Approximation of the Traveling Salesman Problem utilising a genetic algorithm in a parallel system

Michael Barth,
26206

March 15, 2009

Abstract

This document gives an introduction to genetic algorithms with a short detour to their history, describing how evolution works in nature and how genetic algorithms try to mimic this functionality. No mathematical background nor proof is given in this document. Further, it describes the development of a simple genetic algorithm in C++, providing excerpts of source code demonstrating the basic functionality. Afterwards, a more flexible approach based on the simple genetic algorithm is shown, including a description of its design and development.

Subsequently, a very basic introduction to the traveling salesman problem and the complexity class of np-complete is given. Follwing that, steps necessary to customise the genetic algorithm for the traveling salesman problem are described in detail, including the necessary problem encoding and a basic coverage on elitism. The developed algorithm is tested and the approximated results are evaluated.

To conclude this work the parallelisation using the OpenMP API [2] is described, as well as some problems typical to parallelisation. This document does not offer an introduction nor tutorial to the OpenMP API. Afterwards a performance comparison of the genetic algorithm running in sequential and parallel will be disclosed, drawing conclusions based upon the results.

This work was done within the scope of the *Hochschule für Technik und Wirtschaft Aalen*, Germany, as a mandatory project work over the course of the 6th semester.

Contents

Chapter 1

Introduction

1.1 Motivation

While sorting problems are solved to near optimality, search problems still remain challenging for todays computers. Simply speaking, a search algorithm takes a given problem as input and returns a solution to the problem as output.

There are several distinct approaches to solve a search problem in an efficient manner. While there are search algorithms that solve specific search problems with an optimal time and space efficiency, there is no general search algorithm that solves all search problems with optimal efficiency. On top of that, there exist quite a few problems which cannot be solved within a reasonable time frame at all. These problems are called NP-complete[1]. Problem-specific information is often needed to solve a search problem in an efficient manner, too. This is not desirable.

Genetic algorithms (GAs) are one approach to this problem. Belonging to evolutionary computation, they offer some advantages such as being independent of problem-specific information while maintaining good performance or an inherent resistance to local optimum traps.

1.2 Outline

Beginning with chapter 2, this work provides a basic introduction to genetic algorithms. Though it does neither provide the mathematical background nor proof for the basics covered – other literature is recommended for this purpose – it spotlights the special features genetic algorithms offer. Relying on the work done by Goldberg [1], a first simple genetic algorithm (SGA) was developed in C++ which will be showcased, followed by a description of the approach taken to enhance this first attempt with more flexibility. Last, a performance evaluation and comparison is done.

Chapter 3 focuses on the Traveling Salesman Problem. Starting with a short introduction to the problem, the chapter elaborates on which steps were taken to adjust the genetic algorithm to approximate an optimal tour. The algorithm and its performance are then evaluated.

[1]For an introduction to NP-complete see section 3.1.1

Going on, chapter 4 brings attention to one of the most notable features of a genetic algorithm, namely its intrinsic parallelism, and how this knowledge was used to optimise the performance of the algorithm utilising the OpenMP API [2].

Last but not least a conclusion is drawn, summarizing the insights of this work in chapter 5.

1.3 Goal of this work

The main goal of this work was to get a grip on several topics: First off, getting a basic understanding of parallel programming. As this work does not further first-hand threaded programming experience, using OpenMP provides an easy way to gain a basic understanding of parallel programming issues like racing conditions, mutual exclusion as well as synchronisation. Concepts independent of programming language or platform.

Second, I had the opportunity to gain some firsthand experience on genetic algorithms. The idea behind them, how they work, what the critical areas are performance-wise and last but not least how to utilise them to solve search problems.

Third, intensifying my C++ skills and applying some best practices [3]. Gaining experience at designing and modeling a software architecture which is flexible and modular, while still keeping an eye on performance.

1.4 Approach

Based on the pascal implementation of a simple genetic algorithm by Goldberg [1], a C++ port utilising object-orientation was developed. First tests were conducted using simple search problems.

Afterwards, the simple genetic algorithm was refactored into an enhanced genetic algorithm (EGA) to be more modular by design, allowing for extension so adaption to a broader array of search problems would be easy. The same simple search problems used in the previous iterations were used to compare the performance of the modular approach to the simpler one. Optimisations were conducted based on this. After this was done, appropriate extensions to approximate the TSP were written and tested as well. These tests included verification if an optimal tour was found and evaluation of the performance cost to do so.

The last step was to parallelise the EGA by using OpenMP pragmas and to evaluate, once again, its performance.

Chapter 2

Genetic algorithms

This chapter provides a basic coverage on genetic algorithms, describing their functionality in section 2.1. Following that, a simple genetic algorithm (SGA) was developed which will be elaborated in section 2.2. The SGA was further enhanced to offer more flexibility. This enhanced GA is called EGA and described in section 2.3. Concluding, the SGA and EGA are evaluated and compared, especially their performance, in section 2.4.

2.1 Fundamentals

Genetic algorithms are based on nature and the theory of evolution, therefore belonging to the field of *evolutionary computing* [8]. Evolutionary computing itself again relies on the Darwinian principles, such as natural selection or survival of the fittest.

The principles of genetic algorithms are quite simple yet powerful. Using a few basic principles – namely reproduction, a fitness function and selection – an encoded problem is solved over the course of several generations of a population composed of individual solutions to the problem. But before we go into detail, let's cover the historical background from which this approach emerged. Following that, the functionality will be elaborated first by theory, then by example.

2.1.1 Historical Background

At first, three *interpretations* of evolutionary computing emerged:

> *The use of Darwinian principles for automated problem solving originated in the fifties. It was not until the sixties that three distinct interpretations of this idea started to be developed in three different places.*
>
> *Evolutionary programming was introduced by Lawrence J. Fogel in the USA, while John Henry Holland called his method a genetic algorithm. In Germany Ingo Rechenberg and Hans-Paul Schwefel introduced evolution strategies. These areas developed separately for about 15 years. From the early nineties on they are unified as different representatives ("dialects") of one technology, called evolutionary computing.* (see [8])

This indicates that genetic algorithms are just one way of translating the principles of nature into artificial systems, known as evolutionary computing. Nevertheless we shall focus on this aspect of evolutionary computing.

Initially biologists and geneticists startet experimenting with the translation of natural processes into artificial systems in order to further understand the ways of nature, specifically to answer the question how evolution works. The first work was done in 1954 by Nils Aall Barricelli. These early approaches already contained the essentials of the emerging distinct interpretations mentioned before.

Genetic algorithms first became popular when John Henry Holland released his publication[1] *Adaptation in Natural and Artificial Systems* in 1975, introducing *Holland's schema theorem* [9] which is widely viewed to explain the power of genetic algorithms.

Later in 1989, Goldberg released a book on genetic algorithms with the title *Genetic algorithms in Search, Opimization & Machine Learning* [1] which can be viewed as a summary of the research done on genetic algorithms up to then, enhanced with Goldberg's own research.

According to Goldberg "the goals of [Holland's] research [has] been twofold: (1) to abstract and rigorously explain the adaptive processes of natural systems, and (2) to design artifical systems software that retains the important mechanisms of natural systems." [1]

2.1.2 Functionality

As mentionend in section 2.1.1, genetic algorithms are based on the important mechanisms of natural systems. But what exactly are the important mechanisms?

If one takes a look at the creatures nature created, one will notice nature produced very robust creatures in the course of evolution. Creatures hardly "crash" while processing something like computers do. Goldberg describes *robustness* as "the balance between efficiency and efficacy necessary for survival in many different environments." [1] Computer systems, especially operating systems but regular software as well, are also operated in a widely different landscape of hard- and software environments which often cause problems. Such programs could greatly benefit from improved robustness.

The variety of environments attaches importance to one specific feature: adaptability. Higher adaptability means creatures or systems can perform their duty longer and better, bringing more flexibility to the table. [1] Eventually, software could grow into something new by adapting to changes. Change is one ever-present constant, else there would be no need for new software in the same old branches over and over again. One ultimate system would be enough to satisfy the needs of the current and all coming generations. But since this is not the case, adaptability is key to the longevity and usefulness of software.

To understand how a genetic algorithm works we will take a look at its prototype: nature.

[1]Which originated from studies of cellular automata, conducted by Holland and his students at the University of Michigan.

How evolution works

The following example was taken from Jean-Philippe Rennard [4] and slightly modified for the purpose of clarity:

> *Imagine the prehistoric ancestor of whales, the basilosaraus. It was about 15 meters long and weighted 5 tons. It had a quasi-independent head and posterior paws. He moved using undulatory movements and hunted small prey.*

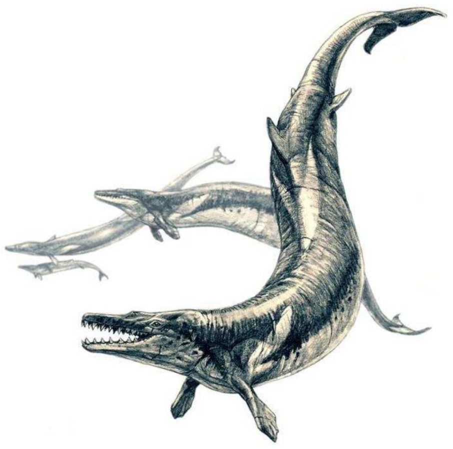

Figure 2.1: Reconstruction of a basilosaurus. Image by Pavel Riha (Wikipedia [7])

> *Movement in a viscous element like water is very hard and requires big effort. To hunt, the basilosarus must have had enough energy to move, control its trajectory and snatch its prey. The anterior members of basilosaurus were not really adapted to swimming. To adapt them, a double phenomenon must occur: the shortening of the "arm" with the locking of the elbow articulation and the extension of the fingers which will constitute the base structure of the flipper.*
>
> *Through time, subjects appeared with longer fingers and shorter arms. They could move faster and more precisely than before, and therefore, live longer and have many descendants.*
>
> *Meanwhile, other improvements occurred concerning the general aerodynamic like the integration of the head to the body, improvement of the profile, strengthening of the caudal fin...finally producing a subject perfectly adapted to the constraints of an aqueous environment.* (see [4])

This is the basic principle behind genetic algorithms: The higher the fitness of a basilosaurus, the longer it will live. Whereas higher adaptability helps to react to new circumentstances more rapidly, increasing its fitness. A basilosarus with higher fitness is more likely to attract a partner, which leads to descendants that will carry on his genes. Eventually the improved basilosaurus genes with short arms and long fingers will prevail. But adaption can only take place between two generations. Existing basilosaurus won't be able to grow short arms and long fingers within their lifetime, but their offspring may be born possesing those features.

Three basic operators

Every creature has its blueprint stored in the form of DNA in its body cells. DNA consists of strings which encode short groups of linked features[2]. Let's consider such a DNA string as a binary string, whereas the position of a binary digit indicates its meaning (e.g. paw or fingers) and the value expresses which kind of state it possesses (e.g. 0 = short, 1 = long). By combining those properties, we are able to express features like short paw or long finger.

In order to survive the basilosaurus must reproduce itself by mating with a partner. In this process an existing basilosaurus couple procreate new basilosaurus', their offspring. The offspring is composed by a mix of the features of their mother and their father or by completely copying the identical features of one parent[3]. In biology, the process of mixing the DNA strings of the parents is called *crossover.* Crossover allows for certain basilosaurus features – or groups of features – to be combined in a possibly new way which might be more useful in the present or future environment.

However, this does not allow the population of basilosaurus as a whole to develop new features since it's only a recombination of existing features. The development of new features is taken care of by mutation. In our binary string example, this would be achieved by randomly flipping a binary digit.

From this example, we can extract the three following basic operators:

- reproduction
- crossover
- mutation

These three operators constitute the essential functionality of any genetic algorithm.

A GA by example

So much for theory, let's do a small simulation "by hand". We want to find a maximum for the function $f(x) = x^2$ with $0 \leq x \leq 15$.

To get started we will need to encode x in an appropriate way. A binary string representing a binary number will suffice in this case, but there are other problems which necessitate other encodings (we will get back to this in chapter 3). The binary number starts with the most significant bit and has a maximum of four digits. Consider the following population, containing four individual chromosomes (short *individuals*) and their corresponding fitness rounded naturally:

```
A   1 1 0 0     fitness: 0.8        decoded x value: 12
B   1 0 1 0     fitness: 0.6        decoded x value: 10
C   0 0 1 1     fitness: 0.2        decoded x value: 3
D   0 0 0 0     fitness: 0.0        decoded x value: 0
================
Generation 1         sum: 1.6
```

[2]Goldberg calls them *building blocks* in [1].

[3]Though it is unlikely this will happen in nature.

The fitness function for this example is pretty simple. It's calculated by decoding the binary number into a decimal number k, which is used in the following formula:

$$fitness \quad = \quad \frac{k}{\max(x)} \quad = \quad \frac{k}{15}$$

Of course, such a function would be absurd since the dividend is the maximum of the function which we're looking for, rendering the search for it absurd. But after all, this demonstration should be as simple as possible.

Going on, to generate the next population the individuals of our current generation have to reproduce: Two individuals will be randomly chosen as mating partners – though the random selection is influenced by the fitness value: Individuals with higher fitness should possess a higher probability of being selected.

We select (admittedly not very randomly) individual A and B to mate. To create the offspring[4], we will need to determine a random crossover site, let's say we rolled xsite = 2.

A	1 1 x 0 0
B	1 0 x 1 0

Beginning with child A', all the bits of parent A are transfered to the first child until the crossover site is met. Then the remaining bits starting from the crossover site of parent B are transfered to the child. Likewise, but vice versa, we transfer all bits from parent B until the xsite to child B', then transfer all bits beginning at the xsite from parent A to B'.

Incidentally, after crossing a mutation happens in child B', randomly flipping the bit at position 4. The offspring A' and B' looks like this:

A'	1 1 1 0	
B'	1 0 0 1	without mutation: 1 0 0 0

The next children are produced using B and C as parents. After this, we have a population of four individuals, meaning no further reproduction is necessary.

```
A'   1 1 1 0     fitness: 0.9     decoded: 14
B'   1 0 0 1     fitness: 0.6     decoded: 9
C'   1 0 1 1     fitness: 0.7     decoded: 11
D'   0 0 1 0     fitness: 0.1     decoded: 2
=================
Generation 2       sum: 2.3
```

As can be seen, individual D completely dropped out since it was not selected for mating because of its low fitness value. Also, the best fitness increased from 0.8 to 0.9, as well as the sum of fitness which increased from 1.2 to 2.3.

Altogether, the new generation is superior to the old. There is an obvious improvement in the population as well as in single individuals. If we were to continue this example the maximum would be eventually found after a certain number of generations.

[4]We always create two children by default.

2.1.3 The Lingo

Since evolutionary computing is based on nature or, to be more specific, on biology it borrows some terms from genetics. To conclude the introduction to genetic algorithms we shall take a look at the lingo of genetics.

The strings storing features[5] are called *chromosomes* like their biological counterparts. Each chromosome is composed of *genes*, whereas every gene contains two properties: Its value called *allele* and its position called *locus*. Remember, the position – or locus – encodes which feature the gene stands for (in other words its meaning) while the allele itself expresses how the feature might look like.

A single chromosome could be described as a particular proposal for solution to a given problem. Each chromosome implicitly posseses a *fitness value*, indicating how good the solution is. Goldberg further adopts the terms genotype and phenotype from biology. A genotype is a complete set of chromosomes describing the "construction and operation of some organism. In artifical genetic systems the total package of strings is called a *structure* [...]". The phenotype represents "the organism formed by the interaction of the [genotype] with its environment [...]. In artifical genetic systems, the structures decode to form a particular *parameter set*, *solution alternative*, or *point* (in the solution space)". [1] In other words, he discriminates between chromosomes (strings or blocks of features), genotypes (solution sets using schemata to describe possible alleles) and phenotypes (actual solution sets using hard-coded alleles).

In this work, I did not find any use in discriminating between those three terms. When I use the term chromosome I mean the actual solutions using hard-coded alleles.

In genetics, a *genome* is a set of chromosomes (simply speaking). Therefore I chose this term to represent my set of chromosomes. A *population*, in addition, contains a genome. One could also label the genome as the genetic pool of a population.

2.2 A simple genetic algorithm

The implementation of a simple genetic algorithm shown in figure 2.2 is based on the pascal implementation of Goldberg [1]. It was adapted to object-orientation and converted to C++.

As can be seen[6], the three main operators – reproduction, crossover and mutation – reside within the `GeneticAlgorithm`-interface. Although reproduction is not present at first sight, one can guess it's hidden somehow within `raiseGeneration()`.

Basically, `raiseGeneration()` "wraps" `generation()`, which implements the functionality of reproduction, by calling it and doing some statistics before and after the call. See listing 2.1 for the actual implementation in code.

[5]Or groups of features called *building blocks* by Goldberg [1].

[6]More or less obviously.

Listing 2.1: Implementation of raiseGeneration()

```cpp
void SimpleGeneticAlgorithm::raiseGeneration()
{
    new_pop->incGeneration();    // increment generation
                                 // counter by 1
    generation();                // generate new generation
                                 // by reproduction
    // Statistics
    stats.calcStatistics(*new_pop);
    stats.report( new_pop->getGeneration() );

    *old_pop = *new_pop;         // copy content of of new_pop
                                 // and store in old_pop
}
```

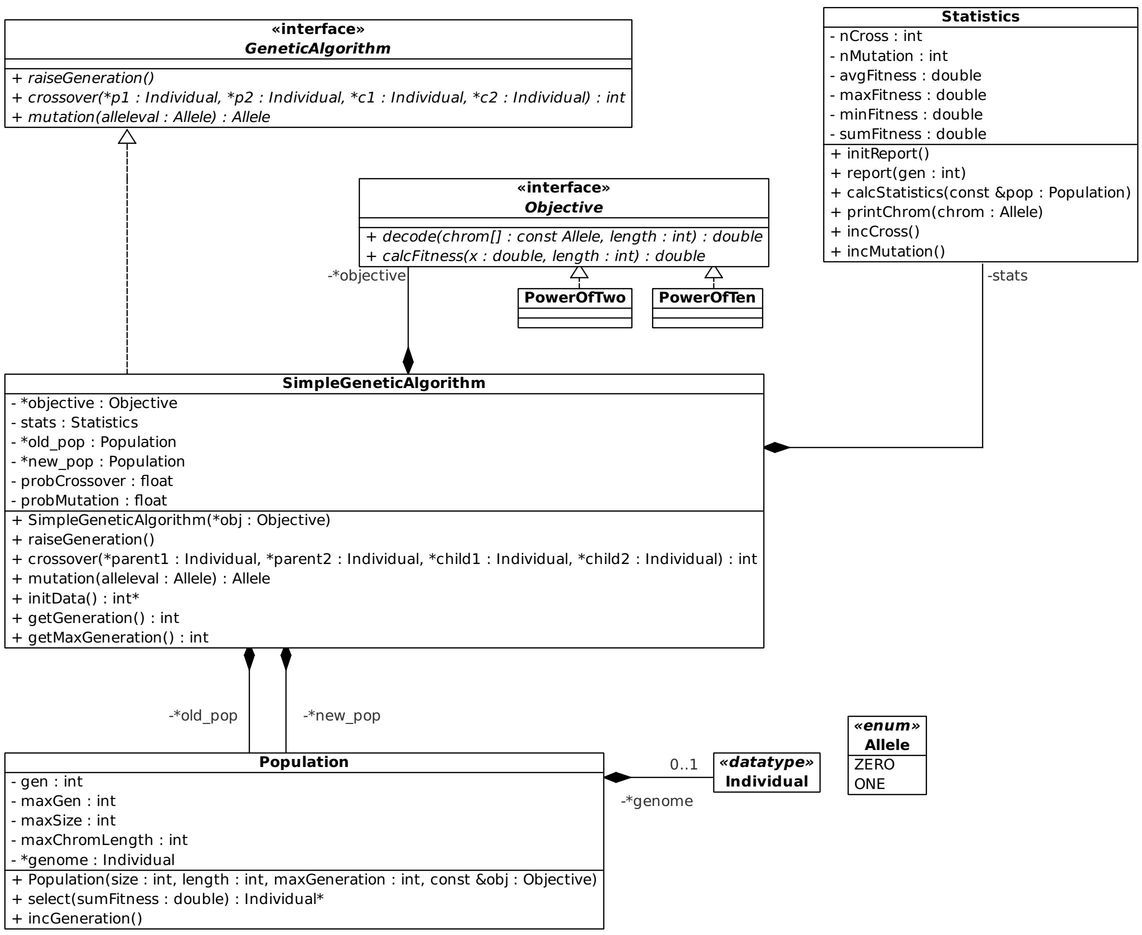

Figure 2.2: UML class diagram of the adapted simple genetic algorithm

Selection is done using the principle of a weighted roulette wheel[7]: Think of a roulette wheel containing slots. Each individual gets a share of connected slots which is in relation to its fitness: Higher fitness means a bigger share of slots for the individual, lower fitness means a smaller share of slots.

[7]Also known as *fitness proportionate selection*.

In order to select an individual we spin the wheel and see which slot wins. From the winner slot we can determine the corresponding individual.

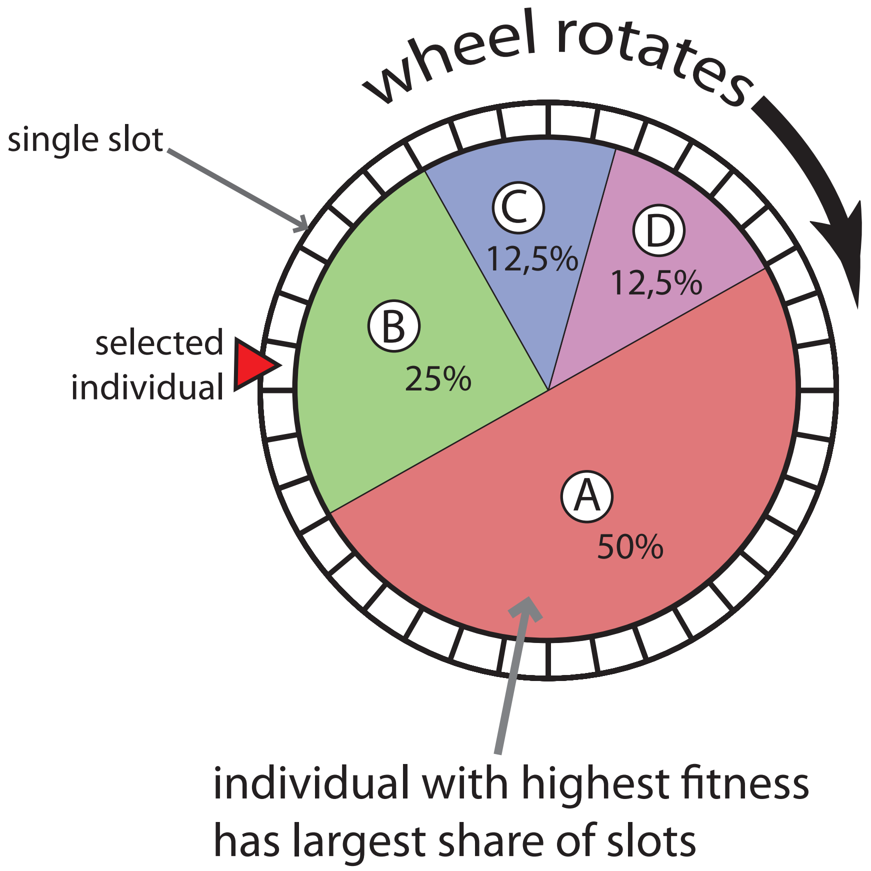

Figure 2.3: Illustration of the roulette wheel selection

Since there is a lot of randomness involved in a genetic algorithm it became apparent that a robust and authentic random number generator is key to finding optimal solutions. Looking into a recommendation of Professor Dr. Christoph Karg, I finally implemented "ran3" from *Numerical recipes in C*[8] [5].

As only one instance of the RNG is necessary the singleton pattern [6] was used to ensure only one instance will be present at any time. When the genetic algorithm was parallelised, the singleton pattern was adjusted in that every thread received its own unique instance of the RNG. Figure 2.4 illustrates the RNG class. Both, the SGA and EGA, use this implementation, so they share it's code base.

A genetic algorithm, implemented by `SimpleGeneticAlgorithm` in this case, is composed by two `Population`-objects: `old_pop` and `new_pop`. As a matter of principle, a genetic algorithm only requires one population to work with. But working with two makes it easier, because while generating the new generation the GA still requires access to the individuals of the old generation to chose parents from. At this point a temporary copy of the current generation is necessary so newly generated children cannot be selected as parent for their own generation.

[8]See page 283, second edition.

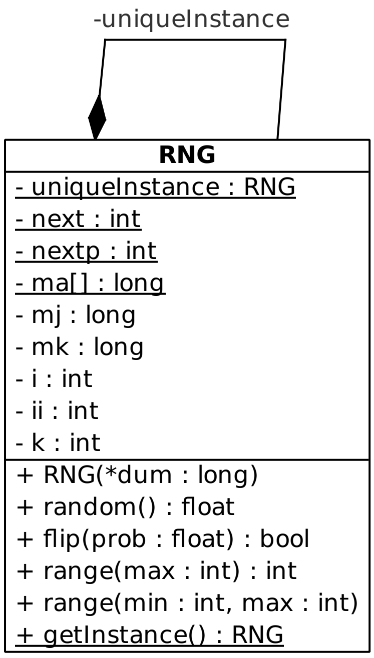

Figure 2.4: UML class diagram of the random number generator "ran3"

For simplicities sake, I chose to compose my GA of two Populations: The old one, representing the current generation, and a new one, to store the chromosomes of the next generation while it's being generated – overriding *only the chromosomes* of its previous population.

After reproduction is done, the two generations are swapped, so the latest generation always resides within `old_pop`.

An individual is represented by the structure shown in listing 2.2:

Listing 2.2: Member variables of an individual structure

```
// Use enum so we don't depend on specific values
enum Allele {
    ZERO,
    ONE
};

struct Individual {
    int length;              // length of chromosome array
    double x;                // decoded value
    double fitness;
    int xsite;
    Allele *chromosome;      // number in binary format
                             // stored in Allele array
    // ...
```

Lastly, the `Statistics`-class is used to keep track of the number of crossover and mutation operations done, as well as the best, worst and average fitness of the population. It also provides some report functions to print the statistics to the console while the GA is running.

2.2.1 Pros and cons

The major advantage of the simple approach is its speed. As we will see later at the performance evaluation, the simple non-extensible approach is by a good deal faster than the more modular approach.

A huge drawback, on the other hand, is its flexibility. Although adding new objectives was made easy by programming against an interface, it was forgotten to do the same for individuals. Inheriting from `individual` would not be a good idea, since it's not supposed to be inherited of and therefore its methods are non-virtual[9]. Apparently, adding a new encoding to this kind of design would not be as easy as it could be.

But let's say we somehow added a new type of individual to store the new encoding. It would be only then that we discover another problem with the design: Placing the crossover and mutation operator in the genetic algorithm itself may seem smart at first, but sacrifices easy extensibility. Being done adding the new encoding, how should the GA be able to handle it? It certainly does know how to cross and mutate binary encoded individuals, but this is done in a completely different way for permuation encoded individuals.

The `select()`-method didn't seem to belong to the `Population`-class as well. Also, it wasn't extensible itself, but there are by far more selection schemes out there than just roulette wheel.

As this example demonstrates, easy extensibility certainly is not one of the features at which the SGA will shine.

2.3 Modularising the genetic algorithm

The next step was to change the first approach (see section 2.2) and turn it into something more extensible, to enhance it. Extensive refactoring was necessary to achieve this. Most of the limitations of the first approach became apparent only after its implementation process began, thus many ideas that were incorporated into the enhanced version (called EGA, meaning enhanced genetic algorithm) emerged while and after the implemention of the SGA.

A note on pointers

All pointers used in the enhanced genetic algorithm UML diagrams actually are shared pointers of the type `tr1::shared_ptr<typename T>`. It was chosen to depict them as simple pointers in the UML diagrams since this did not interfere with communicating the concepts while helping to avoid bloating the diagrams with unneccessary long type names.

2.3.1 Thoughts on performance

Performance was another consideration during the refactoring process. Since a genetic algorithm spends most of its time executing the reproduction operator, special care had to be applied while refactoring it.

[9]See Scott Meyers' *Effective C++* [3] – Item 36: Never redefine an inherited non-virtual function.

Instancing a new population for every generation is a waste of computational space and time – especially of the latter. Think about all the constructors which are needed to be called when instancing a new population from scratch. If it has a size of n, then as many `individual`-constructors would be called.

In the enhanced approach existing individuals are reused by changing only their referenced data: Their chromosomes will be overridden by the `crossover`-operator, their fitness will be recalculated, and both, the cross site and fitness, will be stored in a member variable.

Listing 2.3: Refactored implementation of raiseGeneration()

```
void EnhancedGeneticAlgorithm::raiseGeneration()
{
    oldPopulation->reproduce(
            crossoverProbability,
            mutationProbability,
            newPopulation,
            statistics);

    swap(oldPopulation, newPopulation);

    oldPopulation->incGeneration(); // increment generation
    newPopulation->incGeneration(); // counter by 1 for both

    // Statistics
    statistics.compute(*oldPopulation);
    statistics.report( oldPopulation->getGeneration() );
}
```

Additionally, safe downcasting from the interface type `Individual` to the actual derived type was used in the corresponding `cross`-method at the beginning (e.g., downcasting an object of type `Individual` to type `BinaryEncoded` in order to get access to some getters and setters only objects of class `BinaryEncoded` offer). Since there are some performance implications using `dynamic_cast<T>(`*expression*`)` according to *Item 27* of Scott Meyers' Effective C++ [3], this was later on changed.

Derived type specific methods were added to the interface to get rid of the need for safe downcasting. An empty implementation was written for those methods which did not make any sense in a specific derived type. This is not the best design, but as most of the computational time is spend in the `cross`-method this was a necessary step (*Pareto Principle*: 80% of the computational time is spend executing 20% of the code [12] – the `cross`-method is the biggest part of those 20% in this program).

So, as can be seen in figure 2.4, so-called *stub methods* were added to the `Individual`-interface.

Listing 2.4: Method stubs in the interface of Individual

```cpp
class Individual
{
public:
    // other method declarations...

    // Generally usable getters and setters
    virtual void setCrossSite(int xsite) = 0;
    virtual double getFitness() const = 0;
    virtual void setFitness(double fitness) = 0;
    virtual int getLength() const = 0;

// The following methods are stubs used to avoid the
// performance cost of dynamic_cast<T>().
                                    // needed by:
    virtual bool getGene(int pos)   // BinaryEncoded
        const = 0;
    virtual void setGene(int pos,   // BinaryEncoded
        bool allele) = 0;
    virtual int getCityId(int pos)  // PermutationEncoded
        const = 0;
    virtual void setCityId(int pos, // PermutationEncoded
        int cityId) = 0;
    virtual double getDistance()    // Statistics
        const = 0;
    virtual void setDistance(       // FitnessFunction
        double distance) = 0;
};
```

2.3.2 Decoupling encodings and selection schemes

As pointed out by the example in section 2.2.1 it was rather hard to add new encodings to the SGA. The first issue which should be solved by the EGA was to allow for easy extensibility. To achieve this for encodings two steps were taken.

The first step was to code against an interface, named `Individual`, instead of using an encoding implementation forthright. This way polymorphism and object composition could do their magic.

The second step was to relocate the `crossover` and `mutation` methods into the individuals themself. So each individual knew how to cross itself with another partner of its own kind. That way we encapsulated the parts that are expected to change from the genetic algorithm and combined it in a single place: The individual, or to be more specific, classes implementing the individual interface.

Figure 2.5 already shows the individual interface declaring methods with the suffix “Parallel”, like `crossoverParallel(...)`. Those functions are further explained at a later time (see chapter 4). For now they can be ignored in good conscience.

To further the decoupling of the individuals from the population a factory was utilised. The `Population`-object just tells the factory what kind of individual it wants (using an enum) and the factory cares for the details.

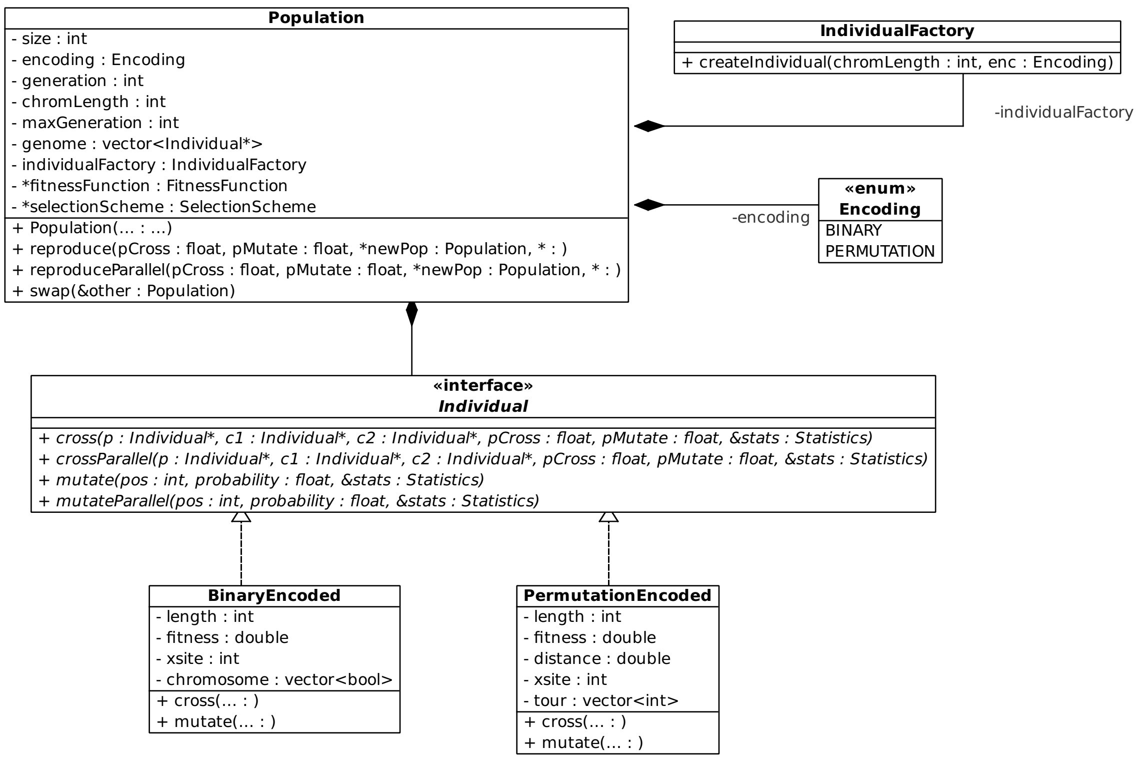
Population
- size : int
- encoding : Encoding
- generation : int
- chromLength : int
- maxGeneration : int
- genome : vector<Individual*>
- individualFactory : IndividualFactory
- *fitnessFunction : FitnessFunction
- *selectionScheme : SelectionScheme
+ Population(... : ...)
+ reproduce(pCross : float, pMutate : float, *newPop : Population, * :)
+ reproduceParallel(pCross : float, pMutate : float, *newPop : Population, * :)
+ swap(&other : Population)
IndividualFactory
+ createIndividual(chromLength : int, enc : Encoding)
-individualFactory
-encoding
«enum»
Encoding
BINARY
PERMUTATION
«interface»
Individual
+ cross(p : Individual*, c1 : Individual*, c2 : Individual*, pCross : float, pMutate : float, &stats : Statistics)
+ crossParallel(p : Individual*, c1 : Individual*, c2 : Individual*, pCross : float, pMutate : float, &stats : Statistics)
+ mutate(pos : int, probability : float, &stats : Statistics)
+ mutateParallel(pos : int, probability : float, &stats : Statistics)
BinaryEncoded
- length : int
- fitness : double
- xsite : int
- chromosome : vector<bool>
+ cross(... :)
+ mutate(... :)
PermutationEncoded
- length : int
- fitness : double
- distance : double
- xsite : int
- tour : vector<int>
+ cross(... :)
+ mutate(... :)

Figure 2.5: Population using an individual interface to allow for easy extensibility

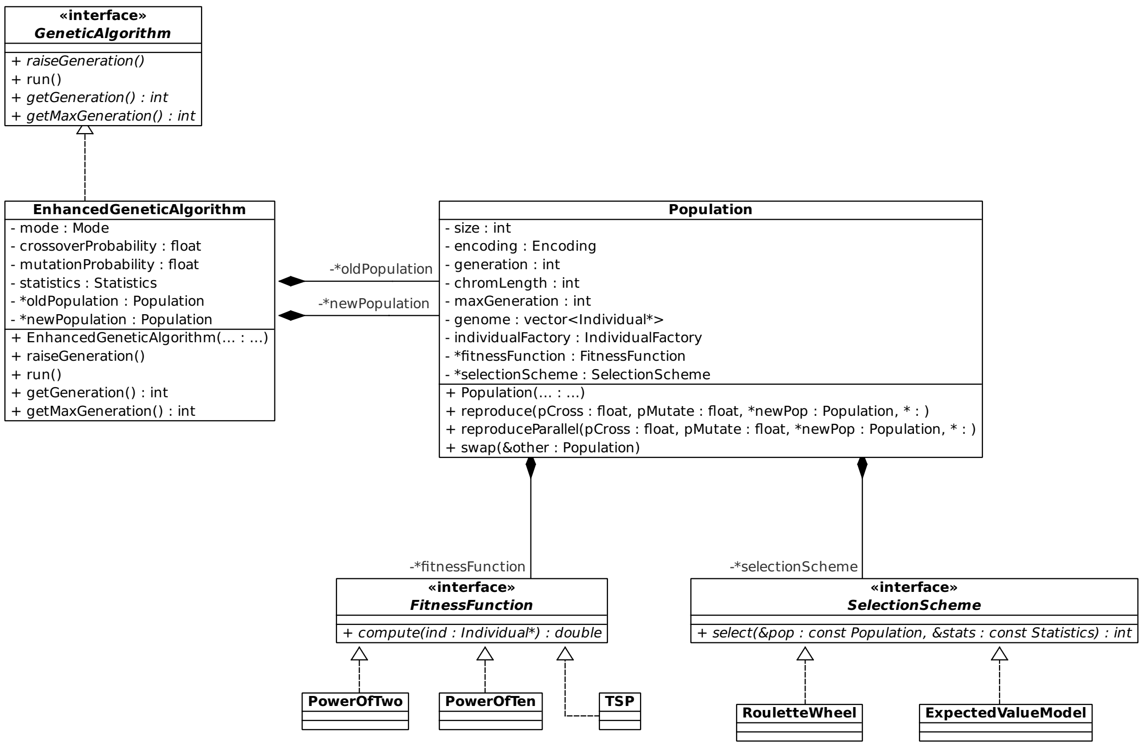
«interface»
GeneticAlgorithm
+ raiseGeneration()
+ run()
+ getGeneration() : int
+ getMaxGeneration() : int
EnhancedGeneticAlgorithm
- mode : Mode
- crossoverProbability : float
- mutationProbability : float
- statistics : Statistics
- *oldPopulation : Population
- *newPopulation : Population
+ EnhancedGeneticAlgorithm(... : ...)
+ raiseGeneration()
+ run()
+ getGeneration() : int
+ getMaxGeneration() : int
-*oldPopulation
-*newPopulation
Population
- size : int
- encoding : Encoding
- generation : int
- chromLength : int
- maxGeneration : int
- genome : vector<Individual*>
- individualFactory : IndividualFactory
- *fitnessFunction : FitnessFunction
- *selectionScheme : SelectionScheme
+ Population(... : ...)
+ reproduce(pCross : float, pMutate : float, *newPop : Population, * :)
+ reproduceParallel(pCross : float, pMutate : float, *newPop : Population, * :)
+ swap(&other : Population)
-*fitnessFunction
-*selectionScheme
«interface»
FitnessFunction
+ compute(ind : Individual*) : double
«interface»
SelectionScheme
+ select(&pop : const Population, &stats : const Statistics) : int
PowerOfTwo
PowerOfTen
TSP
RouletteWheel
ExpectedValueModel

Figure 2.6: Decoupled fitness function and selection scheme

Next up was the decoupling of the selection scheme. Again, an interface was used to accomplish extensibility utilsing the power of polymorphism and object composition.

2.3.3 The cost of modularity

All this changes did not come without a cost. The complexity in contrast to the SGA perceptibly increased with the EGA. Instancing an object of the EGA involves a lot of object composition. To illustrate this, a sequence diagram was created that depicts the process of object instantiation (see figure 2.7).

A factory was employed to reduce the complexity of object creation. To get a shared pointer to an EGA, one has to to create a `GAFactory`, call the `createGA(...)`-method and pass it some enumerated options like which type of problem should be solved, which selection scheme should be used and the likes. The factory will then compose the genetic algorithm in the fashion the user wants.

2.4 Performance evaluation

Both genetic algorithms were run with the objective to maximise the function

$$f(x) = \left(\frac{x}{coeff}\right)^{10}$$

where *coeff* has been chosen to normalise x. This function is rather trivial, so the maximum is found relatively fast. Nevertheless, increasing the population size to a meaningless high amount still has its impact on performance, therefore allowing us to draw conclusions.

The following base parameters were used in every test run[10]:
Crossover probability: 0.6; **Mutation probability:** 0.03

Generations	**Population** size	**SGA** time in seconds	**EGA sequential** time in seconds
100	1,000	0.175 s	4.850 s
	10,000	51.420 s	175.670 s
300	1,000	5.290 s	14.540 s
	10,000	156.600 s	522.160 s

Table 2.1: Several testruns with variable population size and generation number

Generally said, the SGA is faster by a factor of 3 to 4 compared to the EGA. For very small populations combined with a low generation number, its faster by an astounding factor of 27 – which can be considered an outlier since higher generation numbers like 300 relativise this. Additionally, it's uncommon to work with such small populations in the first place. It is assumed startup object construction is responsible for this gap (see 2.3.3).

[10]No compiler optimisations have been used during performance evaluation.

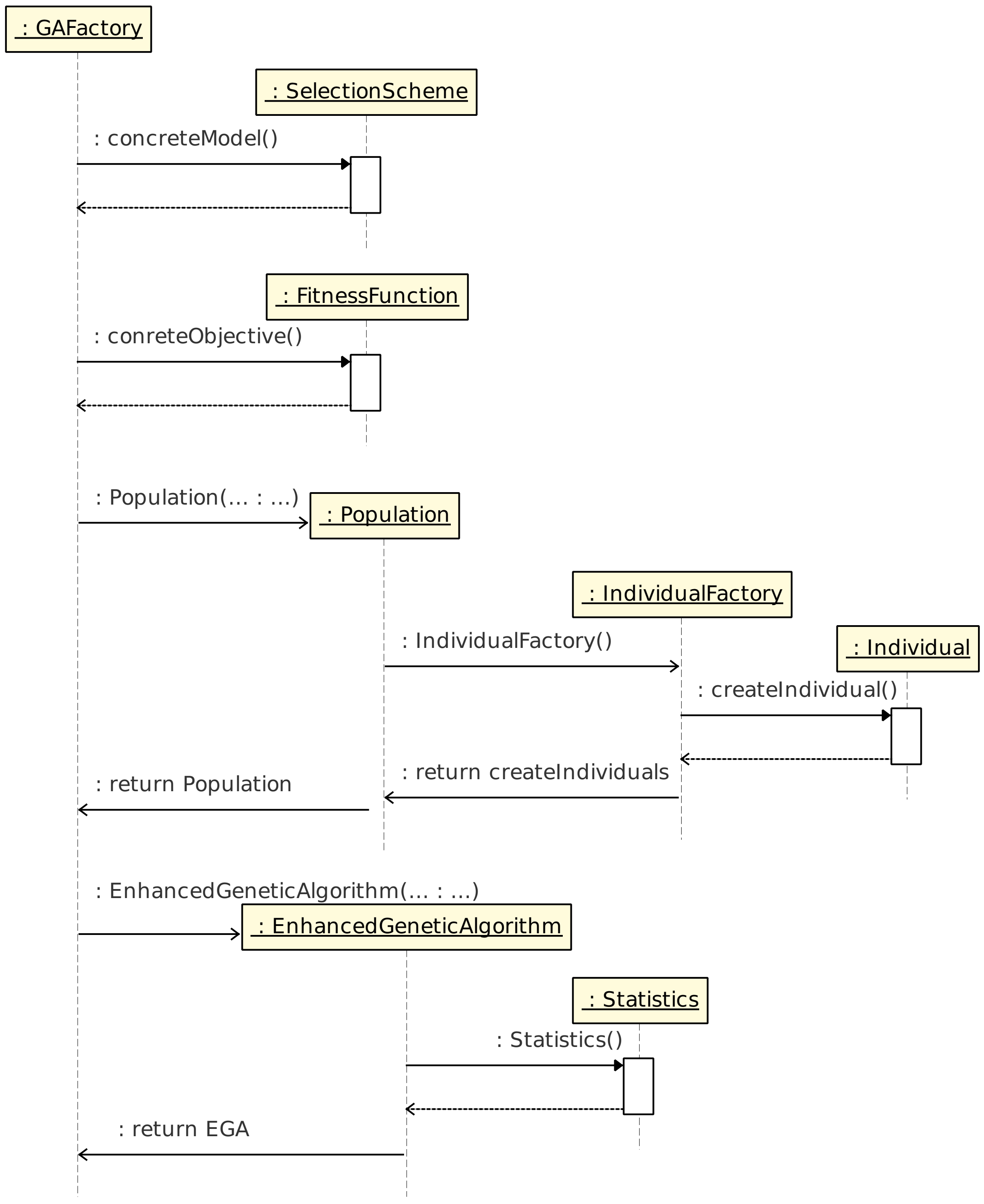

Figure 2.7: The sequence of object creation

Chapter 3

Approximating the Traveling Salesman Problems

The Traveling Salesman Problem (TSP) and the corresponding modifications necessary to approximate it using the enhanced genetic algorithm introduced in section 2.3 are covered in the following chapter.

At the beginning an introduction to the TSP in section 3.1 is given. Afterwards, the customisations necessary to adapt the GA to the TSP are elaborated on in section 3.2. Following, the approximation results of the EGA are evaluated in section 3.3. Finally, the performance is evaluated in section 3.4.

3.1 A short introduction to the Traveling Salesman Problem

A salesman has to visit several cities for business reasons. Since he is a witty salesman, he wants to find an optimal route which allows him to travel to every city exactly once. Also, he wants to minimise the traveled distance to the shortest amount possible in addition to visiting every city exactly once. Fundamentally, that's the traveling salesman problem.

The traveling salesman problem is a *np-complete combinatorial optimisation problem.* [10] What this means is described in the following section.

3.1.1 NP-complete

NP-complete is a complexity class of the computational complexity theory. NP is the abbreviation of *Nondeterministic Polynomial time*, whereas the suffix *complete* indicates it's a subclass of NP. "NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine." [11]

Simply put NP-complete problems could be described as "the set of all decision problems whose solutions can be verified in polynomial time" [11].

For the time being, it is unknown if there exists an algorithm that solves any of the NP-complete problems in an efficient manner. "Although any given solution to such a problem can be verified quickly, there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known." [11]

Since there is no efficient algorithm to solve them "[...] NP-complete problems are often addressed by using approximation algorithms in practice." [11] Genetic algorithms being one of the random approaches to approximate an optimal tour.

3.1.2 As a graph

For this to work, the TSP was required to be modelled as a graph as described in the following quote:

> *[The] TSP can be modelled as a graph: the graph's vertices correspond to cities and the graph's edges correspond to connections between cities, the length of an edge is the corresponding connection's distance. A TSP tour is now a Hamiltonian cycle in the graph, and an optimal TSP tour is a shortest Hamiltonian cycle.* (see [10])

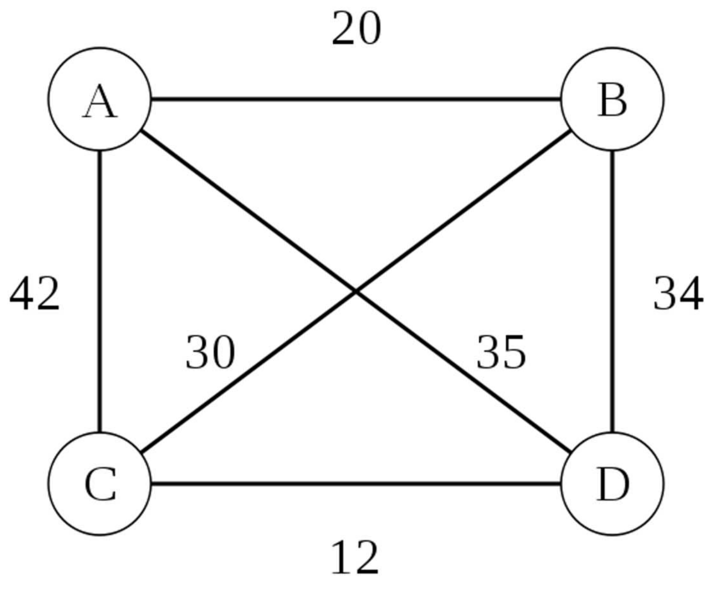

Figure 3.1: Weighted complete graph of 4 nodes (from Wikipedia [7])

The TSP approximated in this work is only the *symmetric* version, meaning the distance between cities is the same in both direction. Let's take a look at figure 3.1 for example: The distance from city A to city B is 20 (not further defined) units. For the symmetric TSP this means the distance from city B to city A will be exactly the same, that is to say 20 units.

3.2 Customising the genetic algorithm

A new encoding type had to be added to the genetic algorithm since a binary encoding would not fit the bill for the traveling salesman problem. The new encoding needed to be able to unambigously identify cities of variable amount and to store a tour in which every city would be visited exactly once.

3.2.1 Introducing permutation encoding

With permutation encoding every city is represented by a number. Each number is unique and unchangeably bound to the city it represents. The tour is defined by the ordering of the numbers. Say we've got eight cities identified by the numbers from 1 to 8. Then a possible tour could be looking like this:

5 3 4 6 7 8 2 1

Starting in city 5 the salesman travels to city 3, thereon to city 4, then to city 6, and so on. It is important to notice that every number must occur exactly once to comprise of a valid tour.

3.2.2 Implementing permutation encoding

Simple things first, storing the city numbers in a `std::vector<int>` solves the part of how the chromosome is saved. Whereas the ordering, as discussed before, represents the tour.

Now for the more difficult part: The crossover operator. The initial approach uses 3 steps.

1. Randomly select a crossover site *xsite*. Transfer all cities of *parent1* to *child1* and all cities from *parent2* to *child2* until the *xsite* is met.

2. Iterate over all cities in *parent2* and mark those already transfered to *child1*. Copy all **unmarked** cities into a temporary variable called *temp1*, preserving their ordering. Do the same for *child2* and *parent1* storing all unmarked cities in *temp2*.

3. Transfer all cities in *temp1* to *child1* beginning at the position following *xsite* ($xsite + 1$). Do the same for *temp2* and *child2*.

In contradiction to the implementation of the crossover operator for binary encoded individuals, mutation cannot be done simultaneously to crossover but has to be done afterwards. This has a simple yet far-reaching reason: Mutation for permutation encoded individuals means swapping two cities, thus it is modifying the ordering of cities. If mutation was done simultaneously to crossover it could not be guaranteed that the tour would not be invalidated by swapping a city that has already been transfered by the crossover operator with one that has not yet been transfered. The crossover operator could not tell apart cities it already transfered from cities that still needed to be transfered anymore, invalidating the effort to keep the tour valid by marking cities.

Consult figure 3.1 for the implementation in C++.

Listing 3.1: PermutationEncoded crossover implementation

```
// Transfer cities until the xsite is met (step 1)
for(int i = 0; i <= xsite; i++) {
    child1->setCityId(i, this->tour[i]);
    child2->setCityId(i, partner->getCityId(i));
}

// Transfer the cities of the other parent to the child,
// omitting already transfered cities since this would
// invalidate the tour (step 2)
if(xsite != (length - 1)) {
    vector<int> remainderPartnerTour;
    vector<int> remainderThisTour;

    // Iterate over all cities of one crossover mate and mark
    // cities which have already been transfered to the child
    for(int k = 0; k < length; k++) {
        bool isPartnerCityUsable = true;
        bool isThisCityUsable = true;

        for(int l = 0; l <= xsite; l++) {
            if(partner->getCityId(k) == child1->getCityId(l))
                isPartnerCityUsable = false;
            if(this->tour[k] == child2->getCityId(l))
                isThisCityUsable = false;
        }

        if(isPartnerCityUsable)
            remainderPartnerTour.push_back(
                partner->getCityId(k) );
        if(isThisCityUsable)
            remainderThisTour.push_back(this->tour[k]);
    }

    // Transfer remaining cities after the xsite
    for(int j = xsite + 1, i = 0; j < length; j++, i++) {
        child1->setCityId( j, remainderPartnerTour.at(i) );
        child2->setCityId( j, remainderThisTour.at(i) );
    }
}

// Randomly swap some cities. This has to happen after cross-
// over is done, else there's a risk of invalidating the tour
for(int i = 0; i < length; i++) {
    child1->mutate(i, probMutate, statistics);
    child2->mutate(i, probMutate, statistics);
}
```

3.2.3 Elitism

As it turned out, the genetic algorithm always got stuck at some point while trying to approxiamte an optimal tour for a given route. This could not be resolved by increasing the population size or the number of generations the GA ran. Also, parameters like crossover probability and mutation probability didn't change much.

The best fitness would always jump up and down, never going beyond a certain threshold. To solve this elitism was introduced to the genetic algorithm. Simply put, all elitism does is to ensure that a specified percentage of the best individuals of the last population will be present in the next generation again. In other words it *preserves genetic data that has proven well.* This way the best fitness cannot get worse since the best solution to the problem cannot go extinct.

Listing 3.2: Elitism implemented in the reproduction method

```
// Elitism: Preserve the best x% of the population
quickSort(0, size - 1);
int elites = size * 0.05;
int xsite = 0;

#pragma omp parallel for
for(int i = 0; i < size; i = i + 2) {
    spIndividual child1( newPopulation->genome.at(i) );
    spIndividual child2( newPopulation->genome.at(i + 1) );

    if(i < elites) {
        // Preserve the best mates
        spIndividual mate1( genome.at(i) );
        spIndividual mate2( genome.at(i + 1) );

        xsite = mate1->cross(
                    mate2,
                    child1,
                    child2,
                    0.0,         // no crossover!
                    0.0,         // no mutation!
                    statistics);
    }
    else {
        // Randomly select two mates with roulette wheel
        int m1 = selectionScheme->select(*this, statistics);
        int m2;
        do { m2 = selectionScheme->select(*this, statistics);
        } while(m1 == m2);
        spIndividual mate1( genome.at(m1) );
        spIndividual mate2( genome.at(m2) );

        xsite = mate1->cross( ... ); // lots of params...
    }
    // ...
}
```

Figure 3.2 illustrates how elitism was achieved. First the population needs to be sorted by fitness so we know where to get the best x indiviuals of the population. It's sorted in descending order, meaning individuals with a higher fitness will reside at the top, those with lower fitness at the bottom. This helps to reduce the sort effort a little since the preserved "elite individuals" will be stored in the top positions of the genome so they will already be sorted for the next pass. Then a specified percentage – 5% in this case – of the indivials will be copied without any modification to their chromosomes (crossover and mutation probabilities = 0.0 ensures this).

3.2.4 Partially matched crossover

David E. Goldberg introduced a partially matched crossover (PMX) in his publication [1]. For comparisons, PMX was added to the EGA. Figure 3.3 illustrates its functionality. For further details concerning the functionality of PMX, please refer to [1].

Listing 3.3: Partially matched crossover implementation

```
float nRandom = RNG::getInstance()->random();
int lo_xsite, hi_xsite;

// Randomly determine if crossover shall happen
if(nRandom < probCross) {
    // Difference of lo_xsite and hi_xsite should exceed 1
    // so cities lie in between the borders
    lo_xsite = RNG::getInstance()->range(0, (length-1));
    do{
        hi_xsite = RNG::getInstance()->range(0, (length-1));
    } while( abs(lo_xsite - hi_xsite) <= 1 );

    if(lo_xsite > hi_xsite) {
        int tmp = lo_xsite;
        lo_xsite = hi_xsite;
        hi_xsite = tmp;
    }
}
else {
    lo_xsite = (length - 1);
    hi_xsite = 0;
}

// Transfer all cities to the children...

// Here resides the main functionality of PMX!
for(int i = lo_xsite; i <= hi_xsite; i++) {
    child1->swap(i, child1->find_city(partner->getCityId(i)));
    child2->swap(i, child2->find_city(this->getCityId(i)));
}
```

`find_city(cityId)` simply locates and returns the position of the specified city.

3.2.5 Calculating the fitness

Last but not least we have to calculate the fitness, else the genetic algorithm would not find an optimal tour since he would not know if the search is making any progress, that is, if the tours are improving.

First it is necessary to calculate the distances traveled. This work exclusively makes use of the `EUC_2D` coordinates provided by TSPLIB [13]. `EUC_2D` simply means 2-dimensional euclidian coordinates. To calculate them an auxiliary method was written which can be reviewed in figure 3.4.

Listing 3.4: Auxiliary method to calculate an euclidian distance

```
double Tsp::calcEucDistance(
        const spCity a,
        const spCity b) const
{
    double distance = pow( (b->x - a->x), 2)
        + pow( (b->y - a->y), 2);
    distance = sqrt(distance);

    return distance;
}
```

Which is just a shorthand for

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The distance computation of a given tour is as simple as iterating over all cities and summing up the pairwise distances. See figure 3.5 for the implementation.

Listing 3.5: Computation of entire tour distance in the fitness function

```
double sumTraveledDistance = 0;

for(int i = 0; i < (ind->getLength() - 1); i++) {
    spCity a = (cities.find( ind->getCityId(i) ))->second;
    spCity b = (cities.find( ind->getCityId(i + 1) ))->second;

    sumTraveledDistance += calcEucDistance(a, b);
}

// return to the start of the tour
int lastCity = ind->getLength() - 1;
spCity a = (cities.find( ind->getCityId(0) ))->second;
spCity b = (cities.find( ind->getCityId(lastCity) ))->second;

sumTraveledDistance += calcEucDistance(a, b);
ind->setDistance(sumTraveledDistance);

double fitness =
    (1 / sumTraveledDistance) * ind->getLength() * 100;
// increase distance between better and worse chromosomes
fitness = pow(fitness, 2);
```

At the end it is important to return to the starting city. After all, the salesman does not want to spend the night under a park bench simply because someone forgot to plan for his trip home.

All that's left to do now is to translate the traveled distance somehow into a fitness value which increases as the distance decreases. The following approach was chosen:

$$fitness = \left(\frac{1}{\text{sumTraveledDistance}} \cdot \text{numOfCities} \cdot 100\right)^2$$

3.3 Evaluating the approximated results

Finally, tests could be run to see how well the EGA can approximate tours.

Note on used test data

The symmetric traveling salesman problem data from TSPLIB [13] was used as test data for all tests executed in this work. Only data using the `EUC_2D` coordinates were used.

3.3.1 The first test

The first test that ran was `berlin52.tsp` for which the EGA should find an optimal tour visiting 52 locations in the german capital. Figure 3.2 depicts how close the EGA using the initial implementation for the TSP and using Goldbergs PMX came.

Using a population of 30,000 over 500 generations the EGA found a route with a total distance of 9234.78. In comparison, the optimal solution found by [13] has a total distance of 7542. The result from the EGA is still a bit off from the optimal distance. PMX even found a solution that's a little bit closer with a total distance of 9081.63. Probably a larger population would have been necessary to get better results since the improvement seems to have gotten in a stalemate at generations over 350.

The tests were run given an execution time of 20 minutes and 33 seconds utilising a computer with 8 cores (2 Quad-core Intel(R) Xeon(R) E5335 @ 2.00 Ghz).

3.3.2 Another test

So the first test didn't get to the optimum. For the second test `eil51.tsp` was chosen as test data. Being a 51-city problem and therefore playing nearly in the same league as `berlin52.tsp` from the first test, a bigger population of 50,000 was chosen this time. Running over 500 generations again figure 3.3 shows the result of the test run.

The shortest tour being at a distance of 426, the EGA came pretty close with a distance of 486.74 for the initial approach and a distance of 481.28 for PMX. Obviously, the two results are pretty close to each other and to the optimal tour as well. Execution time was about 2 hours this time, which is a drastic increase from the first test run.

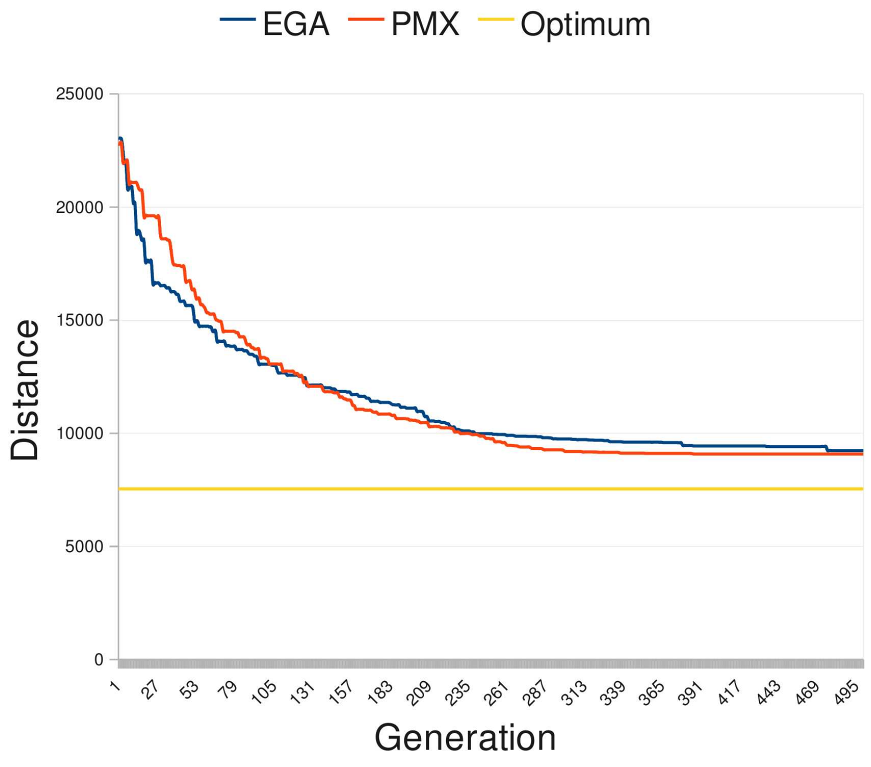

Figure 3.2: Approximation of an optimal tour with 52 locations in Berlin

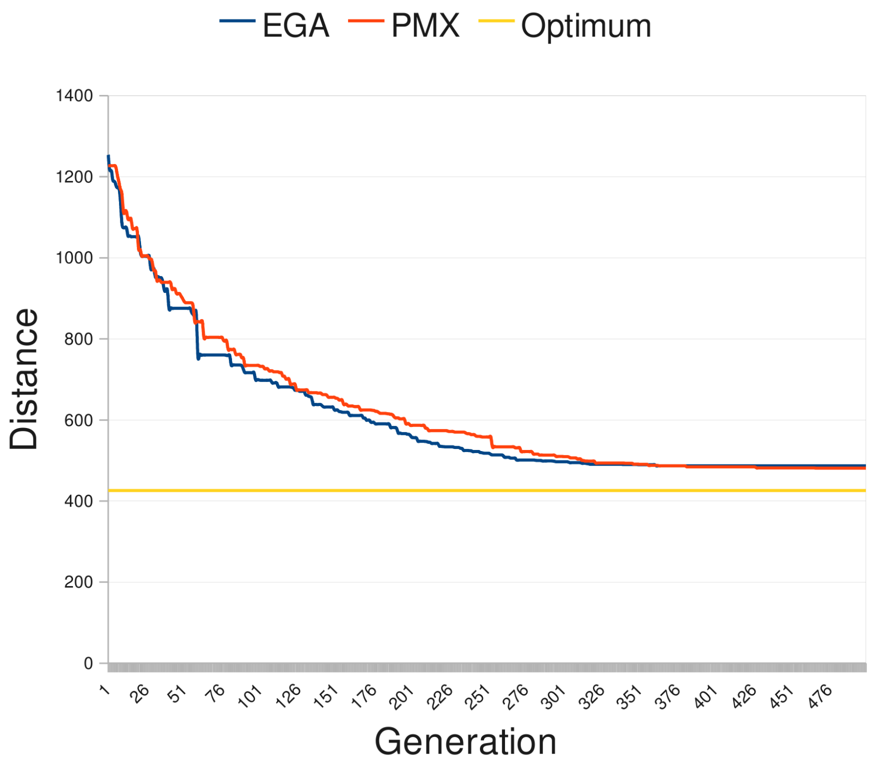

Figure 3.3: Approximation of a 51-city problem

3.4 Performance evaluation

Several tests were conducted using `st70.tsp` – a 70-city problem by Smith/Thompson based von `EUC_2D`-coordinates. Table 3.1 shows the results of tests run with the following parameters, unless noted otherwise:

Crossover probability: 0.6

Mutation probability: 0.03

Generations: 100

Population	**EGA TSP**		**EGA TSP with PMX**	
size	time	best fitness	time	best fitness
1,000	10.200 s	9.68	10.963 s	10.92
10,000	53.909 s	11.14	49.799 s	13.78
Mutation: 0.2				
1,000	15.895 s	6.30	8.956 s	6.67
10,000	59.168 s	6.70	52.569 s	6.50
Crossover: 0.8				
1,000	12.789 s	10.19	11.394 s	10.84
10,000	55.029 s	11.28	51.548 s	10.32

Table 3.1: Testruns with variable crossover and mutation probability using *st70.tsp* as test data

Both, the TSP of this work as well as Goldbergs PMX, were run in parallel for this performance evaluation.

Based on table 3.1 it can be said that the TSP crossover operator of this work is pretty close to Goldbergs PMX operator performance-wise, with PMX achieving a little bit higher fitness value most of the time and being a tad faster.

Increasing mutation from 3% to 20% did not improve the test run, quite the opposite: As execution time rose significantly for the TSP operator of this work, the fitness value sunk to the worst value of all tests for both. PMX did not suffer that much from the increased mutation probability, in the case with a population of 1,000 it was even faster then before which is probably an outlier.

Contrary, increasing the crossover probability had a rather small impact on performance increasing execution time by a small amount while increasing the fitness value a little bit, too, for the TSP of this work. On the other side it did worsen the results of PMX remarkably.

To sum it up, PMX had its best results using the initial parameters while increasing the crossover probability to 80% both increased the execution time and fitness values of the TSP operator of this work. Generally, PMX seems to be the marginally better operator.

Chapter 4

Parallelising the genetic algorithm

With the genetic algorithm being practically finished at this point, the following chapter covers the last task to be done: Parallelising the genetic algorithm. Beginning in section 4.1 with some thoughts on the implications this step will cause, a discussion about the critical areas follows afterwards. Going on, the act of parallelising itself is elaborated in section 4.2. Concluding, the hour of truth has arrived: The parallelised genetic algorithm will be evaluated and compared against the sequential version in section 4.3.

4.1 Preceding thoughts on the parallelisation

Looking over the code with parallelisation in mind one can find ample of opportunities to parallelise even while looking at the code with only parallelising `for`-loops in mind – which are by far the easiest and most obvious starting points to parallelisation. But there is a cost to parallelisation in that it causes overhead: Communication has to be handled somehow between the branched off threads and the main thread, possibly they even need to be synchronised with the main thread at some point. Even more, when there are some variables involved the question arises how to handle them in case of concurrent access: Do threads just concurrently read the variables or is there a risk of concurrent write as well? In case of write access, is it desirable to lock them or would that nullify the performance gain of parallelisation in the first place?

Local copies of those variables would solve the problem of concurrent write access, but is interconnected with the cost of allocating those local copies, thus expending memory and a little bit of computational time – time we had in mind to decrease by parallelising. Though its a small cost, it still is a cost and small costs can sum up – it's best to keep that in mind. Also, at the end of the parallelised code block when the branched threads are done and terminated by the operating system, the local copies of each branched thread needs to be reunited with the initial variable again.

With all these considerations in mind it is not advisable to recklessly parallelise every `for`-loop one encounters in the program code but to think about which ones yield more benefits than they cost. Then, there is code that is sequential by nature and cannot be parallelised at all.

Leaning on the Pareto Principle [12] we can safely assume most of the code does not even need to be parallelised since the overhead cost would nullify or even exceed the benefits. Some times the increase in performance is so marginal it's simply not worth the effort.

4.1.1 Critial areas

The first step would be to profile the program and search for critical areas or hot spots that are most likely to be executed most of the time or many times. Those are the spots we need to focus our attention on.

The critical areas performance-wise can be deduced rather easily in this case. Crossover and mutation both happen while reproduction takes place. Reproduction itself is a `for`-loop iterating over the entirety of the population – which is typically rather large for the tsp. At runtime, the algorithm spends most of the time reproducing new generations. Since the population tends to get rather large for the TSP, this is the first place to go.

4.2 Realisation with OpenMP

For easy comparability, the existing sequential methods were not altered to run in parallel. Instead, additional methods were added bearing the suffix *Parallel*. An option was added to allow the user to choose between sequential and parallel mode by appending `s` or `p` as startup parameter after specifying the problem.

`./bin/ega tsp s`	Starts with TSP in sequential mode
`./bin/ega p2 p`	Power of Two (x^2) in parallel mode

4.2.1 The reproduction method

Parallelising the `reproduction(...)`-method of `Population` was the first approach. As can be seen in figure 4.1, reproduction is nothing more than one single and, above all, expensive `for`-loop iterating over the entire population, randomly picking two individuals from `old_pop`, crossing them, and storing the offspring in `new_pop`.

Parallelising reproduction implicitly parallelises crossover and mutation, too, since they are called within reproduction meaning each thread reproduces a certain section of the population, including crossing and mutating that section.

OpenMP is a very powerful framework in that it allows the developer to focus on the critical areas instead of hassling with the implementation details of threading. `#pragma omp parallel for` takes care of the actual act of parallelisation, creating as many threads as feasible (e.g., one for each logical processor) or instructed, handing each thread a distinct chunk of the loop to execute.

Listing 4.1: Parallelised reproduction method of class Population

```
#pragma omp parallel for     // this is all that's necessary
                             // to parallelise a for-loop
for(int i = 0; i < size; i = i + 2) {
    spIndividual child1( newPopulation->genome.at(i) );
    spIndividual child2( newPopulation->genome.at(i + 1) );

    if(i < elites) {
        // Preserve the best individuals...
    }
    else {
        // Randomly select two mates and cross them...
    }

// Do some basic set up for the children
// ...
}
```

4.2.2 The crossover method

Parallelising the `crossover(...)`-method without parallelising the outer reproduction method did not yield any significant performance benefit – in fact the performance gain was barely noticeable, lying somewhere around 1-2%.

Listing 4.2: Parallelised crossover method of a permutation individual

```
#pragma omp parallel for    // First parallelisation
for(int i = 0; i <= xsite; i++) {
    // transfer cities until xsite is met
}

#pragma omp parallel for    // Second parallelisation
if(xsite != (length - 1)) {
    vector<int> remainderPartnerTour;
    vector<int> remainderThisTour;

    // Iterate over all cities of one crossover mate to mark
    for(int k = 0; k < length; k++) {
        bool isPartnerCityUsable = true;
        bool isThisCityUsable = true;

#pragma omp parallel for    // Third parallelisation
        for(int l = 0; l <= xsite; l++) {
            // mark city if already transfered
        }

        // store city in remainder vector if not marked
    }

#pragma omp parallel for    // Fourth parallelisation
    for(int j = xsite + 1, i = 0; j < length; j++, i++) {
        // transfer remaining cities after the xsite
    }
}
```

It is assumed that the overhead of creating and terminating threads in the `crossover(...)`-method is too costly while providing just a diminuitive acceleration. As can be seen in listing 4.2 four separate parallelisations are necessary to achieve parallelisation in crossover.

For this reason the first approach of parallelising reproduction was chosen.

4.3 Performance evaluation

Similar to the performance evaluation in chapter 3 section 3.4, `st70.tsp` was used to test the performance of the EGA with the following parameters, unless otherwise noted:

Crossover probability: 0.6

Mutation probability: 0.03

Generations: 100

Population	**EGA sequential**		**EGA parallel**		**SU**
size	time	best fitness	time	best fitness	factor
1,000	14.170 s	12.08	10.200 s	9.68	1.4
10,000	294.610 s	14.55	53.909 s	11.14	5.5
Mutation: 0.2					
1,000	23.302 s	6.69	15.895 s	6.30	1.5
10,000	306.410 s	7.12	59.168 s	6.70	5.2
Crossover: 0.8					
1,000	27.940 s	12.42	12.789 s	10.19	2.2
10,000	297.165 s	14.15	55.029 s	11.28	5.4
Generation: 300					
1,000	41.790 s	20.21	17.400 s	15.73	2.4
10,000	833.031 s	29.89	135.914 s	25.56	6.1

Table 4.1: Comparing the sequential ega to the parallel version

The same effect as in the previous two performance evaluations can be observed: For smaller populations combinded with less generations the difference in performance diminishes. Thus it is not surprising to see the parallel version being only moderately faster compared to the sequential version for populations of 1,000 individuals running over 100 generations.

Though it should be noted that the execution time of the parallel EGA was more than cut in half when the crossover probability was increased to 80%. The same can be said for the test run over 300 generations.

By dividing the execution time of the sequential version through the execution time of the parallel version we get the factor of speedup (abbreviated **SU** in table 4.1) achieved by parallelisation.

The factor of speedup for populations of 1,000 and a generation runtime of 100 is around 1.4 and 2, meaning the program runs approximately 40-100% faster on a parallel machine with 8 cores. According to figure 4.1 this is not quite what we can expect of a 8 core computer, so it's a rather weak speedup. A program with a 95% parallel portion may have a speedup up to a maximum of factor 6.0, whereas the EGA resides within the factor of speedup a program would achieve with only 50% parallel portion. Does this mean the genetic algorithm has only a small parallel portion?

Looking at the results with a population of 10,000 individuals we find ourselves at a speedup factor of 5.2 to 6.1 which is pretty as much as can be expect with the used test configuration. 6.1 is even a little over the maximum possible speedup, which can be justified by measuring inaccuracy.

Oddly enough, the fitness generally seems to be higher by a small amount for the sequential version. No explanation was found for this phenomenon, but since the speedup by far outweights the loss of fitness, this is not a serious issue.

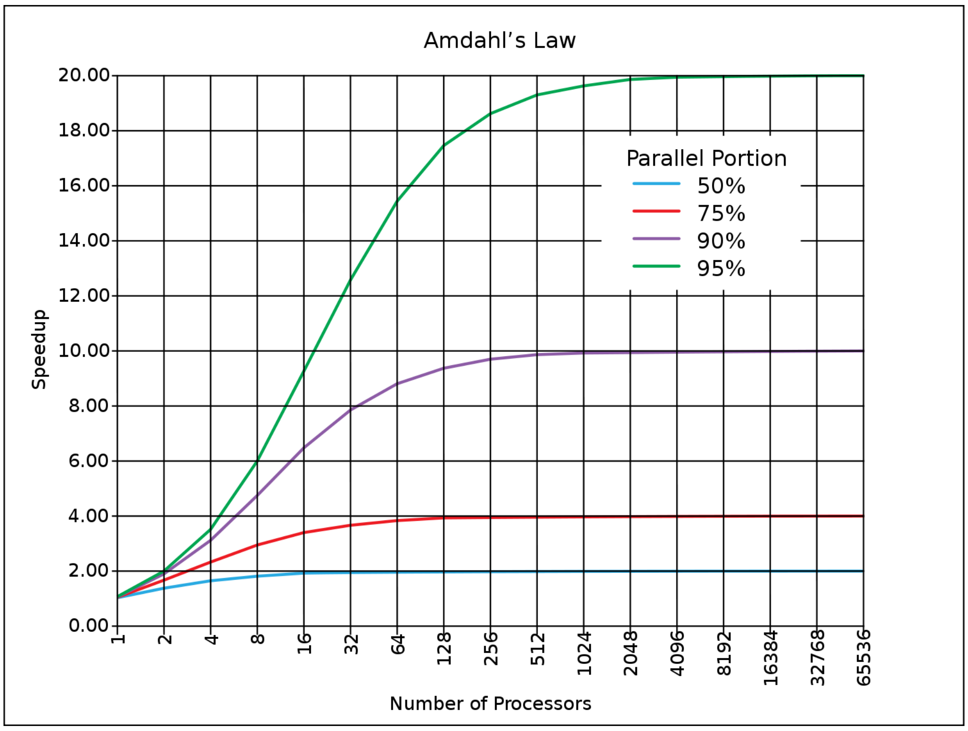

Figure 4.1: Amdahl's Law showing the maximum possible speedup of parallelisation (Image from Wikipedia [7])

So, it is true that genetic algorithms do posses intrinsic parallelism providing the best speedup possible for parallel programs. Although the EGA needs a certain workload to maximise its speedup benefit by parallelisation, which is not hard to find considering complex problems like the TSP or other NP-complete problems.

Chapter 5

Conclusion

Getting involved with genetic algorithms was an interesting and rewarding endeavour. Genetic algorithms can be applied in a broad area of application, ranging from optimisation problems like the traveling salesman problem as discussed in this work to artifical intelligence. Over the course of this work the fundamentals of genetic algorithms have been learnt and applied. First in a simple inflexible approach, then with a more sophisticated design. Last, it was proven that genetic algorithm posses a strong intrinsic aptitude for parallelisation.

Parallelisation – especially the feature of parallelisability – is very important in times when single core processors are at their physical limits. Todays and future programs have to be designed and developed with multi-core computers in mind to fully exploit the possibilites of tomorrows personal computers. Even at the time of this work it is virtually impossible to purchase a computer with only a single core processor in stores. Furthermore, 8-core processors are barely the peak. It can be expected that future computers will have several hundred cores or more. As figure 4.1 indicates is the limit expected to lie somewhere around 4096 cores, which is still far ahead from a present-day perspective. But as software development often takes several years until completion, we have to design and develop the software not for todays computers but for tomorrows.

Considering the complexity of threaded programming coupled with the complexity of the paradigm of parallelism it is likely that parallelisation frameworks will prevail in the future, for it is unlikely a way will be found to reduce the complexity of the paradigm itself. All that can be done is to reduce the complexity of "manually" applying threading by leaving the implementation details for frameworks to take care of. This way developers can focus on the difficulties of thinking parallel – which is not usual for humans that are sequential by nature.

Appendix A

Ehrenwörtliche Erklärung (Declaration of Honour)

Ich versichere hiermit, dass ich meine Projektarbeit mit dem Thema

Erstellung eines genetischen Algorithmus zur Approximation des Traveling Salesman Problems (Rundreiseproblem) in einem parallelen System

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Michael Barth
Aalen, 15.03.2009

Betreuung

Die Betreuung fand statt durch Professor Dr. Christoph Karg der Hochschule für Technik und Wirtschaft Aalen. Vielen Dank dafür an dieser Stelle!

Appendix B

List of Abbreviations

D

DNA – Deoxyribonucleic acid

E

EGA – Enhanced Genetic Algorithm

G

GA – Genetic Algorithm

R

RNG – Random Number Generator

S

SGA – Simple Genetic Algorithm
SU – Speedup

T

TSP – Traveling Salesman Problem

X

xsite – Crossover site

Bibliography

[1] **David E. Goldberg**
Genetic Algorithms in Search, Optimization & Machine Learning
Addison-Wesley; 1989

[2] **OpenMP Architecture Review Board**
The OpenMP API specification for parallel programming
http://openmp.org/wp

[3] **Scott Meyers**
Effective C++: 55 Specific Ways to Improve Your Programs and Design
Addison-Wesley; 1997

[4] **Jean-Philippe Rennard Ph.D.**
Introduction to Genetic Algorithm
http://www.rennard.org/alife/english/gavintrgb.html; May 2000

[5] **William H. Press, Saul A. Teukolsky, and William T. Vetterling**
Numerical Recipes in C: The Art of Scientific Computing
Cambridge University Press; 1992

[6] **Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides**
Design Patterns. Elements of Reusable Object-Oriented Software.
Addison-Wesley Longman; 1995

[7] **Wikipedia**
http://en.wikipedia.org/

[8] **Wikipedia**
Evolutionary computing
http://en.wikipedia.org/wiki/Evolutionary_computation

[9] **Wikipedia**
Holland's schema theorem
http://en.wikipedia.org/wiki/Holland%27s_Schema_Theorem

[10] **Wikipedia**
Traveling Salesman Problem
http://en.wikipedia.org/wiki/Traveling_salesman_problem

[11] **Wikipedia**
NP-complete
http://en.wikipedia.org/wiki/NP-complete

[12] **Wikipedia**
Pareto principle
http://en.wikipedia.org/wiki/Pareto_principle

[13] **Ruprecht-Karls-Universität Heidelberg**
TSPLIB
http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

List of Figures

List of Tables

Listings