Master Thesis

Investigation of the fitness uniform selection scheme in evolutionary algorithms

Author(s):
Pongan, Daniele

Publication Date:
2002

Permanent Link:
https://doi.org/10.3929/ethz-a-004296467

Rights / License:
In Copyright - Non-Commercial Use Permitted
Daniele Pongan

Investigation of the Fitness Uniform Selection Scheme in evolutionary algorithms

Diploma Thesis at the
Institute of Computational Science

Principal Adviser: Prof. Petros Koumoutsakos, Computational Science, ETHZ
Co-Adviser: Dr. Marcus Hutter, Istituto Dalle Molle di Studi sull’Intelligenza Artificiale

March 2002
Abstract

Evolutionary algorithms are gaining importance as a way to solve problems for which no simple solution algorithms are available. One of the critical points in using them is the choice of the right selection pressure which influences the ratio of exploration/exploitation of the genetic algorithm. A wrong selection pressure may cause the failure of the entire algorithm.

A new selection scheme which avoids at all the decision of a right selection pressure has been developed. Its name is FUSS (Fitness Uniform Selection Scheme) and the goal of this diploma work was to investigate it, especially comparing its performance on various test problems with the performance of other selection schemes. During the work simulations were also run to confirm with practical evidences some results anticipated by theoretical analysis.

FUSS tries to generate a selection pressure towards sparsely populated fitness regions, rather than towards better fitness values, in order to prevent premature convergence of the algorithm to local optima. We will show how this makes FUSS more suitable than standard selection schemes for some kinds of problems.

Zusammenfassung


Standard Selektionschemata erzeugen einen Selektionsdruck gegen bessere Fitnesswerte. FUSS erzeugt hingegen einen Selektionsdruck gegen Fitnessgebiete, die wenige Individuen enthalten, um eine vorzeitige Konvergenz zu vermeiden. Wir werden zeigen, dass FUSS für einige Probleme mehr geeignet ist als gewöhnliche Standard-Selektionschema ist.
## Contents

1. Introduction .................................................. 7

2. The FUSS algorithm ........................................... 9  
   2.1. Standard selection schemes ................................. 9  
   2.2. Description of FUSS ...................................... 10  
   2.3. Differences between FUSS algorithm and incremental GA ........................................ 12  
   2.4. The Scale Independent Fitness Selection (SIF) ...................................... 15

3. Details on implementation ..................................... 17  
   3.1. The GALIB genetic library ................................ 17  
   3.2. Implementation of FUSS .................................. 17  
   3.3. Implementation of SIF ................................... 18  
   3.4. Implementation problems ................................ 19

4. Inspected problems ............................................. 21  
   4.1. A simple example problem ................................ 21  
      4.1.1. 2-dimensional example ................................ 21  
      4.1.2. 3-dimensional example ................................ 25  
      4.1.3. Continuous fitness ................................... 26  
   4.2. Traveling Salesman Problem ............................... 30  
      4.2.1. Problems variants .................................... 30  
      4.2.2. First test: generations needed for solution .................... 32  
      4.2.3. Comparison of FUSS and SIF on the first test .................. 36  
      4.2.4. Second test: fitness after fixed generations ................ 36  
      4.2.5. Third test: solution for real problems .................... 41  
      4.2.6. Fourth test: comparison of fitness evolution ................ 43

5. Discussion of the results .................................... 55  
   5.1. Strengths of FUSS and SIF ............................... 55  
   5.2. Weaknesses of FUSS and SIF ............................ 56  
   5.3. Comparison of FUSS and SIF ............................. 56

6. Conclusions and future directions .......................... 57  
   6.1. Conclusions ............................................... 57  
   6.2. Future directions ........................................ 58

A. Program Code ................................................. 63  
   A.1. Genetic algorithms ...................................... 64  
      A.1.1. Fitness Uniform Selection Scheme (FUSS) code .............. 64
A.1.2. Scale Independent Fitness (SIF) code .......................... 67
A.1.3. Incremental algorithm with roulette selection code ........... 71
A.1.4. Incremental algorithm with tournament selection code .......... 74
A.1.5. Random search algorithm code ................................ 76
1. Introduction

The name "evolutionary algorithms" (EAs) identifies a wide range of strategies that can be used to solve optimization problems. We may define all these methods as algorithms which utilize in some way the collective learning process of a population of individuals (also called genomes) representing solutions to the problem [10].

Genetic algorithms (GAs) belong to the evolutionary algorithms. They employ mutation and crossover operators to search the solution space, creating new genomes. A selection operator is used to select the most promising individuals, in order to give a selection pressure towards better fitness values. The choice of the right selective pressure is a key issue when working with genetic algorithms. If it is too large, the algorithm may converge soon to an only local optimal solution, exploiting too much. A high selection pressure may cause an excessive deletion of genetic material, preventing the genetic algorithm from escaping from local optimum wells (premature convergence). On the other hand, some selection pressure is essential to ensure the convergence to good problem solutions. Without it, an evolutionary algorithm could not be effective and would find no suitable solution at all.

Common techniques for avoiding premature convergence are based on dynamic adaptation at run-time of the parameter values for mutation, recombination and selection, which influences the selection pressure [11, 12, 13, 14].

The Fitness Uniform Selection Scheme (FUSS), developed by Marcus Hutter at IDSIA (Istituto Dalle Molle di Studi sull'Intelligenza Artificiale), tries to avoid premature convergence by forcing a selection pressure towards sparsely populated fitness regions [1]. FUSS does not favour individuals with a good fitness, making their reproduction more probable. It instead tends to give more reproduction chances to genomes whose fitness are rare among the population. Since a typical population in a genetic algorithm is composed by many unfit and very few fit individuals, this should in fact help the emergence of good solutions.

In this work we will examine the behaviour of FUSS and of a derived selection scheme, SIF, on several test problems. The performance of an algorithm using FUSS will be compared especially with that of an incremental algorithm, which is structurally the most similar to it.

FUSS was implemented as C++ code using a freely available genetic algorithm library.

This report is structured as follows:

- In chapter 2, a description of FUSS and its characteristics is given. The differences between an algorithm using FUSS and a standard incremental algorithm\(^1\) are highlighted. After that, SIF is also described and explained.

- In chapter 3, an overview of the implementation is given. The genetic algorithm library used is shortly described and the peculiarities of the implementation of FUSS and SIF

\(^1\)We use the expression "standard incremental algorithm" to mean an incremental algorithm with fixed population size employing standard selection schemes.
are shown. This is needed in order to understand some problems encountered during the testing phase.

- Chapter 4 presents the different problems used to test the algorithms and to compare them. For every problem a brief description, some details of the implementation and the outcomes are depicted.

- In chapter 5 we discuss the results obtained, illustrating the weak and the strong spots of the algorithm inspected.

- In chapter 6, finally, the conclusions are presented and some future direction on this topic are addressed.

- The appendix contains the listing of the source code for the genetic algorithms written and employed during this work.

**Note:** Throughout this report, the expressions "evolutionary algorithm" and "genetic algorithm" are used quite indifferently. The reader should be aware that nowhere the term "evolutionary algorithm" is referred to other methods such as evolution strategies or evolutionary programming.
2. The FUSS algorithm

A large variety of genetic algorithms exists [8]. Some of them replace the entire population with new individuals at every generation. Others replace only a fixed amount of individuals, or produce multiple populations between which migration may occur.

The incremental GA is more similar to the FUSS algorithm than every other EA. So here we will describe it, showing also its normally used selection schemes\(^1\), in order to better explain the difference between it and the Fitness Uniform Selection Scheme.

The incremental GA works by adding only one "child"\(^2\) into the population at every generation. The child is produced by creating a copy of a selected individual, the parent, and mutating it. If we also want to allow crossovers between individuals, two parents may be selected; in that case the child will be the result of the crossover of these two individuals.

After having created the child, it is added to the population. Since the size of the population is normally fixed, one individual must be removed so that the child can replace it. This element is then definitively discarded. Normally the worst genome of the population is replaced; however, other replacement schemes are possible, such as random choice.

The entire process is repeated, like in every GA, until a suitable stopping criterion is met.

2.1. Standard selection schemes

Every selection scheme normally used in the incremental genetic algorithm tends to increase the average fitness of the population by selecting with larger probability individuals with larger fitness.

- The roulette selection scheme, used in maximization problems, selects every individual with a probability which is proportional to the element’s fitness. Obviously, genomes with largest fitness have an advantage.
- In tournament selection, individuals compete against each other until a suitable number of winners is chosen.
- Ranking selection sorts the genomes according to their fitness. The selection likelihood of every individual is then proportional to the individual’s rank in the sorted list. This selection scheme has been demonstrated to be equivalent to tournament selection and therefore will not be used in our simulations [17].

\(^1\)Nearly every selection scheme used in the incremental GA may also be used in other genetic algorithms. If the selection is not deterministic, a scheme that selects one individual can also choose fifty of them. However, this doesn’t apply if no random elements are involved in the choice (“get the individual with the best fitness” will return always the same element).

\(^2\)Or two children, in some implementations.
• Using truncation, only the \( n \) best individuals are selected (if \( n = 1 \), we have the "select the best genome" scheme).

• Other selection schemes are a combination or variation of these above. In any case, all of them are based on the fitness of the individuals and favour elements which score better.

All these usual selection schemes push the population towards a better mean fitness, removing from population the worst individuals. As we have explained in chapter 1, this may be responsible for the premature convergence of the genetic algorithm, when the problem that must be optimized is not a simple one.

FUSS tries to prevent the premature convergence of the algorithm by removing the source of that behavior: the selection pressure towards better fitness values. It does not discriminate between best and worst fitness. It uses instead a new approach to the problem.

2.2. Description of FUSS

FUSS bases on the idea that we are not interested in a population with a good fitness average; what we really want is to find a good solution for the problem at hand. This means that our genetic algorithm should try to find an individual with a fitness which is good enough, even if the rest of the population scores horribly on the given fitness function. How is it possible to find such an individual, but at the same time to prevent getting stuck in a local optimum?

FUSS does that by selecting the individual which will reproduce without directly looking at its fitness. It computes instead a fitness value and then selects the individual whose fitness lies closest to the desired value (or it randomly selects one of the genomes with the desired fitness, if more than one are present). Since the desired fitness is taken at random from all possible fitness values of the population, individuals having a fitness which is rare in the population have a large probability of being chosen. On the contrary, genomes with a common fitness level have only a small selection likelihood. This should permit the exploration of all possible fitness levels, so that a suitable solution is finally found.

Moreover, the absence of a selective pressure towards better fitness should prevent getting trapped in local optimum well.

Now a more schematic description of the FUSS genetic algorithm is given:

• Assume that the minimum and the maximum fitness in the actual population are \( \text{fitness}_{\text{min}} \) and \( \text{fitness}_{\text{max}} \) respectively.

• Select a suitable valid fitness \( f \in [\text{fitness}_{\text{min}}, \text{fitness}_{\text{max}}] \), normally by uniformly selecting a random value in the given interval.

• Now select the individual in the population with fitness nearest to \( f \). If more than one genome fulfills this requirement, one of them is randomly chosen. This could easily happen if fitness is expressed with integer numbers; in that case, many genomes may share the same fitness.

• If crossover is allowed, another individual may be selected. The child will then derive from the crossing of the two parents.
• Mutate the genome. Many genetic algorithms let the individual mutate with a probability \( < 1 \). For FUSS, it is important that every child created is different from the parent. If it is not so, the new individual will simply add nothing new to the algorithm, since it will occupy the same fitness level of its parent (more details on this in section 2.3). So, if the child was not created using the crossover operator, it should be mutated.

• Now add the newly created genome to the population, without removing any other individual. With FUSS no replacement of old genomes occurs. The population size keeps growing\(^3\), and so should do the fitness variety.

As it can be seen from the algorithm description, FUSS does not favour individuals with good fitness, nor does it penalize genomes which have bad fitness values. Genomes have an advantage in reproducing themselves when they are situated in scarcely populated fitness regions. If many individuals share the same fitness level, their probability of being chosen to become parent of new genomes decreases. Even if their fitness value is selected by the algorithm, every one of the \( n \) genomes with that fitness is chosen with a \( \frac{1}{n} \) likelihood. On the other hand, a single individual possessing a fitness value much different from every other fitness will be chosen not only if its fitness is selected by the algorithm, but also if a near value is chosen. This increases a lot its probability of reproducing (see figure 2.1 for an example).

![Figure 2.1](image)

Figure 2.1: In this figure the fitness distribution of a population of six individuals is shown. Five of them have a fitness value 3, while only one (\( a \)) has fitness 0. If a fitness value between 0 and 1 is selected by the algorithm, \( a \) will certainly be chosen and will reproduce. Since FUSS selects the fitness uniformly, the probability that \( a \) reproduces is \( \frac{1}{6} \). On the other hand, \( b \) reproduces if fitness 2 or 3 is selected and \( b \) is chosen among its four “neighbours”. Its probability of reproduction is only \( \frac{1}{2} \times \frac{1}{3} = \frac{1}{6} \), much smaller than for \( a \).

Clearly, genomes with rare fitness values should be selected for reproduction most of the times. Since their children are always mutated (or, at least, are the result of a crossover), the newly created genomes should begin to occupy also the near fitness levels, which perhaps were empty until then. New fitness levels should also be “discovered”, changing the values of \( \text{fitness}_{\text{min}} \) and \( \text{fitness}_{\text{max}} \). At the same time, however, the fitness values which were

---

\(^3\)This is true only for the theoretical case. When simulating on real computers, a maximum size for the population must obviously be set, since the memory of any real machine is not infinite (see section 3.2 for details).
rare at the beginning of the computation should become more and more frequent among the population individuals. This decreases the selection likelihood for the corresponding genomes, which were often selected during the first generations.

As a final result, FUSS tends to spread the population individuals among all available fitness levels, "searching" all of them. But since no genome is removed from the population, there is no risk of abandoning a search path which, although ineffective at the moment, can produce much better individuals with a few mutations. The "final product" of the FUSS algorithm is a population where the individuals are distributed over all possible fitness levels (see also figure 2.2).

It must also be pointed out that, judging from its behaviour, FUSS will rapidly find some good solutions for a problem only if the fitness values of these good solutions are rare in the initial population. If the starting population shows a high density of individuals with a nearly optimal fitness and a low density of unfit individuals, the latters will be initially favoured. So, FUSS may also cause a selection pressure towards bad solutions.

We emphasize here the strong points of the FUSS algorithm:

**selection based on fitness distribution:** the Fitness Independent Selection Scheme prevents the algorithm from choosing only (or with a much larger likelihood) good individuals for reproduction. This other way of acting can be dangerous when the function that must be optimized is not simple and optimal solution are found only "passing through" bad ones. In addition to that, the fact that also bad individuals may reproduce prevents the algorithm from getting stuck in local optimum well. In standard EAs, a wrong mutation may cause the population to evolve in a bad direction. With FUSS this is avoided, since no "evolving path" is abandoned.

**reproduction possible for all genomes:** all individuals may be selected for reproduction, at least with a small probability. Other selection schemes, like truncation, do not allow that. This guarantees that if an optimal solution can be found by mutating one of the individuals in the actual population, given that no genome is ever removed, sooner or later that mutation will occur and so the solution will be found.\(^4\)

**no genome removed:** in the theoretical definition of FUSS\(^5\) no replacement is considered and newly created individuals are simply added to the population. This prevents any genetic material from being discarded. In usual genetic algorithms, it may happen that an optimal genome is never produced because the genetic material needed to "build" it was removed from the population. This can not occur when using FUSS. Even in practical, real experiments, when sooner or later some genomes must be removed, FUSS does it in an efficient way, reducing the risk of deleting an important individual (see chapter 3).

### 2.3. Differences between FUSS algorithm and incremental GA

As it may be noticed, we talk of FUSS not only as a new selection scheme, but also as a specific independent genetic algorithm. Why?

\(^4\)However, normally time matters, and it is possible that the solution will not be found soon enough. See chapter 5 for details on FUSS problems.

\(^5\)In section 3.2 we see how real world is a little different.
Figure 2.2.: Effects of proportionate, truncation, tournament, uniform and FUSS selection on the fitness distribution in a generation based EA. The left/right diagrams depict fitness distributions before/after applying the selection schemes depicted in the middle column.
The answer is that FUSS introduces some concepts which distinguish its algorithm from every other standard EAs. FUSS is not simply a common selection scheme which can be used interchangeably with others schemes inside an incremental GA. To usefully work, some changes must be made to the normal evolutionary algorithm execution. Apart from the typical selection function, the use of FUSS implies the following adaptation to the usual algorithm:

**growing population (no replacement):** every standard genetic algorithm normally works with a population of fixed size. So, if we insert some new genomes into the population, some others must be removed (replaced) from it. The idea to let the population grow as much as it can is relatively new for a GA. Although there are some algorithms where the population may grow or shrink depending on the circumstances, usually their size cannot become arbitrarily large (see [9]. See also [2] for a rare example of a variable population size). This is due to the fact that standard operators normally vary their effect depending on the actual population size. Parameters for mutation and recombination operators which are effective with a given population may be useless with smaller populations. In standard GAs, the population size is not different from any other parameter; they must all be adapted to each other with a fine-tuning work [9]. On the contrary, FUSS selection criteria do not depend on the population size, which may grow as much as needed. The only limit is the memory of the machine on which the simulation runs. Even when there is no more memory at disposal, some individuals may be removed without reducing the effectiveness of the algorithm (see section 3.2).

**mutation probability ≥ 1:** in standard evolutionary algorithms the mutation probability is usually ≪ 1 [10]. In these algorithms, a child perfectly identical to its parent is not useless, since it contributes to increase the number of individuals with that fitness value (which, hopefully, is a good one) in the population. That will influence the behaviour of the GA, changing the ratio of \( \frac{\text{good}}{\text{bad}} \) individuals and, therefore, the future selections.\(^7\) With FUSS, however, such a child will bring nothing new to the population. Since FUSS selection cares only about the different fitness values present in the population, a fitness level containing one genome is not different from one containing two copies of the same individual. This selection scheme should help finding new fitness levels, not simply new genomes (solutions).

For this reason all new individuals created by this algorithm should mutate at least once, unless they are created via a crossover.

The last point shows what is also a weak spot of FUSS: all individuals with the same fitness are treated by the selection function as equivalent genomes. However, except for the simplest functions, this is not always convenient. Consider the realistic problem of a simulation, running on a computer, which tries to solve a difficult problem. When the free memory of the machine is exhausted, some individuals of the population must be removed to permit insertion of the new genomes. With FUSS, the most direct way of doing that is to erase individuals from fitness levels where more than one element are present, so that all fitness values are still represented in population. But what if a removed genome was a very good one, maybe only

\(^{6}\)OK, a probability, because of its definition, cannot be > 1. Here by "prob(mut) ≥ 1" it is meant that the new genome could be mutated more than once at every mutation step.

\(^{7}\)Alternatively, if the replacement strategy is to replace the parent of the new genome, giving the child a chance of not being mutated is a way to ensure a genome the possibility to pass to the next generation. It is easy to see the benefit of such a strategy in GAs which replace the entire population at every generation.
"one mutation away" from the global optimum solution? From the algorithm point of view, it was perfectly equivalent to the other genomes with that fitness, which perhaps lead to no good solutions. Obviously, there is no way of always telling for sure which genome will lead to optimum solutions and which will not. This shows that in a realistic case, when the ideal condition of an infinitely growing population is dropped, even the FUSS algorithm may fail in preserving a convenient genetic variety.

2.4. The Scale Independent Fitness Selection (SIF)

**Purpose of SIF**

Observing how FUSS works, one could think that its pressure towards a more uniformly distributed fitness may not be enough to let good individuals emerge from the population. Moreover, when a little bit of greedy search techniques are required to find the optimal solution of a problem, FUSS will show a bad performance, since it never favours best fitness values as such (unless, of course, they are rare in the population, what is normally true at the beginning of a simulation but false after many generations). We will see in section 4.2.5 how this problem appears in real cases.

To partially solve this problem, one could change the way by which the new desired fitness is selected by the algorithm. E.g., instead of choosing a uniformly distributed random number in \([\text{fitness}_{\min}, \text{fitness}_{\max}]\), one could choose it in \([\text{fitness}_{\min}, 2 \cdot \text{fitness}_{\max}]\), assuming that a maximization problem must be solved. This way, half of the times the genome with the largest fitness is selected and can reproduce. Anyhow this arises another question: is it reasonable that the best genome is selected at least half of the times, while the second-best has no privileges and is selected as often as the worst one?

Marcus Hutter found an efficient way of solving this problem, creating a reasonable compromise between the FUSS selection scheme and the greedy approach to a problem. Its name is SIF (Scale Independent Fitness Selection). It was also implemented and its performance was compared to FUSS performance (chapter 4).

**Description of SIF**

SIF is a variation of FUSS; it works exactly the same way FUSS works, changing only the function which selects the fitness. Assuming maximization as optimization direction, SIF gives a high selection likelihood to fitness values close to \(\text{fitness}_{\max}\), and decreasingly lower likelihoods to smaller fitness. The function which gives the selection probability \(p(f)\) for a given fitness \(f\) is:

\[
p(f) = \frac{c}{\ln(|F|)} \cdot \frac{1}{\varepsilon \cdot |\text{fitness}_{\max} - f| + 1}
\]

where \(|F|\) is the number of fitness levels which exists between \(\text{fitness}_{\min}\) and \(\text{fitness}_{\max}\). \(c\) is a constant and \(\varepsilon\) is the interval between a fitness level and the following one. E.g. if fitness is expressed with integer numbers, \(\varepsilon = 1\); on the other hand, if fitness is expressed with real numbers, ideally \(\varepsilon \to 0\). We can consider the set of fitness levels as the set \(F = \{\text{fitness}_{\min}, \text{fitness}_{\min} + \varepsilon, \text{fitness}_{\min} + 2 \cdot \varepsilon, \ldots, \text{fitness}_{\max}\}.\)
The constant factor $\frac{c}{|F|}$ ensures a correct normalization of the probabilities, that means that $\sum f p(f) = 1$. It can be shown that $c \to 1$ for $|F| \to \infty$. For any $|F|$ big enough, it is safe to assume $c = 1$, with little approximation for every practical problem.

The +1 in the denominator has been added to regularize the expression for $f = \text{fitness}_{\text{max}}$.

With such a function, when $f \equiv \text{fitness}_{\text{max}}$ we have roughly $p(f) \equiv 1$, while $p(f) \equiv \frac{1}{|F|}$ otherwise.

If instead of maximizing we need to solve a minimization problem, we can adapt the SIF probability distribution simply by replacing in the formula $|\text{fitness}_{\text{max}} - f|$ with $|\text{fitness}_{\text{min}} - f|$.

As it can be seen from the expression above, the selection probability $p(f)$ for a given fitness level $f$ is not constant throughout the entire simulation, but changes accordingly to the changes of $\text{fitness}_{\text{max}}$ (or $\text{fitness}_{\text{min}}$ if a minimization problem is handled) and the increase of the number of fitness levels $|F|$.

Using SIF instead of FUSS the population should converge towards a situation where the fitness is distributed according to the function $p(f)$ (with $\text{fitness}_{\text{max}}$ or $\text{fitness}_{\text{min}}$ equal to the desired optimum fitness). Better fitness are therefore favoured, but the population will still not converge completely to the optimum value.
3. Details on implementation

This chapter will superficially explain some details of the implementation of the different algorithms. They are necessary to understand the problems encountered during the various simulations performed. In this chapter we will also highlight the differences between the theoretical definition of FUSS and its practical implementation.

To inspect the details of the implementation code, see the appendix.

3.1. The GALIB genetic library

The implementation code should give a flexible basis to test and compare on several problems FUSS, SIF and an incremental algorithm using standard selection schemes. At the beginning of this project a choice must be made between two alternatives: either program the entire code from scratch, or use a genetic library to implement the new algorithms. After having examined several freely available libraries, we found one that seemed perfect for our purposes: GALib [16]. So it was decided to use it, but that decision showed later to be a mistake.

GALib is a rather complete C++ genetic library which seemed to offer all what it was needed to perform the desired experiments. Many different genetic algorithms were implemented, including the incremental one. It had built-in selection schemes, replacement and termination functions, genomes with ready-to-use crossover and mutation operators. Moreover, it promised to be easy customizable; new genetic algorithms, selection schemes and genomes could have been easily added. It also offered some problem examples, which could be used to test our algorithms.

Adding FUSS to GALib as a new genetic algorithm class seemed a simple task which could avoid much work and a loss of time.

3.2. Implementation of FUSS

The implementation of the FUSS algorithm showed to be harder than expected. This was due to the particularities of FUSS which make it different from any other GA (e.g. standard algorithms implemented in GALib used only fixed population size).

The unrealistic assumption of an infinitely growing population must be obviously modified for the practical implementation of FUSS, given that no computing machine can have an infinite memory. A maximum for the population size was used instead; the maximum value could be changed for every computation.\(^1\) During simulations, when the population size reached that maximum an individual was deleted at every generation, before inserting the newly created genome into the population. So a common replacement, like in other genetic algorithms, took place.

\(^1\)Alternatively, one could set a maximum number of individuals for every fitness level.
However, theoretically FUSS does not remove individuals from the population, in order to avoid discarding genetic material. How can it deletes a genome without breaking that premise? The best choice is probably to remove an individuals from the most occupied fitness level, which must contain more than one genome. That ensures that no fitness level is "lost" during computation. After a level has been occupied, the number of individuals in it may vary but never drop to zero again.

In the program, the number of elements in each fitness level must be recorded, in order to always identify the most occupied level when needed for replacement. So the fitness levels must first be identified. That was straightforward when the fitness function was discrete; every possible discrete fitness value represented a level. When the fitness was continuous, however, that was not possible and some discretization must be applied. In that case the minimum and the maximum possible fitness ($f_{\text{min}}$ and $f_{\text{max}}$) were identified. These were chosen so that every feasible fitness for the given problem was contained in the interval $[f_{\text{min}}, f_{\text{max}}]$.

After that a value $\varepsilon$ was determined so that the feasible fitness values could be discretized as $\text{Fitness} = \{f_{\text{min}}, f_{\text{min}} + \varepsilon, f_{\text{min}} + 2 \cdot \varepsilon, \ldots, f_{\text{max}}\}$. Every interval $[f_{\text{min}}, f_{\text{min}} + \varepsilon, f_{\text{min}} + 2 \cdot \varepsilon, \ldots, f_{\text{max}} - \varepsilon, f_{\text{max}}]$ represented a fitness level.

In the simulations reported in chapter 4 the value $\varepsilon$ was always chosen so that all levels could be simultaneously occupied by 2-3 individuals. This means that the maximum population size was always two-three times the number of feasible fitness levels.

3.3. Implementation of SIF

SIF implementation posed practically the same problems of FUSS.

With SIF the set of fitness levels used by the replacement function was also used for computing the selection likelihood $p(f)$ (see section 2.4). That means that the $\varepsilon$ present in the previous section was the same $\varepsilon$ used in the formula

$$p(f) = \frac{1}{\ln(|P|)} \cdot \frac{1}{\varepsilon \cdot |\text{fitness}_{\max} - f| + 1}$$

That was not the best possible implementation for SIF. It worked well only for discrete fitness functions. The problem is due to the fact that in a population where the fitness is distributed according to the function $p(f)$ many individuals possess a good fitness value, while only few genomes have a bad fitness. If the fitness levels have a fixed range $\varepsilon$, the level corresponding to the best fitness will contain much more individuals than the worst fitness level. All these fit individuals would be considered equivalent from the algorithm point of view, which is not good when a level contains in fact many different fitness values. For such a situation (SIF selection and continuous fitness function) it would be better to use a time dependent level range $\hat{\varepsilon}$ defined as the average fitness distance between neighboring individuals. That means, when $f^t_{\text{max}}$ and $f^t_{\text{min}}$ are respectively the maximum and minimum fitness in a population of size $|P|$ at time $t$:

$$\hat{\varepsilon} = \frac{f^t_{\text{max}} - f^t_{\text{min}}}{|P| - 1}$$

Genome selection would then better be carried out by taking a uniform random fitness $f$ from the interval $[f^t_{\text{min}} - \frac{1}{2} \hat{\varepsilon}, f^t_{\text{max}} + \frac{1}{2} \hat{\varepsilon}]$ and selecting the individual of the population with the fitness nearest to $f$. [1].

18
3.4. Implementation problems

After having implemented the algorithms, simulations were started to test them. Soon it became clear that the used library had a great weakness: it was very inefficient. The experimental simulations ran very slowly. A closer look to the library code explained that problem: GALib used an internal sorted list to keep track of the individuals in the population. The list was sorted according to the fitness. When an individual replaced another one in the population, it simply took the place of the deleted genome in the list, possibly making the sort invalid. Therefore the entire population was re-sorted after every replacement. This is efficient only when all genomes are replaced at every generation. If only one genome is inserted (and one deleted), as in our cases, it would be far better to insert it directly in the right list position.

To avoid this problem, the genetic algorithms must be re-implemented so that they no longer used the internal sorted list to keep track of the individuals.²

Another problem could not be avoided and influenced the computation time during simulations. GALib linearly searched the entire population at every generation to record the statistical values of the population (maximum fitness, minimum fitness, etc.). That was also very inefficient when only one genome is added at every step. These statistical values where in any case needed by our algorithms. Since there was no way of recording them more efficiently without rewriting half of the library, this slow-down must be accepted.

Because of the inefficiency of the GALib implementation, our simulations were quite slow. The computation time for the updating of the statistic values was proportional to the population size, which is far from the optimum when only one or two genomes are changed at every generation (it could be done in constant time). That prevented us from performing long computations, which would have needed too much time. Moreover, the maximum population size for the FUSS algorithm was never set too high, since it influenced the simulation time.

After having seen these problems, unfortunately, it was not possible to simply abandon GALib and implement FUSS and the other algorithms without programming libraries. That would have meant to loose even more time in programming and debugging, postponing again the simulation beginning.

²This is the mean of the prefix "Fast" in the name of many genetic algorithm classes used during the simulations; these classes were optimized to avoid that problem (see appendix).
4. Inspected problems

Various test problems were implemented to investigate the behaviour of the FUSS algorithm, using the software described in chapter 3. In current chapter these problems are described. They were chosen among others as benchmarks which could test the particularities of FUSS and SIF algorithms.

The performances shown by these two algorithms are compared with the performance of an incremental GA, given equivalent parameter settings. This is an obvious choice, since incremental algorithm is the most similar to FUSS. In addition to that, using its usual selection schemes it is subject to the problem of getting trapped in local optima, what FUSS should avoid. So a comparison of the two could show how FUSS profits of this advantage.

4.1. A simple example problem

The first problem implemented was a simple one, created "ad hoc" to demonstrate that FUSS can indeed be superior to the standard incremental genetic algorithm. The problem is theoretically analyzed by Marcus Hutter in [1]. The simulations made had the purpose to confirm the theoretical results obtained and to verify the correctness of the implementation.

4.1.1. 2-dimensional example

Description

Consider the case of individuals which are coordinates in the unit square $I$. Each genome can be expressed as a tuple of real numbers $(x, y) \in I = [0, 1] \times [0, 1]$.

Two random numbers $a$ and $b$ are defined together with a value $\Delta$, with $\Delta \ll 1$ and $a, b \in [0, 1 - \Delta]$.

Two "features" are then defined that may be "possessed" by any individual: $I_1$ and $I_2$.

- Individual $i$ possesses feature $I_1$ if $i \in [a, a + \Delta] \times [0, 1]$.
- Individual $i$ possesses feature $I_2$ if $i \in [0, 1] \times [b, b + \Delta]$.

A fitness function $f : I \rightarrow \{1, 2, 3, 4\}$ is defined as follows.

- Individuals possessing feature $I_2$ but not $I_1$ have fitness 1.
- Individuals possessing feature $I_1$ but not $I_2$ have fitness 2.
- Individuals possessing no feature have fitness 3.
- Individuals possessing both features have fitness 4.
\[ f(x, y) = \begin{cases} 
1 & \text{if } (x, y) \in I_2 \setminus I_1 \\
2 & \text{if } (x, y) \in I_1 \setminus I_2 \\
3 & \text{if } (x, y) \notin I_1 \cup I_2 \\
4 & \text{if } (x, y) \in I_1 \cap I_2 
\end{cases} \]

Figure 4.1: The 2-dimensional simple example problem

Given that the fitness must be maximized, it is clear from the definition that for an individual is better to have no features rather than only one of them (fitness 3 instead of 1 or 2). But possessing both of them gives the maximum possible fitness (4). With \( \Delta \ll 1 \), only a small fraction of the unit square represents a fitness different from 3, which is a local maximum. Creating a population of individuals at random will generate many genomes with fitness 3 (approximately a fraction \( (1 - \Delta)^2 \)), some with fitness 1 or 2 (about \( 2 \cdot \Delta \)) and very few, maybe none, with fitness 4 (a fraction \( \Delta^2 \)).

Assume further that the used mutation operator modifies only one coordinate of an individual (it doesn't matter how), while leaving the other unchanged.

It should now be obvious why such a problem favours the FUSS approach, rather than the usual way of solving optimization problems that incremental GA uses [1]. If no optimum solution has been found yet, an individual could reach the optimum fitness 4 with one mutation only if it previously lied in a region of fitness 1 or 2. However, common selection schemes tend to favour, during the selection phase, genomes with better fitness values, here those possessing fitness 3. Selections of unfit individuals are rare, decreasing the probability of a winning mutation which produces an optimum genome.

With FUSS, on the contrary, the algorithm selects and mutates two thirds of the times individuals belonging to any fitness level lower than the local optimum [1].

**Implementation**

After the problem was implemented, simulations were run trying to solve it with FUSS, an incremental GA and a random search. In random search, a new random individual is simply created at every generation and added to the population, replacing another random selected genome. It was used simply to show that the incremental algorithm does not score substantially better than it on this problem.

For every solution scheme 50 simulations were repeated, recording the number of generations\(^1\) needed to find the global maximum of the function. The average number of generations needed was then computed.

The parameters used for the simulation are given in table 4.1.\(^2\) For a description of the roulette selection scheme see [8].

The mutation operator used worked by setting one of the two coordinates of the genome

\(^1\)A generation is defined as one cycle selection-mutation-insertion in population. At every generation only one genome is added to (and one possibly removed from) the population. Here a generation increment does not imply the substitution of all individuals, as in other evolutionary algorithms.

\(^2\)Maybe a population size of 300 individuals was too big for the incremental algorithm; it could determine a slower convergence rate. However, a precise fine tuning of the algorithm parameters is a difficult task [9]. The aim was to furnish both to FUSS and to incremental GA similar conditions, not to give the best possible parameter values. The same approach to the problems was used all throughout this work.
(randomly chosen) to a random number uniformly distributed between 0 and 1.\textsuperscript{3} Mutation probability was 1.

Crossover, not always used, took the first coordinate of one parent genome and the second coordinate of the other parent genome to construct a new individual. Crossover probability was 0.5 (crossover occurred in the average half of the times). Random search could also employ crossover to create new individuals, instead of always produce random genomes. To not make this operation senseless, a population size $> 1$ was used also for this algorithm.

The replacement strategy for the incremental algorithm was “parent”: after creating a new genome, its parent was removed from population.

### Results

The average number of generations needed to find a solution depends on the value of $\Delta$. Therefore simulations were run using different $\Delta$s; afterwards, the empirical function for the number of generations needed was interpolated from the experimental results (see table 4.2).

#### Table 4.2.: Expected number of generations needed to find a solution to the 2-dimensional example problem

<table>
<thead>
<tr>
<th>algorithm</th>
<th>theoretical value</th>
<th>experimental value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUSS without crossover</td>
<td>$\sim \frac{1}{\Delta}$</td>
<td>$\sim 2.9996 \cdot \left(\frac{1}{\Delta}\right)^{1.066}$</td>
</tr>
<tr>
<td>FUSS with crossover</td>
<td>$\sim \frac{1}{\Delta}$</td>
<td>$\sim 3.2027 \cdot \left(\frac{1}{\Delta}\right)^{1.066}$</td>
</tr>
<tr>
<td>incremental without crossover</td>
<td>$\sim \frac{1}{\Delta^2}$</td>
<td>$\sim 10.27 \cdot \left(\frac{1}{\Delta}\right)^{1.5339}$</td>
</tr>
<tr>
<td>incremental with crossover</td>
<td>$\sim \frac{1}{\Delta^2}$</td>
<td>$\sim 12.56 \cdot \left(\frac{1}{\Delta}\right)^{1.4605}$</td>
</tr>
<tr>
<td>random search without crossover</td>
<td>$\frac{1}{\Delta^2}$</td>
<td>$\sim 0.8628 \cdot \left(\frac{1}{\Delta}\right)^{2.03}$</td>
</tr>
<tr>
<td>random search with crossover</td>
<td>$\frac{1}{\Delta^2}$</td>
<td>$\sim 1.2478 \cdot \left(\frac{1}{\Delta}\right)^{2.09}$</td>
</tr>
</tbody>
</table>

The theoretical values were confirmed by the practical simulations, stating clearly the superiority of FUSS on this kind of problem (at least in comparison to incremental GA). This is more evident when looking at the graphs produced by these computations (graphs 4.2 and 4.3; dotted lines are the interpolated functions).

The incremental algorithm scored in fact better than predicted, being anyway far from reaching FUSS performance. Why did it get such good results? Probably because the advantage of starting with 300 different individuals was considerable, when relatively few gen-

\textsuperscript{3}Bit-flipping mutation was also tried, but was less effective.
Figure 4.2.: Bilogarithmic graph showing the average number of generations needed to find an optimum solution for the 2-dimensional example problem, depending on the value of $\frac{1}{\Delta}$.

Figure 4.3.: Bilogarithmic graph showing the average number of generations needed to find an optimum solution for the 2-dimensional example problem with crossover, depending on the value of $\frac{1}{\Delta}$.
erations were needed to solve the problem. So if we consider only the more difficult simulations, with smaller \( \Delta \)s, for which more generations are needed and therefore the starting advantage is negligible, better interpolations should be obtained. In fact, employing only the results for \( \Delta \leq 0.0125 \), the interpolated function for the simulation without crossover is 
\[ \sim 1.1404 \cdot \left( \frac{1}{\Delta} \right)^{1.9759} \]
For the case when crossover is used, using the last values we can interpolate the function 
\[ \sim 3.7957 \cdot \left( \frac{1}{\Delta} \right)^{1.7265} \]
better than the previous one. With more efficient simulation programs, it would be possible to perform much longer computations, possibly obtaining more precise results.

4.1.2. 3-dimensional example

Description

The example problem described in section 4.1.1 can quite easily be extended to more than 2 dimensions. As usual, a value \( \Delta \ll 1 \) and \( D \) random values \( a_1, a_2, a_3, \ldots, a_d \) are defined. A generalized \( D \)-dimensional fitness function for a \( D \)-dimensional individual \( x \) can then be expressed as:

\[
f(x) = [(D + 1) \cdot \prod_{d=1}^{D} \mathcal{X}_d(x)] - \left[ \max_{1 \leq d \leq D} \mathcal{X}_d(x) \right] + D + 1
\]

where \( \mathcal{X}_d(x) \) is the characteristic function of feature \( I_d \) defined as:

\[ \mathcal{X}_d(x) = \begin{cases} 
1 & \text{if } a_i \leq x_i \leq a_i + \Delta \\
0 & \text{else}
\end{cases} \]

For \( D = 2 \), \( f \) coincides with the fitness function given in section 4.1.1. For \( D = 3 \) the following situation exists:

\[
f(x) = \begin{cases} 
1 & \text{if } x \in I_3 \setminus (I_1 \cap I_2) \\
2 & \text{if } x \in I_2 \setminus I_3 \\
3 & \text{if } x \in I_1 \setminus (I_2 \cup I_3) \\
4 & \text{if } x \notin (I_1 \cup I_2 \cup I_3) \\
5 & \text{if } x \in (I_1 \cap I_2 \cap I_3)
\end{cases}
\]

It is worth noting that increasing \( D \) the problem really becomes more and more difficult, because the ratio \( \frac{\text{maximum fitness region}}{\text{D-dimensional unit cube}} \) decreases rapidly (following \( \frac{1}{\Delta^D} \)). So for \( D = 3 \) the fractions of \([0,1]^3\) where \( f = 1/2/3/4/5 \) are approximately \( \Delta^2/\Delta^2/\Delta^3/1/\Delta^3 \).

Implementation

The problem was implemented only for \( D = 3 \)

Its implementation was practically identical to the 2-dimensional version. The same parameters were used, with the exception of the mutation probability. Due to the implementation of the mutation function, mutation likelihood could not be exactly 1. For the simulations a probability of 0.9 was therefore chosen.

The mutation operator changed a single coordinate of the genome, setting it to a random number uniformly distributed between 0 and 1. The other coordinates were not touched.

Crossover created new genomes by randomly selecting every coordinate from one of the two parent individuals. Contrary to the 2-dimensional problem, a genome could thus take all its genetic material from the same parent, becoming a perfect copy of it.
Results

Apart from the incremental algorithm, the results obtained from the simulations reflected the theory, like in the preceding case. However, the precision was not as good as with \( D = 2 \), especially on the FUSS algorithm. This was perhaps due to the insufficient mutation probability. Most probably, the simulations performed were too few to obtain precise results, but the implementation problems pointed out in chapter 3 were a serious obstacle against longer computations.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>theoretical value</th>
<th>experimental value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUSS without crossover</td>
<td>( \sim \frac{1}{\Delta} )</td>
<td>( \sim 99.378 \cdot (\frac{1}{\Delta})^{1.515} )</td>
</tr>
<tr>
<td>FUSS with crossover</td>
<td>( \sim \frac{1}{\Delta} )</td>
<td>( \sim 159.13 \cdot (\frac{1}{\Delta})^{0.884} )</td>
</tr>
<tr>
<td>incremental without crossover</td>
<td>( \sim \frac{1}{\Delta} )</td>
<td>( \sim 355.21 \cdot (\frac{1}{\Delta})^{1.9637} )</td>
</tr>
<tr>
<td>incremental with crossover</td>
<td>( \sim \frac{1}{\Delta} )</td>
<td>( \sim 521.52 \cdot (\frac{1}{\Delta})^{1.4879} )</td>
</tr>
<tr>
<td>random search without crossover</td>
<td>( \sim \frac{1}{\Delta} )</td>
<td>( \sim 1.2215 \cdot (\frac{1}{\Delta})^{2.99} )</td>
</tr>
<tr>
<td>random search with crossover</td>
<td>( \sim \frac{1}{\Delta} )</td>
<td>( \sim 1.8579 \cdot (\frac{1}{\Delta})^{2.99} )</td>
</tr>
</tbody>
</table>

The incremental GA scored surprisingly good on the problem. Although obtaining the worst results for every \( \Delta \) considered, it scaled much better than a random search. It must be pointed out that the number of values (corresponding to simulations) used for interpolating the results was smaller than for the FUSS algorithm. For FUSS, simulations were run for 14 different \( \Delta \) values. For the other two algorithms, due to the long computation time, fewer \( \Delta \) values were used. Perhaps the values obtained were not statistically significant; again, the long computation times were an obstacle against more precise simulations.

However, the simulations confirmed that FUSS is better than an incremental algorithm for this problem. A pure random search was the best choice for relatively big values of \( \Delta \), but it scaled very badly (figure 4.4). Another interesting theoretical results confirmed by these measures was the fact that, contrary to the 2-dimensional case, here the use of crossover can improve FUSS performance of a factor \( \sim \frac{1}{\Delta} \) (figure 4.5).

4.1.3. Continuous fitness

Description

The continuous variants of the discrete fitness functions used in the preceding simulations were developed. These were:

\[
2D : \quad f(x + a, y + b) = 2 + \frac{x^2}{x + \Delta x} + \frac{y^2}{y + \Delta y} + 4 \cdot \frac{e^{-(x^2+y^2)}}{\Delta^2}
\]

\[
3D : \quad f(x + a, y + b, z + c) = 2 + \frac{x^2}{x + \Delta x} + 2 \cdot \frac{y^2}{y + \Delta y} + 3 \cdot \frac{z^2}{z + \Delta z} - 1 \cdot \frac{x^2+y^2}{x+y+\Delta x} - 1 \cdot \frac{x^2+z^2}{x+z+\Delta x} - 2 \cdot \frac{y^2+z^2}{y+z+\Delta y} + 3 \cdot \frac{e^{-(x^2+y^2+z^2)}}{\Delta^2}
\]

26
Figure 4.4.: Bilogarithmic graph showing the average number of generations needed to find an optimum solution for the 3-dimensional example problem without crossover, depending on the value of $\frac{1}{\Delta}$.

Figure 4.5.: Bilogarithmic graph showing the average number of generations needed to find an optimum solution for the 3-dimensional example problem with crossover, depending on the value of $\frac{1}{\Delta}$. For FUSS, also the result of the simulation without crossover is depicted in order to allow a comparison.
Here $x, y$ and $\varepsilon$ represent the difference between the coordinates of the individual and the "goal values" $a, b$ and $c$.

Simulations were made using FUSS to maximize these functions. The expected results should mirror the results obtained with the discrete fitness functions, at least with small $\Delta$ values. See figure 4.6 for a view of the 2-dimensional continuous function. Judging from the function graph, it can be deduced that the continuous function mirrors the discrete one better when $\Delta$ is small. For all $\Delta$ values, however, the continuous problem will probably be a more difficult task for FUSS than the discrete one, especially in 3 dimensions. The reason is that much more fitness levels are involved and the "optimum region" of the unit square is smaller than in the discrete case (see next subsection for more details).

![Figure 4.6: Continuous fitness function for the 2-dimensional simple problem, with $\Delta$ set to 0.1 (left) and 0.01 (right) respectively. Goal values are $a = b = 0.5$.](image)

**Implementation**

The parameters used in the implementation were quite identical to the discrete case. Two of them must in any case be changed: maximum population size and desired optimum value.

In the discrete case there were only four possible fitness for the 2-dimensional problem: 1, 2, 3 and 4. On the contrary, using continuous values all floating points numbers between 1 and 4 were potential fitness. For this reason the maximum population size for FUSS algorithm was set to 1000 individuals, so that the population could better scatter over all fitness levels.

The desired optimum value was used to stop the computation: when an individual with the desired fitness was created, the simulation stopped. It was clearly quite impossible to reach exactly the optimum fitness with the continuous function, so the desired value could not be set to 4 (2D), respectively 5 (3D). In the discrete case, an individual was considered an optimum solution and stopped the algorithm when it was inside the maximum fitness region. In 2D that meant, for any individual $i$:

$$\forall i = (x, y) : \text{if } (x, y) \in [a, a + \Delta] \times [b, b + \Delta] \Rightarrow f(i) \geq 4 \Rightarrow \text{stop}$$

To do something analogous in the continuous variant, we should use $[a - \frac{1}{2} \Delta, a + \frac{1}{2} \Delta] \times [b - \frac{1}{2} \Delta, b + \frac{1}{2} \Delta]$ instead of $[a, a + \Delta] \times [b, b + \Delta]$ and find a suitable smaller value $\varepsilon$ to replace 4. Unfortunately, this was not possible. Such a $\varepsilon$ would have been smaller than the local
maximum 3. E.g. if $i = (a + \frac{1}{2}\Delta, b + \frac{1}{2}\Delta) \Rightarrow f(i) < 3$. Using a $c$ fulfilling that condition would have caused the algorithm to stop even for genomes completely outside the optimum region, but with $\text{fitness} \approxeq \text{local maximum}$.

The optimum condition should therefore be restricted. We stated that an individual having any coordinate equal to the desired value would have stopped the algorithm, if the difference between the other coordinate and the other desired value was less than $\frac{1}{2}\Delta$. Differently said, a $c$ value was chosen so that:

$$\forall i = (x, y) : if \; \begin{cases} x = a \land y \in [b - \frac{1}{2}\Delta, b + \frac{1}{2}\Delta] \\ or \\ x \in [a - \frac{1}{2}\Delta, a + \frac{1}{2}\Delta] \land y = b \end{cases} \Rightarrow f(i) \geq c \Rightarrow \text{stop}$$

The 3D variant was analogous.

Suitable values for $c$ which fulfilled the above requirement for every $\Delta$ were found. The desired fitness values were $c = 3.3$ for the 2-dimensional and $c = 4.1$ for the 3-dimensional case.

This choice for the desired optimum fitness should quite accurately mirror the discrete case. Nevertheless the continuous problem was more difficult than the discrete one. With the discrete 2-dimensional fitness function, the optimum fitness region was a $\Delta \times \Delta$ square. With the continuous function the region was roughly restricted to a circle with diameter $\sim \Delta$.\(^4\)

Obviously, decreasing or increasing the $c$ parameter would influence the entire computation, making it easier or more difficult for FUSS to find a solution. However, quite good results were obtained with the given values.

**Results**

The expected outcomes were confirmed by the experiments. Table 4.4 shows the expected number of generations needed to find a suitable solution for the continuous problem as it was interpolated from the simulation results. A comparison to the discrete case is given.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>D</th>
<th>experimental value (discr.)</th>
<th>experimental value (cont.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUSS without crossover</td>
<td>2</td>
<td>$\sim 2.9996 \cdot \left(\frac{1}{\Delta}\right)^{1.066}$</td>
<td>$\sim 6.3555 \cdot \left(\frac{1}{\Delta}\right)^{1.1577}$</td>
</tr>
<tr>
<td>FUSS with crossover</td>
<td>2</td>
<td>$\sim 3.2027 \cdot \left(\frac{1}{\Delta}\right)^{1.066}$</td>
<td>$\sim 5.6602 \cdot \left(\frac{1}{\Delta}\right)^{1.166}$</td>
</tr>
<tr>
<td>FUSS without crossover</td>
<td>3</td>
<td>$\sim 99.378 \cdot \left(\frac{1}{\Delta}\right)^{1.515}$</td>
<td>$\sim 424.41 \cdot \left(\frac{1}{\Delta}\right)^{1.6801}$</td>
</tr>
<tr>
<td>FUSS with crossover</td>
<td>3</td>
<td>$\sim 159.13 \cdot \left(\frac{1}{\Delta}\right)^{0.884}$</td>
<td>$\sim 318.57 \cdot \left(\frac{1}{\Delta}\right)^{1.7374}$</td>
</tr>
<tr>
<td>incr. without crossover</td>
<td>2</td>
<td>$\sim 10.27 \cdot \left(\frac{1}{\Delta}\right)^{1.5339}$</td>
<td>$\sim 28.373 \cdot \left(\frac{1}{\Delta}\right)^{1.6093}$</td>
</tr>
<tr>
<td>incr. with crossover</td>
<td>2</td>
<td>$\sim 12.56 \cdot \left(\frac{1}{\Delta}\right)^{1.4605}$</td>
<td>-</td>
</tr>
<tr>
<td>incr. without crossover</td>
<td>3</td>
<td>$\sim 355.21 \cdot \left(\frac{1}{\Delta}\right)^{1.9637}$</td>
<td>$\sim 646.57 \cdot \left(\frac{1}{\Delta}\right)^{2.2014}$</td>
</tr>
<tr>
<td>incr. with crossover</td>
<td>3</td>
<td>$\sim 521.52 \cdot \left(\frac{1}{\Delta}\right)^{1.4879}$</td>
<td>-</td>
</tr>
</tbody>
</table>

\(^4\)Actually, it was more an ellipse.
The continuous function proved to be harder to optimize than the discrete one, as it was expected. The function values obtained confirmed nevertheless the theory assertions. It should be noticed that, in the 3-dimensional continuous problem, crossover does not improve FUSS performance. This is an obvious consequence of the fact that there are far more than five different fitness levels, when the fitness function is continuous, and above all they are distributed in the unit cube in a completely different way. The respective position of the different fitness levels was the key that allowed the discrete 3D problem to take advantage of the crossover operator [1]. Using a continuous fitness function, the fitness levels have other shapes and positions. That is true both for the 2-dimensional and the 3-dimensional case. Compare for example the region with fitness 2 in figure 4.1 with the overview of the region where fitness $\geq 2$ in figure 4.7.

In figure 4.8 a comparison of FUSS and incremental GA with the discrete and the continuous fitness function for 2 dimensions is given (the graph for the simulations employing crossover is similar and therefore it is not depicted).

![Figure 4.7: Region of the unit square in which the fitness is approximately 2 in the continuous 2-dimensional simple problem ($\Delta = 0.1$).](image)

### 4.2. Traveling Salesman Problem

The superiority of FUSS with respect to a standard incremental algorithm has now been proved for at least one problem. However, it is surely more interesting to compare both algorithms on more difficult problems, related to real world tasks.

The Traveling Salesman Problem (TSP) is undoubtedly a good problem example. TSP is difficult (it is an NP problem, see for example [3]) and can be applied to many practical domains such as scheduling, cryptanalysis, molecular biology, Bayesian networks, clustering, etc. For these reasons it is often used as a test bench for algorithms [4].

#### 4.2.1. Problems variants

Basically, the traveling salesman problem is defined as the problem to find the shortest tour (path) which permits to visit a given number of cities, touching them exactly once.
Three variants of this problem were used to compare the performances of FUSS, incremental GA and SIF. After describing each variant, in sections 4.2.2 to 4.2.6 various tests based on these problems are described and their results are given.

**TSP with random distance matrix (DTSP)**

A first way of defining a TSP problem is by giving its *distance matrix*. The $N \times N$ distance matrix of a symmetric TSP problem with $N$ cities is a matrix with

$$a_{ij} = \text{distance between city } i \text{ and city } j, \quad \forall i, j \in \{0, 1, 2, \ldots, N - 1\},$$

where $a_{ij} = a_{ji}$ and $a_{ii} = 0$ for every $i$ and $j$.

In the implementation used in the tests, the distance matrix was randomly created before every simulation with values of $a$ uniformly distributed in $[0, 1]$.

Note that there (probably) were no two identical simulations, since the distance matrix was created every time anew. Moreover, it was not possible to know in advance the exact length of the best TSP solution, since the problem was created by the machine during simulation and its data remained hidden. Anyhow, it was not necessary to know the exact solution, as it is known that for such problems a bound of $\sim 2,0415$ exists for the expected length of an optimal tour, when $N$ is big enough. Most important, the purpose of the simulations was not to find solutions for the different TSP instances, but mostly to compare the performances of different evolutionary algorithms. That latter task can be accomplished without knowing any solution at all.

**Random Euclidean TSP (ETSP)**

A TSP can also be created by randomly distributing $N$ cities in a unit square. (Differently said, $N$ cities are created by randomly selecting $N$ coordinates $(x, y) \in [0, 1] \times [0, 1]$). Two
cities can have the same coordinates.

The distance between two cities is then given by the euclidean distance of their coordinates. The ETSP is normally a problem which is intrinsically less difficult to solve than the DTSP [15].

Even for this implementation the TSP instances remained hidden in the computer simulation, but as already said that was not important for our purposes.

**Note:** The euclidean TSP can be constructed both in 2 dimensions and in 3 dimensions. The results given later concerns only the 2-dimensional case, even though also the other problem was implemented; there simply was no time to thoroughly examine it.

**Real TSP instances**

Some real TSP instances were implemented and used to test the algorithms. Speaking of real instances, we mean both TSP examples created from real cities and little TSPs created by hand, useful for examining FUSS behaviour on small problems.

Contrary to the preceding variants, here the optimum solution for every TSP instance was known. Some tests which were not feasible with DTSP and ETSP were thus possible. In any case, the inefficiency of the library implementation prevented us from testing FUSS, SIF and the incremental algorithm on instances larger than a few dozen cities.

Source for (partially solved) real instances of TSP problems were [6] and [7].

Now a description of the simulated test problems is given, along with their results. For all tests, the problem of minimizing the path length was transformed into a maximization problem by using the following fitness function:

\[ fitness(i) = \frac{c}{l(i)} \]

where \( c \) is a constant, \( i \) is a path (an individual) and \( l(i) \) is its length. In the following simulations \( c = 200 \). That fitness function had the advantage of compressing the large, uninteresting region of long paths to smaller fitness range.

**4.2.2. First test: generations needed for solution**

**Description**

The first test aimed to show which algorithm converged faster towards the solution. Since the best tour for the given instance could not be known, only the number of generations\(^5\) needed to reach a given fitness was computed. The goal fitness was set relatively low (33.33, which meant a path length of \( \frac{200}{33.33} \approx 6 \); approximately 300% of the optimum path length for DTSPs). This also made simulations much faster, a very useful characteristic when using an inefficient, slow implementation.

**Implementation**

Simulations were made using FUSS and incremental GA with the parameters shown in table 4.5. Both the euclidean TSP and the distant matrix TSP variants were computed, with and

---

\(^5\) Remember that one generation corresponds to an individual insertion (and deletion, for incremental GA).
without crossover. Since the time needed to reach a given fitness depends on the number \( N \) of cities in the TSP instance, the problem was simulated for various \( N \in [15, 29] \). The number of cities was chosen so that the problems were big enough to be interesting without needing too much computation time to be solved. For each test problem, 10 simulations were run and the average number of generations needed was recorded. For the incremental algorithm two different selection schemes were implemented: roulette selection and tournament selection [8].

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Selection Scheme</th>
<th>Initial Pop. Size</th>
<th>Maximal Pop. Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUSS</td>
<td>FUSS</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>Incremental</td>
<td>Roulette</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>Tournament</td>
<td></td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>

The mutation operator changed the position of a city in the path. The city to move and its new position were determined at random. Mutation probability was set as close as possible to 1.

The crossover function was a partial match crossover (see [8] for an exhaustive explanation of the partial match algorithm). Crossover probability was 0.2 (in average, a crossover occurred every five generations).

Both operators were not the best operators that can be implemented for a TSP problem. But since the purpose was to compare the two algorithms and not to solve the problem as fast as possible, that did not matter.

The replacement strategy used for the incremental algorithm when solving TSP instances removed the worst genome from the population.

**Note:** the incremental evolutionary algorithm had an advantage, starting with 500 randomly created individuals instead of 1. It will be seen that this advantage could not prevent the algorithm from being outperformed by FUSS.

**Results**

Contrary to the simple example problem, here the use of crossover seemed to never improve the algorithm performances. This could be expected, since primitive crossover operators are not effective with TSP problems. To obtain good results, one should use operators specifically designed for that problem [4].

Tournament selection proved to be more suitable for the problem than roulette selection, but both were no match for FUSS. The obtained results are shown in figures 4.9, 4.10, 4.11 and 4.12.

For small \( N \) (say \( N < 18 \)), in the distant matrix TSP, the incremental algorithm often reached the desired fitness already at the beginning. That meant that at least one of the 500 individuals randomly created for the initial population had a path length \( l(i) < 6 \). With \( N > 18 \) or solving ETSP that never happened. However, we can not tell that incremental GA is better than FUSS for these small problems, since starting with 500 different genomes was clearly a big advantage.
Figure 4.9.: Graph showing the average number of generations needed to reach a given path length in a DTSP of $N$ cities.

Figure 4.10.: Graph showing the average number of generations needed to reach a given path length in a DTSP of $N$ cities, using crossover.
Figure 4.11.: Graph showing the average number of generations needed to reach a given path length in a ETSP of $N$ cities.

Figure 4.12.: Graph showing the average number of generations needed to reach a given path length in a ETSP of $N$ cities, using crossover.
4.2.3. Comparison of FUSS and SIF on the first test

Description

The same test described in section 4.2.2 was computed using SIF, in order to compare its performance with the performance of FUSS.

Implementation

The parameters used were the same already given in the preceding section. As before, simulations were made with DTSP and ETSP, both with and without crossover.

Results

A comparison of the simulation results is given in figures 4.13, 4.14, 4.15 and 4.16.

It is interesting to note that SIF seems to have worse performances than FUSS. FUSS therefore appears to converge faster than SIF in TSP problems. Remembering the fact that SIF should favour individuals with a better fitness, this fact is quite surprising. In any case, FUSS superiority is not clear: e.g. see figures 4.13 and 4.15.

4.2.4. Second test: fitness after fixed generations

Description

The second test was similar to the first one. Instead of using a fixed fitness value and compute the number of generations needed to reach it, a fixed generation number was set and the maximal fitness obtained at that generation was recorded.

Implementation

FUSS and incremental algorithm using roulette and tournament selection were tested, as usual with and without crossover.

Simulations were made using DTSP and ETSP problems of \( N \in [10, 20, 30, \ldots, 100] \) cities. For every \( N \), the simulation was stopped after \( N^2 \) generations and the best (maximum) fitness was recorded. That was repeated 10 times and the average fitness value was computed.

The same parameters and operators of the first test were used.

Results

The results mirrored the outcomes of the first test. However, for \( N = 10 \) the incremental algorithm reached quite always a better fitness than FUSS. That was not significant, due to the fact that incremental started with 500 different individuals instead of 1. Besides, already with \( N = 20 \) FUSS was clearly better, despite of its initial disadvantage.

The figures 4.17, 4.18, 4.19 and 4.20 depict the results of the experiments. The graphs show the minimum path length reached instead of the maximum fitness, in order to make the difference between roulette and tournament selection more visible.\(^6\)

\(^6\)As before, the relation between the path length \( l(i) \) and the fitness \( f(i) \) of an individual \( i \) was \( f(i) = \frac{200}{l(i)} \).
Figure 4.13.: Graph showing the average number of generations needed to reach a given path length in a DTSP of $N$ cities.

Figure 4.14.: Graph showing the average number of generations needed to reach a given path length in a DTSP of $N$ cities, using crossover.
Figure 4.15.: Graph showing the average number of generations needed to reach a given path length in a ETSP of $N$ cities.

Figure 4.16.: Graph showing the average number of generations needed to reach a given path length in a ETSP of $N$ cities, using crossover.
Figure 4.17.: Graph showing the average minimum path length reached after $N^2$ generations in a DTSP of $N$ cities.

Figure 4.18.: Graph showing the average minimum path length reached after $N^2$ generations in a DTSP of $N$ cities, using crossover.
average path length

Figure 4.19.: Graph showing the average minimum path length reached after \( N^2 \) generations in a ETSP of \( N \) cities.

Figure 4.20.: Graph showing the average minimum path length reached after \( N^2 \) generations in a ETSP of \( N \) cities, using crossover.
4.2.5. Third test: solution for real problems

Description

The positive aspect of solving real TSP problems is the fact that normally an optimal solution is already known. So it is possible to observe if an algorithm is really able to find the exact solution of a problem, and not only to approximate it.

FUSS was applied to a real problem taken from [7]. The points set of the problem represented 29 cities in the western Sahara region. The city set and the minimum length path are depicted in figure 4.21. FUSS found the optimum solution in 70789 generations. The needed number of generations was not important, however, because the goal of the simulation was only to show that FUSS could indeed find exact solutions for TSPs.\(^7\)

![Figure 4.21: Travelling Salesman Problem for 29 cities in western Sahara: city set and minimum length path.](image)

After that, several small TSP instances were solved using FUSS. These were problems created by hand with regular city structures and \(N \in [6,20]\). E.g. the problem for \(N = 10\) was represented by a \(2 \times 5\) grid of cities, each having distance 1 from its neighbours. Solving the instances, something strange occurred; sometimes FUSS reached a nearly optimum solution in few generations, without being able to then reach the optimum which was only "one mutation away". A typical situation for that problem is depicted in figure 4.22.

This problem was doubtless due to the absence in FUSS of a selection pressure towards better fitness. In a similar situation, an algorithm employing a biased, standard selection scheme would quickly reach the global optimum, since it would often select the best individual for reproduction. SIF should partially avoid this problem.

It must also be stated that TSP instances with an irregular structure do not cause that problem. It depends on the fact that with a regular disposition of the cities (e.g. a \(4 \times 5\) grid) many different solutions have the same path length and therefore the same fitness. Some fitness levels are then heavily occupied, while many others can not be occupied by any solution since there exists no corresponding path length in the graph. This situation is not favourable for FUSS, which works by "trying to fill" all different fitness levels. On the contrary, when

\(^7\)As a comparison, an incremental algorithm could not find a solution in more than 15000 generations. The computation was then stopped.
Figure 4.22.: A typical problem when using FUSS to solve TSP regular instances. FUSS normally finds in a few generations the nearly optimum solution on the left. It then fails in quickly finding the optimum path on the right, although a single mutation will be enough.

the cities are well scattered without order, many different path lengths are possible and FUSS strategy can obtain the best results.

To compare the performance of FUSS, SIF and incremental GA for these small regular TSPs, simulations were made setting a reasonable maximum number of generations allowed. If an algorithm did not find the exact solution of the TSP before the given generation, the simulation was stopped. The average number of generations needed for finding a solution was recorded (when a solution was found before the generation limit) along with the number of failures of the algorithms (when the result was not found before the limit).

Implementation

The algorithms were applied to 12 TSPs with $6 \leq N \leq 20$, with the same parameters and operators of the preceding tests. Beginning with 500 different genomes would give a too big advantage to the incremental algorithm, given the small size of the problems. For this reason the initial individuals of the population were created all equal.

A limit of 10000 generations was set for the simulation. Every computation was repeated 50 times and the average number of generations needed was recorded. The mean was computed without including the 10000 generations needed when the algorithm failed in finding the solution. If an algorithm could never find the solution before the generation limit, a standard value of 10000 was set for the generation average.

Results

The outcomes of the experiments are given in table 4.6 and depicted in figures 4.23 and 4.24. How can they be interpreted?

- First, it is clear from graph in figure 4.24 that if FUSS and SIF do not get stuck and find a solution, they find it more rapidly than an incremental algorithm.

- Second, it appears that SIF does not improve FUSS a lot. Figure 4.24 shows that FUSS normally converges sooner than SIF to the solution. On the other hand, the latter timed
out quite as often as the first.\footnote{Actually, if we interpolate a function on the points of figure 4.23, we obtain for SIF a function which is slightly better than a similar function interpolated for FUSS, but only for small $N$ values. The improvement, however, is not very significant.}

- Third, the graph in figure 4.24 suggests that both FUSS and SIF had more difficulties with the same instances; their graphs follow approximately the same pattern. This ought to confirm the fact that the graph structure has a big influence on the algorithms effectiveness. E.g. the problem with 20 cities appears to be easier to solve than the instance with 18 cities, despite of the larger size. On the other hand, FUSS and SIF seem to have different probabilities of getting stuck on the same instance (figure 4.23). Consider the 16 cities problem: FUSS could not solve the TSP on time 27 times. This is \emph{three times} the "failures" of SIF (9). But when FUSS indeed solved it, it did that in almost half the generations that SIF needed (in average 1980.17 and 3427.76 generations respectively). So FUSS was normally faster for that instance, but had a larger likelihood of getting stuck than SIF.

The failure likelihood depended perhaps more on the problem structure and the algorithm used than on the instance size. It is worth noting that the structure of the TSP instances appeared to influence differently the difficulty of finding a solution and the probability of getting stuck by solving the problem.

- Fourth and last, figure 4.23 shows that incremental GA can nevertheless be better for small TSP instances. Consider the $N = 12$ case: 11 times FUSS could not find a solution before the 10000th generation. SIF could not do it 6 times. For the same problem size, the incremental algorithm employing tournament selection always found a solution before the generation limit.

However, nobody needs a genetic algorithm to find a solution for such small TSP instances. The problem showed by FUSS could be nevertheless important, since small TSPs can appear as local clusters in a bigger TSP.

### 4.2.6. Fourth test: comparison of fitness evolution

**Description**

This was perhaps the most interesting test performed for comparing the various algorithms. The goal was to analyse how the best fitness in the population progressed along the computation. By comparing the fitness evolution for the different algorithms during the simulation, it was possible to get an idea on how the algorithms worked. The strong and weak points of FUSS and SIF should then become clear.

Fitness was computed with the usual formula $f(i) = \frac{200}{i/10}$.

**Implementation**

FUSS, SIF and the incremental genetic algorithm were applied to six different TSP problems: distant matrix TSP with 20, 50 and 100 cities and euclidean TSP with 20, 50 and 100 cities. Roulette selection and tournament selection were used for the incremental algorithm. Simulation were made with and without crossover, as usual.
Table 4.6.: For different TSP instances of size N, the table gives the average number of generations needed to solve them and the number of times that an algorithm could not solve the problem before the 10000th generation (time outs)-over 50 simulations.

<table>
<thead>
<tr>
<th>N</th>
<th>FUSS</th>
<th>SIF</th>
<th>inc. (tour.)</th>
<th>inc. (roul.)</th>
<th>FUSS</th>
<th>SIF</th>
<th>inc. (tour.)</th>
<th>inc. (roul.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>46.48</td>
<td>62.32</td>
<td>168</td>
<td>135.26</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>91.68</td>
<td>111.02</td>
<td>567.18</td>
<td>750.88</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>64.94</td>
<td>113.72</td>
<td>984.98</td>
<td>2422.7</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>264.36</td>
<td>315.08</td>
<td>2110.3</td>
<td>4821.61</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>48</td>
<td>253.28</td>
<td>248.66</td>
<td>2723.86</td>
<td>6092</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>742.02</td>
<td>750.4</td>
<td>4362.98</td>
<td>10000</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>6</td>
<td>0</td>
<td>50</td>
<td>1388.75</td>
<td>2294.95</td>
<td>5155.18</td>
<td>10000</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>12</td>
<td>9</td>
<td>50</td>
<td>1660.38</td>
<td>2666.84</td>
<td>6821.32</td>
<td>10000</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>11</td>
<td>25</td>
<td>50</td>
<td>2286.7</td>
<td>4778.57</td>
<td>7924.28</td>
<td>10000</td>
</tr>
<tr>
<td>16</td>
<td>27</td>
<td>9</td>
<td>39</td>
<td>50</td>
<td>1980.17</td>
<td>3427.76</td>
<td>8952.82</td>
<td>10000</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
<td>36</td>
<td>50</td>
<td>50</td>
<td>4378.4</td>
<td>7185.93</td>
<td>10000</td>
<td>10000</td>
</tr>
<tr>
<td>20</td>
<td>32</td>
<td>26</td>
<td>50</td>
<td>50</td>
<td>2139.56</td>
<td>4897.33</td>
<td>10000</td>
<td>10000</td>
</tr>
</tbody>
</table>

Figure 4.23.: Graph showing how many times each algorithm could not reach an exact solution before the 10000th generation for a TSP with N cities.
During simulations, every 50 generations the best actual fitness of the population was recorded. After 50000 generations the computation was stopped and the fitness values were plotted to see the fitness evolution during the problem solving.

The parameters used were the same of the previous problems (table 4.7). Mutator and crossover operators were the same used in the first test. Mutation probability was set to 1, crossover probability to 0.2.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>selection scheme</th>
<th>initial pop. size</th>
<th>maximal pop. size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUSS</td>
<td>FUSS</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>SIF</td>
<td>SIF</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>incremental</td>
<td>roulette</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>tournament</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>

**Results**

The results of the simulations are depicted in figure 4.25 to 4.30. The graphs concerning the simulations with crossover are not shown, since they generally mirrors those without crossover.

Looking at these figures, many interesting facts can be noticed:

- In TSP, both FUSS and SIF converge much faster than incremental GA towards better fitness values. This result was already seen in connection to the first and the second test. It is a somewhat surprising result, given that FUSS apparently use no selective pressure towards better fitness.
Figure 4.25.: Graph showing the evolution of the best fitness in population for a DTSP problem with 20 cities

Figure 4.26.: Graph showing the evolution of the best fitness in population for an ETSP problem with 20 cities
Figure 4.27.: Graph showing the evolution of the best fitness in population for a DTSP problem with 50 cities

Figure 4.28.: Graph showing the evolution of the best fitness in population for an ETSP problem with 50 cities
Figure 4.29.: Graph showing the evolution of the best fitness in population for a DTSP problem with 100 cities

Figure 4.30.: Graph showing the evolution of the best fitness in population for an ETSP problem with 100 cities
• Moreover, sometimes FUSS converges faster than SIF, even if the latter has a selection biased towards better fitness values.

• Nevertheless, after a rapid convergence phase both algorithms seem to reach a "plateau", finding no better solutions more, even though they have probably not attained the optimum yet.9

• For small TSP instances (here, 20 cities), incremental algorithm may find the (probably) optimal solution even faster than FUSS and SIF; e.g. see graph 4.25. This seems to happen since the latter two converge very rapidly at the beginning of the computation, when few fitness levels are occupied, and very slowly afterwards, when quite all possible fitness have been "found". On the contrary the incremental GA has a smaller but more constant convergence rate.

After viewing the depicted graphs, some questions arose:

1. When FUSS and SIF do not improve their solution for many generations, like FUSS in figure 4.27, are they hopelessly stuck, or is it possible for them to find better solutions later?

2. Will incremental algorithm finally approach the solution found by FUSS and SIF even for TSP bigger than 20 cities, when a sufficient number of generations is considered? That would mean that FUSS and SIF maybe can not be a better choice than incremental algorithm, when looking for an exact solution.

To possibly reply to these questions, the only way was to let the simulation run for far more than 50000 generations. The simulation was done setting 400000 generations as computation limit. The maximum population size was 1000 for the incremental algorithm and 3000 for FUSS and SIF. Both a DTSP and an ETSP instance with 50 cities were solved while recording the maximum population fitness every 100 generations. Results are shown in figures 4.31 and 4.32.

The outcomes of the experiments confirmed that the answer to both previous questions is "yes".

1. Look at graph in figure 4.31: FUSS kept a fixed maximum fitness value for a little bit less than 300000 generations before performing a new improvement. In the other simulation, SIF increased its best fitness again after having been stuck for approximately 200000 generations. So it may be said that with these algorithms you must never despair of improving your solution.

2. On the other hand, at least for the distant matrix TSP the incremental GA eventually attained a maximum fitness better than that of FUSS and SIF. This means that although FUSS and SIF never stop improving, they could do it too slowly when getting close to the solution. A standard incremental algorithm may beat them, when looking for an exact solution and not only for an approximation.

9In the simulation depicted in figure 4.27, for example, FUSS stopped improving after having reached a maximum fitness equal to 444247. That corresponds to a path of length \( \frac{330}{100} \times \frac{4818}{100} \), which is unlikely to be a minimum length path for the problem.
Figure 4.31.: Graph showing the evolution of the best fitness in population for a DTSP problem with 50 cities

Figure 4.32.: Graph showing the evolution of the best fitness in population for an ETSP problem with 50 cities
To check if the outcomes of the latter experiments were unusual or were on average for such problems, the simulations were repeated three times (not more, because of the mentioned long computation times needed) and the results averaged. The values obtained are depicted in figures 4.33 and 4.34. They mirror quite precisely the two preceding graphs, with the exception of the respective performance of SIF and incremental GA in the distant matrix problem. Judging from the graph in figure 4.33, the incremental algorithm normally does not reach the fitness levels attained by SIF before the 4000000th generation. The outcome of our first simulation was an odd case. Nevertheless, it is likely to do that sooner or later. To be convinced of such an assertion, it would probably be enough to take a look at the same graph displayed with a bilogarithmic scale (figure 4.35).
Figure 4.33.: Graph showing the average evolution of the best fitness in population for 3 DTSP problems with 50 cities

Figure 4.34.: Graph showing the average evolution of the best fitness in population for 3 ETSP problems with 50 cities
Figure 4.35: Bilogarithmic graph showing the average evolution of the best fitness in population for 3 DTSP problems with 50 cities

Figure 4.36: Bilogarithmic graph showing the average evolution of the best fitness in population for 3 ETSP problems with 50 cities
5. Discussion of the results

The basic idea behind FUSS and SIF was to develop a selection scheme for genetic algorithms which could avoid the problem known as premature convergence by preventing an algorithm from getting stuck in local optima. The experiments performed showed other interesting features of FUSS and SIF but also some problems common to both algorithms.

In the following sections we will summarize the strong and the weak spots of the algorithms highlighted by the simulations, along with the characteristics which differentiates them.

5.1. Strengths of FUSS and SIF

- The most important positive aspect of FUSS and SIF is surely their impressive initial convergence rate. At least for the Travelling Salesman Problem they clearly outperform an incremental algorithm employing the common selection schemes for many generations. Both algorithms converge fast towards better fitness when the density of fit individuals in the population is low, a fact which is typically true for the initial population of standard EAs. Thus SIF and FUSS converge well at the beginning of the simulation, when standard algorithms (often) do not.

- Standard GAs requires a fine tuning of the simulation parameters (population size, mutation and crossover probability, etc.) in order to achieve a good convergence, not too quick nor too slow, on a given problem. On the contrary, FUSS and SIF tend to automatically create a suitable selection pressure. The mutation likelihood can simply be set to a high value (even 1), while the population size should grow as much as possible. The only parameter that may need an adjustment is the crossover likelihood. Two values concerning the practical implementation must nevertheless be carefully chosen:

  - If an implementation of FUSS or SIF employs a maximum size value to determine when the population should stop growing, this parameter ought to be selected so that the population could occupy all fitness levels. The maximum number of individuals should be larger than the number of feasible fitness levels.

  - Furthermore, if the fitness function for a problem is continuous, a practical implementation may discretize the fitness interval to obtain a finite number of fitness levels. If a parameter $\varepsilon$ is chosen, like in our implementation, to define the range of every level, then it should be selected as small as possible. A large $\varepsilon$, in fact, would prevent the algorithm from identifying fitness improvements in the population.

- At least for the studied TSPs, FUSS and SIF showed that they could suddenly increase their best fitness even after having shown no improvement for many generations. This is probably a confirm that both algorithms fulfill the desired criterion of never get hopelessly stuck in local optimum wells.
5.2. Weaknesses of FUSS and SIF

- The two inspected algorithms also showed some defects. First of all, their convergence rate decreases rapidly when the individuals are finally well scattered over many fitness levels. Intuitively, when the population density in the levels is homogeneous, there is no selection pressure towards any fitness. This situation persists until a new fitness level is "discovered" by means of a mutation or a crossover. Since it becomes increasingly difficult to find new fitness values, fitness improvements become fewer and fewer.

- Because of the slow convergence when approaching the optimal solution, probably FUSS and SIF are not suitable for exactly solving large, difficult problems. A simple incremental algorithm may even beat them on the long run when solving a TSP.

- It was observed in section 4.2.5 that SIF and FUSS risk to get stuck (but not forever) when solving TSPs with a regular city structure. It seems that for these algorithms the effectiveness highly depends on the problem structure. That was not the case for the incremental GA. It may be that FUSS and SIF show this tendency even with other regularly structured problems.

5.3. Comparison of FUSS and SIF

Here some differences in the behaviour of the two inspected algorithms are given.

- In the TSP problems described in section 4.2.6, FUSS seemed to converge faster than SIF. On the other hand, SIF usually reached a plateau in the fitness values later than FUSS, and finally attained better solutions. It could be said that SIF’s behaviour is halfway between the FUSS and the incremental GA behaviour, probably not only for the traveling salesman problem. This would be consistent with the fact that the algorithm acts like FUSS, but also applying some selection pressure towards better fitness values, like the incremental algorithm does.

- Contrary to what was expected, SIF did not appear to avoid the problem of getting stuck when solving regular TSP instances. It simply got stuck with different problem structures (figure 4.23). Before solving a regular problem with one of these algorithms, it would be therefore better to check which one is more suitable for the given problem structure.
6. Conclusions and future directions

6.1. Conclusions

In this work we examined the performance of a newly developed selection scheme, the Fitness Uniform Selection Scheme, along with a variant of it, the Scale Independent Fitness Selection. We showed how FUSS implies some radical changes in the usual way of working with genetic algorithms.

An implementation of FUSS and SIF algorithms was made in C++ language using an existing genetic algorithm library, GALib. The use of the library however proved later to be a mistake, since it strongly affected the efficiency of the computations.

The first simulations performed aimed to confirm the theoretical behaviour of FUSS for a constructed sample problem. The experiments had some surprises in store, but generally gave the expected results. Also a standard incremental genetic algorithm was applied to the problem. A comparison of the performances of the two algorithms proved that FUSS could indeed be superior to an incremental GA for at least one problem.

Then we used FUSS and SIF to solve many variants of the well known Travelling Salesman Problem, often used as a test bench for optimization algorithms. Despite the fact that the simulation possibilities were restricted by the inefficiency of the implementation, various interesting results were obtained. FUSS and SIF were compared to an incremental GA on these problems. They proved to outperform it at least in convergence time. Although they apparently do not favour better genomes in the population, it seemed that in TSP FUSS and SIF converge much more rapidly than an incremental algorithm which uses biased selection schemes. FUSS was born as an algorithm that tried to avoid premature convergence; it may seem ironic that one of its features appears to be a high convergence speed.

FUSS only had problems in finding exact solutions for some regular TSP instances. In situations where an algorithm employing a strong selection pressure towards better fitness values could easily find a solution, FUSS could on the contrary get stuck for long. In addition to that, FUSS and SIF may converge too slowly after a lot of generations, when quite all fitness levels are uniformly "occupied" by individuals.

FUSS, SIF and the incremental algorithm were easily comparable since they worked in a similar manner. The number of generations needed to solve a given problem was used as a measure of the respective performances. However, to compare these algorithms with other solution methods, like ant algorithms for TSPs, the only suitable unit of measure is the computation time. Since the library used for our implementation was very inefficient, in our case such comparisons did not make any sense and were not done.
6.2. Future directions

The many simulations made with FUSS and SIF surely do not clarify all aspects of these algorithms. Some questions were answered, but many more arose. This work is only a starting point in the study of the potential of these genetic algorithms. Even the examination of the performance of FUSS and SIF on the Travelling Salesman Problem may certainly be extended. The algorithms could be applied to big TSP instances and compared to other known algorithms for that kind of problems (ant algorithms, tabu search, ...). It should be tested if the good initial convergence speed of the two algorithms is not restricted to the Travelling Salesman Problem.

The behaviour of SIF and FUSS during the TSP simulations was slightly different. FUSS converged faster, while SIF did it normally for longer time. It would be interesting to compare the effectiveness of these algorithms also on other test benches, to possibly identify the sorts of problem for which SIF is more suitable than FUSS and vice versa.

The inspected algorithms showed a tendency to get stuck when solving traveling salesman instances with a highly regular structure. It would be of interest to inspect other problems with regular structure, in order to know if that was a special case or FUSS and SIF are intrinsically unable to reliably solve regularly structured problems.

An algorithm based on FUSS or SIF will probably show the best performance when used only for approaching optimal solutions. Since many real problems concern the approximation of optimal solutions, these algorithms may find some useful applications. If, however, some problem requires an exact solution, FUSS may be used to approach it. The task of finding that optimum solution could then be carried out by a common local search algorithm employing a selection pressure towards better fitness, which could start from where FUSS stopped. This combinations should unify the best aspects of both kinds of algorithm.

Listing all problems to which FUSS and SIF could be usefully applied is simply impossible; every difficult multimodal problem is a good candidate for testing their performance. They should especially be suitable for solving problems where the risk of entrapment in local optima is extremely high for standard evolutionary algorithms.
Acknowledgments

I would like to thank the following persons:

• Marcus Hutter, who not only suggested me the topic of this work, but also constantly supported and guided me during the last four months.

• Prof. Petros Koumoutsakos, for having accepted to be my supervisor professor and therefore having made possible for me to work on a diploma project with such an interesting topic.

• My family that financially supported me during these years at the ETHZ, allowing me to accomplish my computer science studies. I owe them all my education.

• All people at the IDSIA, that made my working period there a pleasure. I really enjoyed their company.

• Sheila Arrigo, for her linguistic and logistic support.
Bibliography


A. Program Code

For documentation purposes and for future reference, the used C++ code is listed in this chapter. Not all the C++ code produced during this work is listed here, but only the programs which may help understanding the details of the simulations performed. Therefore only the genetic algorithms classes are listed, which were a tiny fraction of the total code. The simulating program and the various problem classes are not given. The uninteresting parts of the code have been removed, so if you find a call to a procedure which is not listed in the appendix, do not worry: it is simply not needed for understanding the algorithms.

All genetic algorithm classes present in this appendix were subclasses of the "GALevelSpecies" class contained in the GALib library. That class is not listed here, because it is not essential for understanding these classes. The header file (.h) with the class declarations is omitted.

The class functions given are practically two: a step() function and a local implementation of a selection scheme. The step() function was called at every generation and should manage the selection of the parent genomes, the use of the mutation and crossover operators and the insertion/replacement of the new genome. It was the heart of the genetic algorithm classes.

The GALib genetic library was required in order to let the programs run. For details on GALib, see [16].

Note: the FUS algorithm code listed in this appendix is the one used for TSP simulations. The algorithm used for solving the simple example problem (section 4.1) was slightly different and is not shown here.
A.1. Genetic algorithms

A.1.1. Fitness Uniform Selection Scheme (FUSS) code

```cpp
/*
 * FastFitIntervalFUSSGA.cpp
 * E:\GATest\FastFitIntervalFUSSGA.cpp
 * Daniele Pongi, 17.12.2001
 *
 * DESCRIPTION:
 * Class which implements a genetic algorithm that uses FUSS algorithm. This means that
 * the algorithm acts as an incremental one, adding only one new genome at every
 * generation, with one difference: no replacement takes place until a given maximum
 * population size has been reached. New individuals are simply inserted into the
 * population, which in turn grows steadily.
 * Mutation is applied only at the selected element.
 * This algorithm class inherits from class FitIntervalFUSSGA. The difference between the
 * two algorithms lies in the fact that this one is optimized, so as to minimize the
 * computation time. In particular, a special data structure is used to keep track of
 * every genome (the popMiroirs[] array). This allows to avoid, for example, the resorting
 * of the entire population after every insertion or replacement. This is an unnecessary
 * expensive operation, when considering that only one genome is inserted/replaced at
 * every generation.
 * The price to pay for this is that, due to the implementation features of the GALib
 * library, it’s impossible to use a selection function other that the one here
 * implemented. The reason for this is that sorting must be made on the popMiroirs[] array,
 * rather than on the effective population.
 * In addition to that, population based evaluators don’t work.
 *
 * last updated: 8.1.2002
 */

#include "fast_fit_interval_fussga.h"

void FastFitIntervalFUSSGA::step() {
    int mut, c;
    int sel = 1;
    GAGenome *mom, *dad; // temporary holders for selected genomes
    mom = &localFUSSSelect(); // select parent
    c = 0;
    // either do crossover or copy the parent genome
    if (GAFlyCoin(pCrossover())) {
        // do crossover
        dad = &localFUSSSelect(); // select parent
        sel++;
        stats.numCross += (*cross) (*mom, *dad, child1, (GAGenome*)0);
        c = 1;
    } else {
        // copy parent genome
        child1->copy(*mom); // copy parent
    }
```
stats.numSel += sel;
mut = child1->mutate(pMutation()); // mutate genome according to prob.
if (mut > 0) {
    stats.numMut += mut;
    c = 1;
}
if (c == 1) {
    // if genome derives from a crossover or has been mutated, it must be evaluated
    child1->evaluate(gaFalse);
}
stats.numEval += c;
// replacement takes place only if the size of the population has grown till maxPopSize
if (pop->size() >= maxPopSize) {
    dad = getGenomeInLevel(mostOccupiedLevel); // choose a genome to replace
    if (dad != NULL) {
        pop->remove(child1, dad); // remove child from population
        decMostOccLevel(); // decrement number of genomes in the level where we took dad
        pop->scale(gaFalse); // scale the population to have child's fitness
        incLevelByFit(child1->fitness()); // increment n. of genomes in the level of child
        replaceInPopMirror(child1, dad); // replace in internal genome list
        child1 = dad; // need to do this, or next step() will mess all up
        stats.numRep += 1;
    }
} else {
    pop->add(child1); // add child to population
    pop->scale(gaFalse); // scale the population to have child1's fitness
    incLevelByFit(child1->fitness()); // increment n. of genomes in the level of child1
    addInPopMirror(child1); // add in internal genome list
    child1 = child1->clone(); // need to do this, or next step() will mess all up
}
stats.update(*pop); // update the statistics for this generation

// the following function selects a genome with the FUSS selection scheme
GA* Genome & FastFitIntervalFUSSGASelect() {
    int tempIndex, lastIndex;
    float tmpMaxFit, tmpMinFit;
    // select the desired fitness uniformly in the interval [fitmin, fitmax]
    float desiredFitness = (*fitSelectFunction)(pop->fitMin(), pop->fitMax());
    int desiredLevel = getLevelByFit(desiredFitness);
    GA* genome = getGenomeInLevel(desiredLevel);
    if (genome != NULL) {
        return genome;
    }
    else {
        // if no genomes exists in the desired fitness level, the internal genome list
        // (popMirror) must be searched to find the genome with the fitness closest to
        // the desired one
        int preceedLevel, nextLevel;
        GA* preceedGen, *nextGen;
        // find a preceding level containing genomes
        preceedLevel = desiredLevel - 1;
        while ((preceedLevel >= 0) && (genInLevel[preceedLevel] == 0)) {
            preceedLevel--;
        }
        //...
// find genome with greatest fitness in this level
if (precedLevel >= 0) {
    tempIndex = firstOccIndex[precedLevel];
    precedGen = popMirror[tempIndex];
    tmpMaxFit = precedGen->fitness();
    lastIndex = tempIndex + genNum[precedLevel];
    for (tempIndex++; tempIndex < lastIndex; tempIndex++) {
        if ((popMirror[tempIndex]->fitness() > tmpMaxFit) {
            precedGen = popMirror[tempIndex];
            tmpMaxFit = precedGen->fitness();
        }
    }
}
// find a following level containing genomes
nextLevel = desiredLevel + 1;
while ((nextLevel < levelNumber) & & (genNum[nextLevel] == 0)) {
    nextLevel++;
}
// find genome with smallest fitness in this level
if (nextLevel < levelNumber) {
    tempIndex = firstOccIndex[nextLevel];
    nextGen = popMirror[tempIndex];
    tmpMinFit = nextGen->fitness();
    lastIndex = tempIndex + genNum[nextLevel];
    for (tempIndex++; tempIndex < lastIndex; tempIndex++) {
        if ((popMirror[tempIndex]->fitness() < tmpMinFit) {
            nextGen = popMirror[tempIndex];
            tmpMinFit = nextGen->fitness();
        }
    }
}
if (precedLevel < 0) {
    return *nextGen;
} else if (nextLevel >= levelNumber) {
    return *precedGen;
}
// return genome with fitness closest to desiredFitness
float distPreced = (float) fabs(tmpMaxFit - desiredFitness);
float distNext = (float) fabs(tmpMinFit - desiredFitness);
return (distPreced < distNext) ? *nextGen : *precedGen;
A.1.2. Scale Independent Fitness (SIF) code

/*
FastFitIntervalSIFGA.cpp
E:\GATest\FastFitIntervalSIFGA.cpp
Daniele Pongan, 20.2.2002

DESCRIPTION:
Class which implements a genetic algorithm that uses FUSS algorithm with a scale
independent selection.
This algorithm class inherits from class FastFitIntervalFUSSGA. The difference lies in
the fact that here the probability to choose the new individual from a given fitness
level is not equal for every level. Level with higher fitness are more probably
chosen, so that individuals with greater fitness are favoured. (only maximization
problems are allowed at the moment).

last updated: 20.2.2002
*/

// the step function of this genetic algorithm must insert a new individual into the
// population at every generation. The new individual must be a copy of a selected one.
// None of the already present individuals is deleted. So no replacement takes place,
// and the population must grow of one unit at every generation (until a maximum is
// reached).
// When the population maximum is reached, replacement takes place at every generation.
// The genome to be deleted from population is chosen between those belonging to the
// most densely populated fitness level.
void FastFitIntervalSIFGA::step() {
    int mut, c;
    int sel = 1;
    GAGenome *mom, *dad; //temporary holders for selected genomes
    mom = &(localSIFSelect()); //select parent
    c = 0;
    // either do crossover or copy the parent genome
    if (GAFlipCoin(pCrossover())) {
        // do crossover
        dad = &(localSIFSelect()); //select parent
        sel++;
        stats.numero += (*cros)(*mom, *dad, child1, (GAGenome*)0);
        c = 1;
    } else {
        // copy parent genome
        child1->copy(*mom);
    }
    stats.nmusel += sel;
    mut = child1->mutate(pMutation()); //mutate genome according to prob.
    if (mut > 0) {
        stats.nmutmut += mut;
        c = 1;
    } if (c == 1) {
        // if genome derives from a crossover or has been mutated, it must be evaluated
        child1->evaluate(gaFalse);
    }
    stats.nmuval += c;
    // replacement takes place only if the size of the population has grown till maxPopSize
    if (pop->size() >= maxPopSize) {
dad = getGenomeInLevel(mostOccupiedLevel); // choose a genome to replace
if (dad != NULL) {
    pop->replace(child1, dad); // put child1 in population, remove dad from it
decMostOccLevel(); // decrement number of genomes in the level where we took dad
pop->scale(faFalse); // scale the population to have child's fitness
incLevelByFit(child1->fitness()); // increment n. of genomes in the level of child1
replaceInPopMirror(child1, dad); // replace in internal genome list
child1 = dad; // need to do this or next step will mess all up
stats.numrep += 1;
}
else {
pop->add(child1); // add child1 to population
pop->scale(faFalse); // scale the population to have child1's fitness
incLevelByFit(child1->fitness()); // increment n. of genomes in the level of child1
addInPopMirror(child1); // add in internal genome list
child1 = child1->clone(); // need to do this, or next step will mess all up
}
stats.update(*pop); // update the statistics for this generation
}

// the following function selects a genome with the SIF selection scheme
GA Genome & FastFitIntervalSIFGA::localSIFSelect() {
    float tmpMinFit = pop->fitMin();
    float tmpMaxFit = pop->fitMax();
    // if maximum or minimum fitness in population is changed, re-compute probabilities
    // for the fitness levels
    if (maximumLevel != getLevelByFit(tmpMinFit)) ||
        (minimumLevel != getLevelByFit(tmpMaxFit)) {
        computeFitnessProb(tmpMinFit, tmpMaxFit);
    }
    // selects a fitness level
double randomNumber = GARandomFloat(0, (float) probSum);
    int desiredLevel = getLevelByFit(tmpMaxFit);
    while ((randomNumber > 0) & (desiredLevel >= minimumLevel)) {
        randomNumber -= fitnessProb[desiredLevel];
        desiredLevel--;
    }
    // get genome in selected fitness level
    GA Genome *genome = getGenomeInLevel(++desiredLevel);
    if (genome != NULL) {
        return *genome;
    }
}
else {
    // if no genomes exists in the desired fitness level, the internal genome list
    // (popMirror) must be searche to find the genome with the fitness closest to
    // the desired one
    int tempIndex, lastIndex;
    int precededLevel, nextLevel;
    GA Genome *precedGen, *nextGen;
    float desiredFitness = minLevelFitness + (desiredLevel * levelRange);
    // find a preceding level containing genomes
    precededLevel = desiredLevel - 1;
    while ((precededLevel >= 0) && (genInFitLevel[precededLevel] != 0)) {
        precededLevel--;
    }
    // find genome with greatest fitness in this level
if ((precedLevel >= 0) {
    tempIndex = firstOccOfLevel[precedLevel];
    precedGen = popMirror[tempIndex];
    tmpMaxFit = precedGen->fitness();
    lastIndex = tempIndex + genInFitLevel[precedLevel];
    for (tempIndex++; tempIndex < lastIndex; tempIndex++) {
        if (((popMirror[tempIndex] )->fitness() > tmpMaxFit) {
            precedGen = popMirror[tempIndex];
            tmpMaxFit = precedGen->fitness();
        }
    }
    // find a following level containing genomes
    nextLevel = desiredLevel + 1;
    while ((nextLevel < levelNumber) && (genInFitLevel[nextLevel] == 0)) {
        nextLevel++;
    }
    // find genome with smallest fitness in this level
    if (nextLevel < levelNumber) {
        tempIndex = firstOccOfLevel[nextLevel];
        nextGen = popMirror[tempIndex];
        tmpMinFit = nextGen->fitness();
        lastIndex = tempIndex + genInFitLevel[nextLevel];
        for (tempIndex++; tempIndex < lastIndex; tempIndex++) {
            if (((popMirror[tempIndex] )->fitness() < tmpMinFit) {
                nextGen = popMirror[tempIndex];
                tmpMinFit = nextGen->fitness();
            }
        }
    }
    if (precedLevel < 0) {
        return *nextGen;
    } else if (nextLevel >= levelNumber) {
        return *precedGen;
    } // return genome with fitness closest to desiredFitness
    float distPreced = (float) fabs(tmpMaxFit - desiredFitness);
    float distNext = (float) fabs(tmpMinFit - desiredFitness);
    return (distPreced > distNext) ? *nextGen : *precedGen;
}

// function that initialise the probabilities for each fitness level
void FastFitIntervalSFGA::initializeFitnessProb() {
    int actualLevel;
    fitnessProb = new double[levelNumber];
    for (actualLevel = 0; actualLevel < levelNumber; actualLevel++) {
        fitnessProb[actualLevel] = 0;
    }
    computeFitnessProb(pop->fitmin(), pop->fitmax());
}

// function to re-compute the fitness level probabilities
void FastFitIntervalSFGA::computeFitnessProb(float minFitness, float maxFitness) {
minimumLevel = getLevelByFit(minFitness);
maximumLevel = getLevelByFit(maxFitness);
int localLevelNumber = maximumLevel - minimumLevel + 1;
probSum = 0;
if (localLevelNumber > 1) {
    float minFitInLevel;
    for (int i = minimumLevel; i <= maximumLevel; i++) {
        minFitInLevel = minLevelFitness + (i * levelRange);
        fitnessProb[i] = 1 /
            (log((double) localLevelNumber)
            * ((1 / levelRange) * (maxFitness - minFitInLevel) + 1));
        probSum += fitnessProb[i];
    }
} else {
    fitnessProb[minimumLevel] = 1;
    probSum = 1;
}
A.1.3. Incremental algorithm with roulette selection code

/*
   FastIncrementalGA.cpp
   E:\GATest\FastIncrementalGA.cpp
   Daniele Pongh, 18.2.2002

DESCRIPTION:
This algorithm class inherits from class GAIncrementalGA. The difference between the
two algorithms lies in the fact that this one is optimized, so as to minimize the
computation time. In particular, a special data structure is used to keep track of
every genome (the popMirror[] array). This allows to avoid, for example, the ressorting
of the entire population after every insertion or replacement. This is an unnecessary
expensive operation, when considering that only one genome is inserted/replaced at
every generation.
The price to pay for this is that, due to the implementation features of the GALib
library, it's impossible to use a selection function other that the one here
implemented. The reason for this is that sorting must be made on the popMirror[] array,
rather than on the effective population.
In addition to that, population based evaluators don't work.

Last updated: 26.2.2002
*/

// the step function of this genetic algorithm must replace a genome of the population
// with a newly created one at every generation
void FastIncrementalGA::step() {
    int mut, c;
    int sel = 1;
    GA::Genome *mom, *dad; // temporary holders for selected genomes
    mom = &(localRouletteSelect()); //select parent
    c = 0;
    // either do crossover or copy the parent genome
    if (GAFlipCoin(pCrossover())) {
        // do crossover
        dad = &(localRouletteSelect()); //select parent
        sel++;
        stats.numcrosso += (*cCross) (*mom, *dad, child1, (GA::Genome*)0);
        c = 1;
    } else {
        // copy parent genome
        child1->copy(*mom);
    }
    stats.numsel += sel;
    mut = child1->mutate(pMutation()); // mutate genome according to prob.
    if (mut > 0) {
        stats.nummut += mut;
        c = 1;
    }
    if (c == 1) {
        // if genome derives from a crossover or has been mutated, it must be evaluated
        child1->evaluate(gaFalse);
    }
    stats.numeval += c;
    // select genome to replace: worst genome will be replaced
    if (this->minimiz() == GA::GeneticAlgorithm::MAXIMIZE) {
        dad = popMirror[0];
    }
}
if (dad != NULL) {
    pop->replace(child1, mom); // put child1 in population, remove dad from it
    pop->scale(gaFalse); // scale the population to have child's fitness
    replaceInPopMirror(child1, mom); // replace in internal genome list
    updateFitnessSum(mom->fitness(), child1->fitness()); // update sum of fitness
    child1 = mom; // need to do this, or next step() will mess all up
    stats.numrep += 1;
}
else {
    GATestError("ERROR WHILE REPLACING GENOME", "Worst genome didn't exist");
    stats.update(*pop); // update the statistics for this generation
}

// roulette selection. Genomes are selected with a probability which is proportional to
// the fitness. It works only if all fitnesses are either positive OR negative.
GAGenome & FastIncrementalGA::localRouletteSelect() {
    int populationSize = pop->size();
    // check that all fitness are positive OR negative
    if ((popMirror[0]->fitness() * popMirror[populationSize - 1]->fitness()) < 0) {
        GATestError("FITNESS VALUE(S) NOT ALLOWED",
            "For this selection function to work, fitnesses must be either all positive"
            "or all negative");
    }
    int i = 0;
    if (popMirror[0]->fitness() < 0) { // negative fitness
        float randomNumber = GARandomFloat(fitnessSum, 0);
        if (this->minmax() == GAGeneticAlgorithm::MINIMIZE) { // maximization problem
            while ((randomNumber < 0) && (i < populationSize)) {
                randomNumber := popMirror[i]->fitness();
                i++;
            }
        }
        else {
            while ((randomNumber < 0) && (i < populationSize)) { // minimization problem
                randomNumber := 1 / popMirror[populationSize - 1 - i]->fitness();
                i++;
            }
            i = populationSize - i;
        }
    }
    else { // positive fitness
        float randomNumber = GARandomFloat(0, fitnessSum);
        if (this->minmax() == GAGeneticAlgorithm::MAXIMIZE) { // maximization problem
            while ((randomNumber > 0) && (i < populationSize)) {
                randomNumber := popMirror[populationSize - 1 - i]->fitness();
                i++;
            }
            i = populationSize - i;
        }
        else {
            while ((randomNumber > 0) && (i < populationSize)) { // minimization problem
                ...
randomNumber = 1 / popMirror[i]->fitness();
i++;
}
}
}
return *popMirror[i];

// function used to update the fitness sum, used for the roulette selection
inline void FastIncrementalGA::updateFitnessSum(float oldFit, float newFit) {
if (this->minimax() == GAGeneticAlgorithm::MAXIMIZE) {
    fitnessSum -= oldFit;
    fitnessSum += newFit;
}
else {
    fitnessSum -= (1 / oldFit);
    fitnessSum += (1 / newFit);
}
}
A.1.4. Incremental algorithm with tournament selection code

/*
 * FastIncrementalTourGA.cpp
 * E:\GTest;FastIncrementalTourGA.cpp
 * Daniele Pongan, 28.2.2002
 *
 * DESCRIPTION:
 * This algorithm class inherits from class GAIncrementalGA. The difference between the
 * two algorithms lies in the fact that this one is optimized, so as to minimize the
 * computation time. In particular, a special data structure is used to keep track of
 * every genome (the popMirror[] array). This allows to avoid, for example, the resorting
 * of the entire population after every insertion or replacement. This is an unnecessary
 * expensive operation, when considering that only one genome is inserted/replaced at
 * every generation.
 * The price to pay for this is that, due to the implementation features of the GALib
 * library, it’s impossible to use a selection function other that the one here
 * implemented. The reason for this is that sorting must be made on the popMirror[] array,
 * rather than on the effective population.
 * In addition to that, population based evaluators don’t work.
 *
 * last updated: 28.2.2002
 */

// the step function of this genetic algorithm must replace a genome of the population
// with a newly created one at every generation
void FastIncrementalTourGA::step() {
    int mut, c;
    int sel = 1;
    GAGenome *mom, *dad; // temporary holders for selected genomes
    mom = &(localTournamentSelect()); // select parent
    c = 0;
    // either do crossover or copy the parent genome
    if (GAFlipCoin(pCrossover())) {
        // do crossover
        dad = &(localTournamentSelect()); // select parent
        sel++;
        stats.numcro += (*scross) (*mom, *dad, (GAGenome*)0);
        c = 1;
    } else {
        // copy parent genome
        child1->copy(*mom);
    }
    stats.numsel += sel;
    mut = child1->mutate(pMutation()); // mutate genome according to prob.
    if (mut > 0) {
        stats.nummut += mut;
        c = 1;
    }
    if (c == 1) {
        // if genome derives from a crossover or has been mutated, it must be evaluated
        child1->evaluate(gaFalse);
    }
    stats.numeval += c;
    // select genome to replace; worst genome will be replaced
    if (this->minimax() == GAGeneticAlgorithm::MAXIMIZE) {
        dad = popMirror[0];
    }
}
else {
    dad = popMirror[populationSize] - 1;
}
}
if (dad != NULL) {
    pop->replace(child1, dad); // put child1 in population, remove dad from it
    pop->scale(gaFalse); // scale the population to have child's fitness replaced
    popMirror(child1, dad); // replace in internal genome list
    child1 = dad; // need to do this, or next step() will mess all up
    stats.numrep += 1;
} else {
    GATestError("ERROR WHILE REPLACING GENOME", "Worst genome didn't exist");
}
stats.update(*pop); // update the statistics for this generation

// tournament selection. Two genomes are randomly selected; then the one with the best
// fitness is returned.
GAGenome & FastIncrementalTourGA::localTournamentSelect() {
    int populationSize = pop->size();
    if (populationSize > 1) {
        int firstIndex = GARandomInt(0, populationSize - 1);
        int secondIndex = GARandomInt(0, populationSize - 1);
        // repeat selection until we have two different genomes
        while (firstIndex == secondIndex) {
            secondIndex = GARandomInt(0, populationSize - 1);
        }
        if (this->minMax == GAGeneticAlgorithm::MAXIMIZE) { // maximization problem
            return (popMirror[firstIndex]->fitness() > popMirror[secondIndex]->fitness()) ?
                *popMirror[firstIndex] : *popMirror[secondIndex];
        } else { // minimization problem
            return (popMirror[firstIndex]->fitness() < popMirror[secondIndex]->fitness()) ?
                *popMirror[firstIndex] : *popMirror[secondIndex];
        }
    }
    else { return *popMirror[0];
    }
A.1.5. Random search algorithm code

```cpp
// A random search algorithm

/*
RandomSearchGACpp
E:\GATest\RandomSearchGACpp
Daniele Pongar, 11.1.2002

DESCRIPTION:
Class which implements a random search. At every step, a new individual is created and
initialized. To handle the case where the initialization function doesn’t create a
random individual, the individual is also mutated a convenient number of time.
No mutation is applied.
This genetic algorithm derives from the SimpleFUSSGA the fact that the new genome is
simply added to the current population, without replacement, until a given population
maximum size has been reached. After that, the new individual replace a genome in the
population selected with the usual method; however, if the population size is 1, the
replacement method doesn’t matter and we have a true random search.

last updated: 11.1.2002

 我 A0+0RPR2: FB2OK,9<;G$,MPR: I I
*/

// the step function of this algorithm insert a new genome into the population without
// removing any individual until a given maximum population size is reached.
// After that, replacement is performed as usual.
void RandomSearchGAC::step() {
    const int initMutNumber = 100;
    int c;
    GAGenome *mom, *dad; // temporary holders for selected genomes
    GAGenome *parent;
    // select two parent genomes
    mom = &(pop->select());
    dad = &(pop->select());
    stats.numSel += 2;
    c = 0;
    if (GAFlipCoin(p->crossover())) { // check if crossover must be done
        stats.numCros += (*cros) (*mom, *dad, child1, (GAGenome*)0);
        parent = mom;
        c = 1;
    } else {
        if (GARandomBit()) { // choose which genome will become parent of child1
            parent = mom;
        } else {
            parent = dad;
        } child1->initialize(); // create new individual
        for (int i = 0; i < initMutNumber; i++) {
            child1->mutate(i);
        }
    }
    stats.numEval += c;
    // replacement takes place only if the size of the population has grown till maxPopSize
    if (pop->size() >= maxPopSize) { // replacement
        if (rs == PARENT) {
            child1 = pop->replace(child1, parent);
        } else if (rs == CUSTOM) {
```
child1 = pop->replace(child1, &ref(*child1, *pop));
}
else {
    child1 = pop->replace(child1, rs);
    stats.numrep ++;
}
else { // no replacement; new individual is simply added to current population
    pop->add(*child1);
}
pop->evaluate(gaTrue); // allow pop-based evaluators to do their thing
stats.update(*pop); // update the statistics for this generation
}