded in the chromosome, with possible genes of 0 or 1. In this implementation, a Genetic Algorithm is used to search through a population of m individuals with chromosomes of length n for a chromosome consisting of only ones. In this case a fitness function is defined to be the sum of all chromosomes, with optimal fitness being equal to the chromosome sum n.

2.3 Program Implementation

The genetic algorithm was implemented in C++ and compiled to C++11 standard using g++. Optimisation flags are discussed in Section 4.3 below.

The program is structured to use of macros for user defined constants and static arrays with memory allocated at compile time. In order to maximise performance, verbose functions were duplicates of the non-verbose counterparts, to avoid passing verbose arguments with subsequent internal branching.

3 Verification

The implementation was verified through explicit inspection of verbose output results and plotting the (normalised fitness over time) for a number of trials. An advantage of the OneMax problem is that the optimal solution and associated fitness is the same as the chromosome size n, permitting easy verification. Because of the stochastic nature of genetic algorithms a random seed was defined as an input parameter to enable deterministic and repeatable results.

Verbose output allowed for explicit verification of the key steps of the Genetic Algorithm, namely computing fitness, *crossover* and *mutation* verification. See Annex A for a single loop of verbose output.

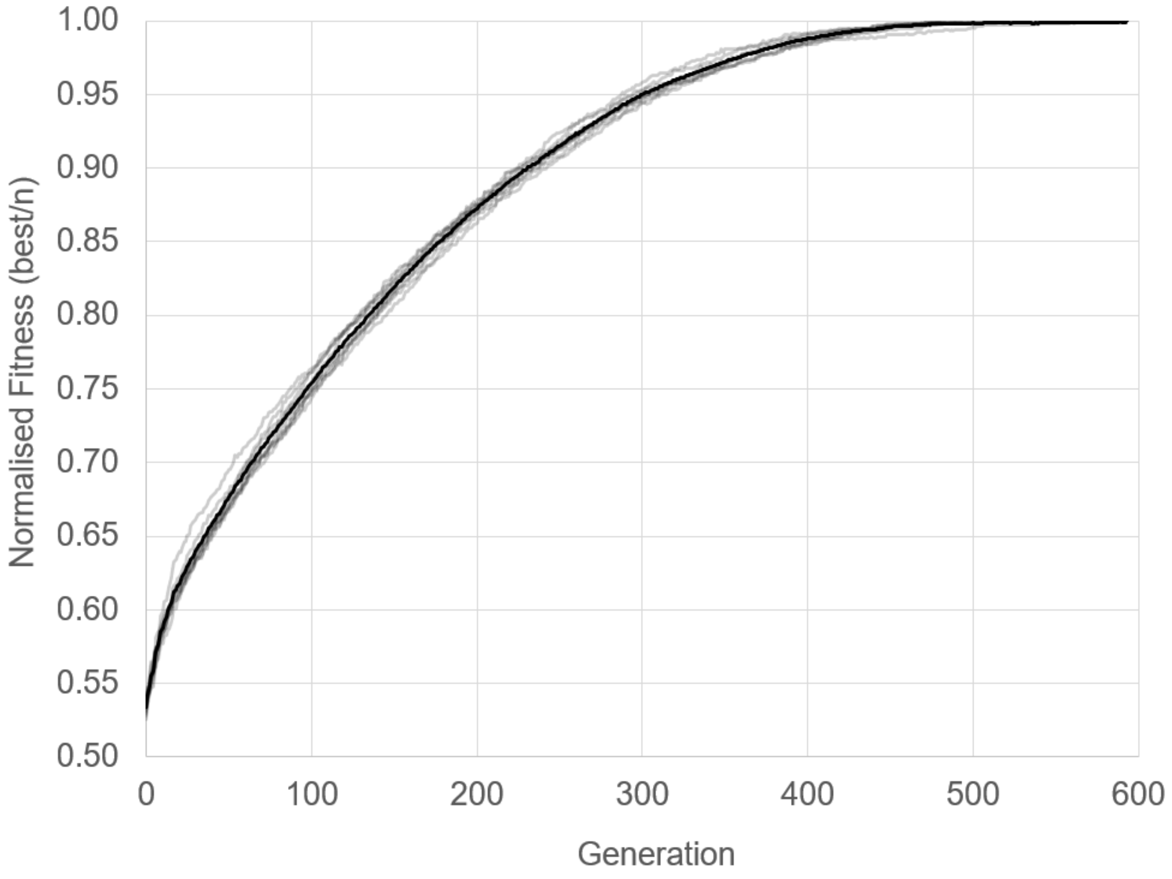

Figure 1: Normalised fitness vs number of generations, n, for a population size of $m = 100$ and a chromosome size of $n = 1024$. Different random seeds where used (light grey results), with the average plotted in black, demonstrating robust convergence for large problem size.

4 Performance Optimisation

4.1 Manual Optimisation

An attempt was made to manually optimise the code based on gprof profiling. The program was compiled with the *-pg -O0* flags set. Based on gprof profiling (see Annex B), the function *Mutate* (see below), was identifed as taking approximately 21 percent of the runtime.

```
void Mutate(int *q, int n, int m)
{
    int mu;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            mu = (std::rand() % n);
            if (mu == 0)
            {
                (q(i, j)) = 1 - (q(i, j));
            }
        }

    }
    return;
}
```

An attempt was made to remove the internal branching as follows.

```
void Mutate(int *q, int n, int m)
{
    int mu;
    for (int i = 0; i < m; i++)
    {

        for (int j = 0; j < n; j++)
        {
            mu = ((std::rand() % n)==0);
            q(i, j) = (q(i, j))*(1-mu) + (1 - (q(i, j)))*(mu); //remove branching
        }

    }
    return;
}
```

However, followup profiling revealed that this actually made performance worse with the function *Mutate* taking approximately 37 percent of the runtime. Therefore, this implementation was reverted and optimisation was focused on compile flags, as discussed in section 4.2

4.2 Optimisation Flags

Because of the number of nested loops in the *Crossover*, *Assess Individual fitness* and *Mutate*, it was inferred that -funroll-loops would yield improvements.

What was unexpected, was the flags -flto and -fuse-linker-plugin in combination yielding a 10-20 percent improvement on -O3 optimisation alone. Although I do not understand this improvement from the documentation, I hypothesise that these flags improve the optimization by exposing more code to the link-time optimizer.

4.3 Performance Results

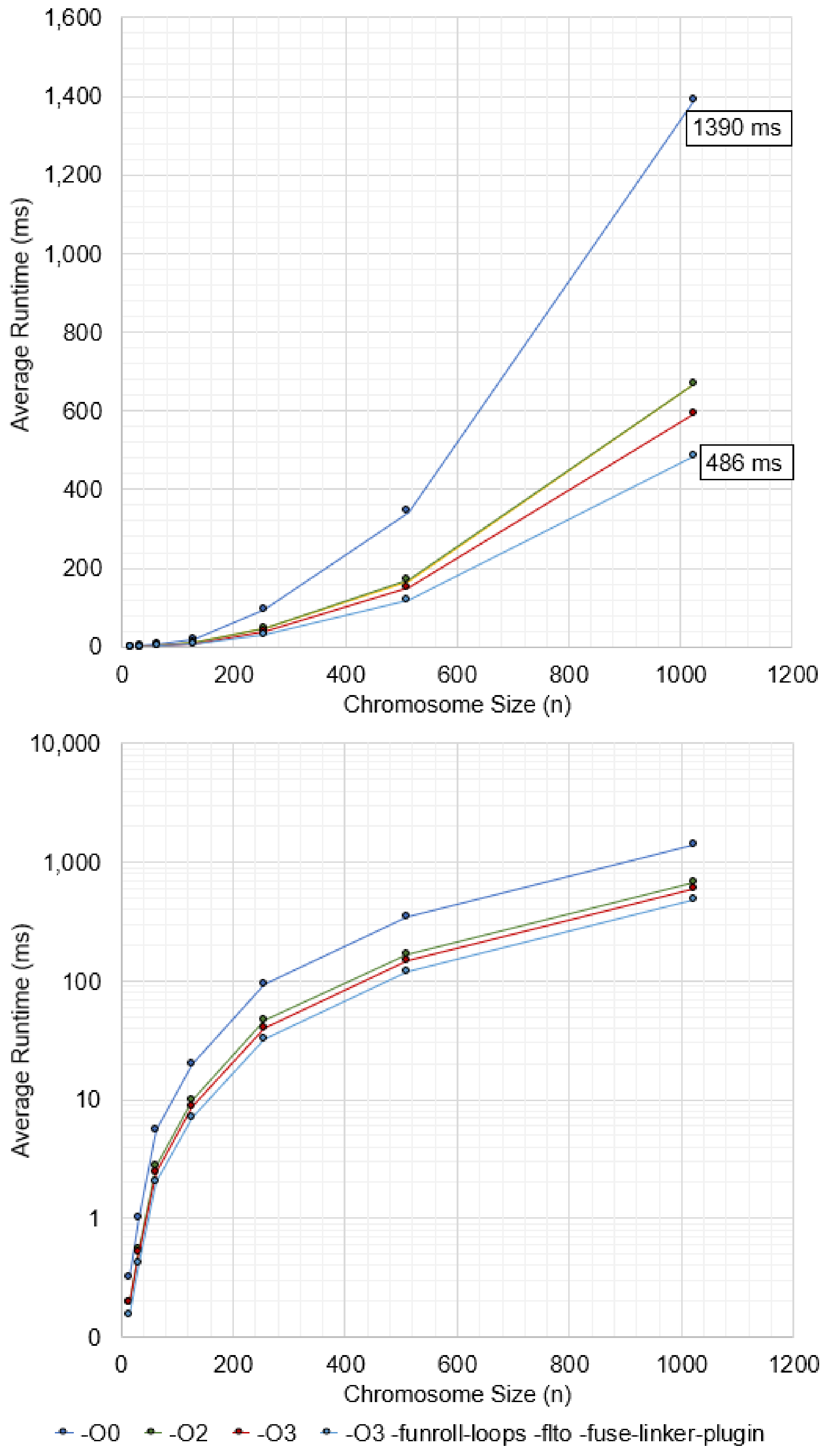

Figure 2: Average runtime (ten trials) vs chromosome size, n, for a population size of $m = 100$ individuals. Based on different optimisation flags a $\sim 2.8\times$ speedup is achieved. These results were computed using the same random seed, $seed = 42$.

next sec Random communication patterns are difficult to implement in MPI. MPI is based on all ranks having a deterministic set of communication patterns.

Annex A - Verbose Example

Using the following inputs, the verbose output for a single loop is displayed as an example.

```
#define n 16                    // Chromosome size (number of genes per individual).
#define m 8                     // Population size (number of individuals).
#define seed 42                 // Psuedorandom number generator seed (std:srand(seed)).
#define maxgenerations 2000     // Maximum number of generations (while loop limit).
#define t 2                     // Tournament size (parents competing for selection).
#define verbose true            // Verbose output (cout) for verification.
#define printfitness false      // Best fitness per generation output (cout).
```

Single loop (generation) verbose output (cout):

```
Running MaxFitness (Verbose)
0011001011011000 -> Individual 0, Fitness: 7
0001101111101011 -> Individual 1, Fitness: 10
0000011011101111 -> Individual 2, Fitness: 9
0101010111101110 -> Individual 3, Fitness: 10
1100010101010110 -> Individual 4, Fitness: 8
0110111001000101 -> Individual 5, Fitness: 8
0010010000011111 -> Individual 6, Fitness: 7
0101011101100100 -> Individual 7, Fitness: 8
 ------------ Generation: 0 Best Fitness: 10 --------------
Running Crossover (Verbose)
Crossover at chromosome 7:
0001101|111101011 -> Parent 1 (Individual 1)
0110111|001000101 -> Parent 2 (Individual 5)
0001101|001000101 -> Child 0
0110111|111101011 -> Child 1
Crossover at chromosome 7:
0110111|001000101 -> Parent 1 (Individual 5)
1100010|101010110 -> Parent 2 (Individual 4)
0110111|101010110 -> Child 2
1100010|101000101 -> Child 3
Crossover at chromosome 15:
000110111110101|1 -> Parent 1 (Individual 1)
010101011110111|0 -> Parent 2 (Individual 3)
000110111110101|0 -> Child 4
010101011110111|1 -> Child 5
Crossover at chromosome 0:
|0001101111101011 -> Parent 1 (Individual 1)
|0001101111101011 -> Parent 2 (Individual 1)
|0001101111101011 -> Child 6
|0001101111101011 -> Child 7
Running Mutate (Verbose)
 0   0   0   1   1   0   1  *1*  0   1   0   0   0  *0*  0   1  -> Child 0 Mutation at *_*
 0   1   1   0   1   1   1   1   1   1   1   0   1   0   1   1
 0   1   1   0   1   1   1   1  *1*  1   0   1   0   1   1   0  -> Child 2 Mutation at *_*
 1  *0*  0   0   0   1   0   0   0   1  *1*  0   0   1   0   1  -> Child 3 Mutation at *_*
 0   0   0   1   1   0  *0*  1   1   1   1   0   1   0   1   0  -> Child 4 Mutation at *_*
 0   1   0   1   0   1   0   1   1   1   1   0   1   1   1   1
 0   0   0   1   1   0   1   1   1   1   1   0   1   0   1   1
 0   0   0   1   1   0   1   1   1   1   1   0   1   0   1   1
Assigning next generation (Verbose)
0001101101000001 -> Child 0 to Individual 0, Fitness: 6
0110111111101011 -> Child 1 to Individual 1, Fitness: 12
0110111111010110 -> Child 2 to Individual 2, Fitness: 11
1000010001100101 -> Child 3 to Individual 3, Fitness: 6
0001100111101010 -> Child 4 to Individual 4, Fitness: 8
0101010111101111 -> Child 5 to Individual 5, Fitness: 11
0001101111101011 -> Child 6 to Individual 6, Fitness: 10
0001101111101011 -> Child 7 to Individual 7, Fitness: 10
```

Annex B - Profiling Results

Initial gprof results, prior to (attempted) manual optimisation of the function *Mutate*.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 38.19      0.45     0.45   144600     0.00     0.00  IndividualFitness(int, int*, int, int)
 21.21      0.70     0.25      482     0.52     0.52  Mutate(int*, int, int)
 17.82      0.91     0.21      482     0.44     0.44  NextGeneration(int*, int*, int, int)
 11.88      1.05     0.14      483     0.29     0.29  MaxFitness(int*, int, int)
 10.18      1.17     0.12      482     0.25     1.20  Crossover(int, int*, int*, int, int)
  0.85      1.18     0.01    48200     0.00     0.01  TournamentSelection(int, int*, int,
    int)
  0.00      1.18     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z14ParseArgumentsiPPc
  0.00      1.18     0.00        1     0.00     0.00  _GLOBAL__sub_I_useseed
  0.00      1.18     0.00        1     0.00     0.00  PrintFitness(int*, int)
  0.00      1.18     0.00        1     0.00     0.00  RandomPopulation(int*, int, int)
  0.00      1.18     0.00        1     0.00     0.00
    __static_initialization_and_destruction_0(int, int)
  0.00      1.18     0.00        1     0.00     0.00
    __static_initialization_and_destruction_0(int, int)
```

gprof results, after to (attempted) manual optimisation of the function *Mutate*. Subsequently, the initial implmentation of mutate was used.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 37.98      0.55     0.55      482     1.14     1.14  Mutate(int*, int, int)
 27.62      0.95     0.40   144600     0.00     0.00  IndividualFitness(int, int*, int, int)
 12.43      1.13     0.18      482     0.37     1.20  Crossover(int, int*, int*, int, int)
 10.36      1.28     0.15      483     0.31     0.31  MaxFitness(int*, int, int)
  8.98      1.41     0.13      482     0.27     0.27  NextGeneration(int*, int*, int, int)
  0.00      1.41     0.00    48200     0.00     0.01  TournamentSelection(int, int*, int,
    int)
  0.00      1.41     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z14ParseArgumentsiPPc
  0.00      1.41     0.00        1     0.00     0.00  _GLOBAL__sub_I_useseed
  0.00      1.41     0.00        1     0.00     0.00  PrintFitness(int*, int)
  0.00      1.41     0.00        1     0.00     0.00  RandomPopulation(int*, int, int)
  0.00      1.41     0.00        1     0.00     0.00
    __static_initialization_and_destruction_0(int, int)
  0.00      1.41     0.00        1     0.00     0.00
    __static_initialization_and_destruction_0(int, int)
```