Evaluating and Improving Steady State Evolutionary Algorithms on Constraint Satisfaction Problems

Koen van der Hauw

9 August 1996

Abstract

Currently there is a growing interest in the evolutionary algorithm paradigm, as it promises a robust and general search technique. Still, in spite of much research, for many people the question remains how good evolutionary algorithms really are. Therefore, in this research, a successful class of evolutionary algorithms, Steady State evolutionary algorithms, is thoroughly examined to find optimal settings on two NP-complete problems: Graph 3-Coloring and 3-Satisfiability. Several versions of the evolutionary algorithm are tested and evaluated and the best version for each NP-complete problem is compared to a good existing algorithm for each problem. Then extensions for the evolutionary algorithm are presented that make the evolutionary algorithms perform better than the more traditional algorithms on the hardest problem instances.

Preface

This research was done as a Master's Thesis for graduating in Computer Science at Leiden University. It is about evolutionary algorithms, search algorithms based on Darwin's evolution theory, and was motivated by my own interest in biology and evolution and my curiosity about its effectiveness in a simulation on a computer. As seeing is believing and because of the opportunity to do this research at the Department of Computer Science in Leiden, I decided to test a class of evolutionary algorithms on some very hard problems to partially answer at least for myself the question about the usefulness of evolutionary algorithms.

I would like to thank my supervisor, Guszti Eiben, for his support and for the freedom he gave me in doing this research. Further thanks to Han La Poutré, my second supervisor, for his corrections and to Zsofi Ruttkay for some suggestions. Lastly I would like thank all the people whose computer I used for the many time consuming experiments that needed to be done and who possibly got annoyed with it.

Contents

Chapter 1

Introduction

This Master's Thesis is about evolutionary algorithms, a class of search algorithms based on biological evolution. Because evolutionary algorithms are supposed to be robust and general search algorithms, there has been a growing interest in evolutionary algorithms during the last decades. Especially the last few years, they are becoming known to a wider public in the computer science community. Still, for many people, the question remains how good evolutionary algorithms are compared to existing algorithms. To get an idea about the answer, in this research two difficult problems are used as a testcase to compare existing algorithms for these problems with an evolutionary algorithm.

We restricts ourselves to the class of Constraint Satisfaction Problems (CSPs), that, simply stated, consist of a number of variables and a number of constraints on the values of these variables. Therefore the conclusions in this research are really only valid for CSPs. The two CSPs that are examined in this research are: Graph 3-Coloring and 3-Satisfiability. Both are known to be NP-complete problems and so, unless P=NP, no complete algorithms with polynomial time complexity are known to exist. In other words no efficient algorithms are known for these hard problems and therefore they are interesting as a testcase. The Graph 3-Coloring problem will be the main problem and the 3-Satisfiability problem will be used to verify the main conclusions.

The objective of this thesis is first to find a good setting for the parameters in the evolutionary algorithm, so that a reasonably optimized version is obtained. This version will be compared with a known, good algorithm to get an idea of the power and use of evolutionary algorithms. Then, when the evolutionary algorithm turns out to be inferior to the existing method, ideas will be suggested and tested to possibly improve the performance of the evolutionary algorithm. Throughout this research, as little domain knowledge as possible will be used, as a main advantage of evolutionary algorithms is their general applicability.

The next chapter is an introduction to the principles of evolutionary algorithms. Chapter 3 describes the class of evolutionary algorithms that is used in this report in detail. In chapter 4, the measures for comparing the algorithms will be defined and discussed. In chapter 5, the Graph 3-Coloring problem and existing algorithms will be described. Chapter 6 is the most important chapter in this report. In it the evolutionary algorithm is optimized, compared to the existing algorithms and some improvements are suggested and tested. In Chapter 7, the main results of the previous chapter will be verified on 3-Satisfiability. Finally, chapter 8 gives a summary of this research, a discussion of the main conclusions and some ideas for further research. In appendix A, a summary of the abbreviations and some notation is given.

Chapter 2

What are Evolutionary Algorithms?

Evolutionary algorithms (EAs) are search algorithms based on biological evolution. Darwin was the first to clearly state the idea of *natural selection* as a principle behind biological evolution [15]. Natural selection or "survival of the fittest"[1] is simply the process by which the best or most fit members of some species have more probability of surviving or by which the weakest members have more probability of dying out so that the weakest individuals will be selected out. As fit members live longer, they will have more offspring[2] so that on average the whole species will become fitter.

Sometimes people who do not want to accept evolution theory, say that natural selection is a purely negative force that just weeds out failures, but is not capable of building up the complexity that we see around us everywhere. They argue that it only subtracts from what is already there and that a more creative "designer" is needed that adds something as well. In Darwin's theory, *mutation* is the force that can add and all it needs is enough *time* to stumble upon the right mutations. Mutations, induced by for example radiation, can change an organism's *genes*, that form the *instructions* for building and regulating an organism and are found in each cell of the organism. By mutating the genes of new offspring, the way the offspring looks, behaves, et cetera can be changed. It should be realized that a small change in genetic material (genotype) can have great effect on the resulting organism (phenotype). So again, all that is needed is enough time for mutation to add to the complexity of organisms. Because our brains are built (by evolution itself) to deal with events on radically different timescales (hours, years or perhaps decades) from those that evolution had available (millions of decades), our intuitive judgments of what is probable turn out to be wrong by many orders of magnitude. Still, however much time we have, it seems impossible to maintain that, for example, an eye could come to existence in one step just by mutation. The answer to this is that evolution is a *gradual* process that cumulatively builds up complex designs from small changes that come from mutations. Of

[1]When a fit organism is defined as an organism which is good at surviving, we seem to get a trivial remark: survival of the organisms that are best at surviving. But maybe as it is so trivial, there's no way of denying it.

[2]In nature often the stronger individuals also have more probability of mating and because of this will create even more offspring.

course each of these small changes should have enough advantage to surviving or it would be selected out again. One could ask what the advantage is of half an eye as the eye is so complex that it does not work properly anymore if changed too much. The advantage of half an eye is simply that it's better than seeing nothing and even one lightcell could be an advantage. In [19], Dawkins very clearly and gradually explains how evolution can build up complex organisms with mutation, natural selection and enough time.

An example, drawn from [57], should clarify the basic ideas from evolution theory. Let's suppose that somewhere and sometime we have a population of cute, furry, little rabbits. Now some of them will be faster and smarter than other rabbits. These faster, smarter rabbits are less likely to be eaten by foxes (we assume a population of foxes as well) and therefore more of them survive to do what rabbits do best: make more rabbits. Of course, some of the slower, dumber rabbits will survive just because they are lucky. This surviving population starts breeding which results in a good mixture of rabbit characteristics: some slow rabbits breed with fast rabbits, some fast with fast, some smart with fast rabbits, and so on. And sometimes, nature throws in a "wild hare" by mutating some of the rabbit genetic material so that perhaps an exceptional smart, dumb or fast rabbit is created. The resulting baby rabbits will (on average) be faster and smarter than the ones in the original population, because more faster, smarter parents survived the foxes. (Fortunately for the foxes, the foxes are undergoing a similar process, otherwise the rabbits might become too fast and smart for the foxes to catch any of them).

Evolutionary algorithms (EAs) are inspired by natural evolution, though simplificate on many issues. Contrary to natural evolution, EAs are *goal-driven*: they are trying to evolve to a prespecified goal. The goal can be the finding of solutions for any problem that needs a solution and the EA maintains a *population* of potential solutions for this problem. In each member or *individual* the necessary information is stored to represent these partial solutions. The information is stored in *genes* and the individuals solely consist of genes, each gene representing a characteristic of the solution. A user defined *fitness function* is used for natural selection and for each individual gives a calculated *fitness* value that specifies how good the individual is. Now breeding is simulated by selecting *parents* from the existing population, called the current *generation*, that are allowed to mate and produce offspring (that are again potential solutions). Mating is done by selecting two parents, that produce one or more children by *crossing over* their genes so that information from both parents is combined in the *children*. The children are *mutated* by random changes in some of their genes to bring in new *diversity* in the population. Often a fixed population size is used and from the original population and offspring population, individuals have to be selected to form the next generation. The individuals are selected by favoring the fitter individuals and so giving them higher probability for surviving. In some EAs, individuals from one generation will not be allowed to exist in the next generation (similarly to many low organisms which reproduce only once). In other EAs, individuals from one generation are also allowed to exist in the next generation (similarly to many higher organisms which can mate several times). Because the fitter individuals are favored, the average population will become fitter and fitter as the generations go by so that the potential solutions will become better and better, approaching the final solution closer and closer.

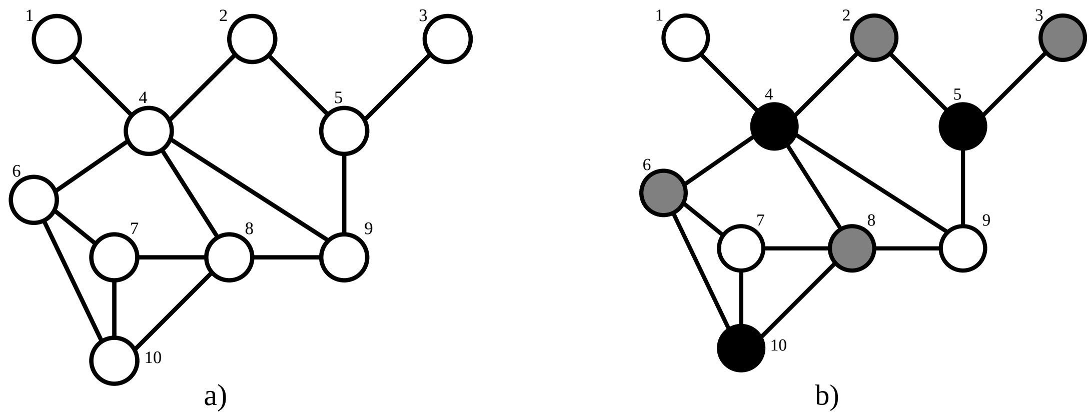


Figure 2.1: a) An example Graph Coloring problem. b) An example Graph 3-Coloring solution.

An example will show how this works in practice. The example problem will be the main problem used in this research: *Graph 3-Coloring*. Suppose that we have a number of *nodes* (points) that are connected by a number of *edges* (lines). An example with 10 nodes is shown in figure 2.1a. The Graph 3-Coloring problem is to color each of the nodes with one of three colors so that no two neighbors (e.g. nodes connected by a line) have the same color. One of the many possible solutions for our example is shown in figure 2.1b. Now all the information that we need to describe a solution is the color that we assign to each node and so a natural gene representation is to use as many genes as we have nodes and to give each gene a possible value between 0 and 2 to represent its color. Our solution would be represented as: *0112210102*. Our fitness function should have its optimal value when we have a solution and we could use for example the number of edges that connect nodes with different colors. The solution in figure 2.1b has the maximal fitness of 14. Now suppose we have a population of 20 individuals or colorings which in the beginning consists of random initialized individuals, so that each gene of each individual has an arbitrary value between 0 and 2 and so many individuals will have low fitness. If we want to create the new population for the next generation we select a number of parent pairs. Each parent is still coloring with non-optimal fitness, otherwise we would stop the EA. Each parent pair is crossed over, for example as shown in figure 2.2. A *crossover point* is selected, the information in the two parents is split at this point and the two halves are recombined in the offspring. Then each of the children is mutated by randomly selecting a few of their genes which are randomly given a color. So the children combine the information from the two parents, sometimes some of the existing information is changed by a mutation and hopefully better individuals are created. Now let's say we crossover 10 parents in each generation, creating 10 children. We have to reduce the resulting population of size 30 to size 20 again and we do this by selecting the ten individuals that are weakest (i.e. have lowest fitness) to die out. So now the next generation possibly consists of better individuals

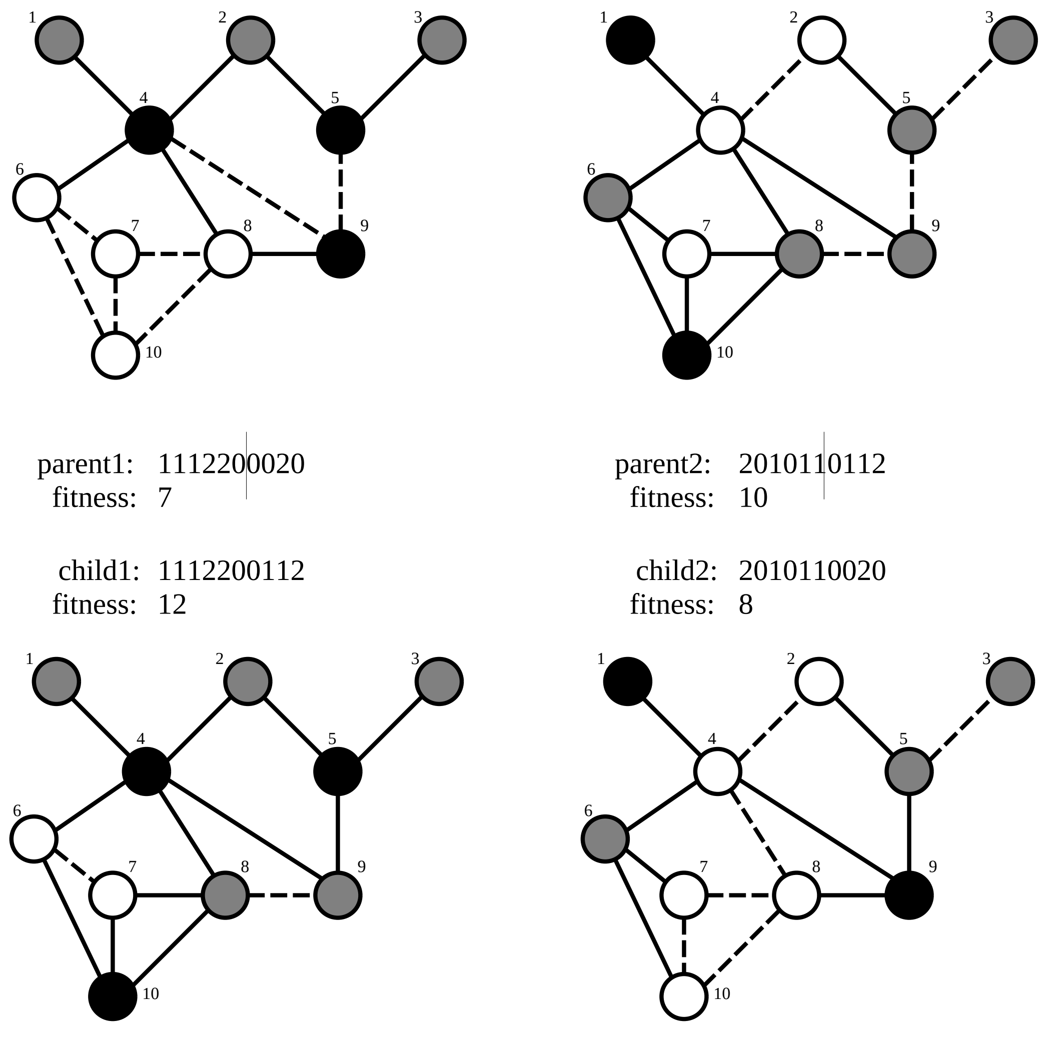


Figure 2.2: Example of crossing over two parents. The dotted lines show the edges that connect nodes with the same color. The line in the representation of the parents shows where the crossover point was selected.

if some of the children were better. If none of the children were better, because of favoring the best individuals, at least we keep the original population so we can try again in the next generation. If enough time is available so that enough generations can be produced, then the population will become better and better, hopefully leading to an individual that has the optimal fitness of 14, so that a solution is found. Of course a simple problem with 10 nodes can be done by hand as well and we don't need an EA for this, but for problems with hundreds of nodes and many edges this becomes infeasible, though EAs have proven to find solutions for this kind of problems as well.

The basic principles of EAs were first described by Holland [46] and further developments and descriptions can be found in many other texts (e.g. [17][42][53][57][67]). EAs are a more general class of algorithms that include genetic algorithms (GAs). One of the main powers of EAs is that they are very general and in fact can be used for any problem for which a fitness function can be defined and a gene representation exists. In the rest of this thesis, a few specific EAs will be tested to see how good they are, whether they can compete with other existing algorithms and how they possibly can be improved.

Chapter 3

The Evolutionary Algorithm in Detail

In this chapter the specific parts of the EA that will be used in later experiments will be described. In figure 3.1, the general EA framework that captures the EAs in this research is shown. A summary of the most important fixed features and parameters is shown in table 3.1 and will be discussed in the next sections.

```
procedure EARun()
    Initialize and Evaluate Population of size $\mu$
    while not Stopcondition
        Select Parents from Population
        Recombine Parents to produce $\lambda$ offspring
        Mutate each of the $\lambda$ offspring
        Evaluate each of the $\lambda$ offspring
        Select New Population of size $\mu$ again
    end
end
```

Figure 3.1: EA Framework.

3.1 A Steady State EA

A major design decision in an EA is how to select the population in the next generation from the current population and its offspring. One possibility is to select the next population from the *combination* of the current population and the offspring, so that in principle an individual can live forever. This is called *preservative selection*, because individuals can be preserved for the next population. This also models higher organisms that can compete

EA Type	steady state
Parent Selection	2-tournament
Deletion Strategy	worst fitness deletion
Stopcondition	solution found or T_{max} evaluations
p_c	1.0
p_m	$\frac{1}{L}$

Table 3.1: Major EA features and parameters that are kept fixed in the experiments.

with their offspring. Still in biology we see that an individual has limited lifespan. To model this in EAs, we select the new population only from the offspring. This is called *extinctive selection*, because the current population will be extinct in the next generation. This also models lower organisms which reproduce once and only live for one generation.

We will use the notation from Evolution Strategies (ES) [67] where μ denotes the size of the population and λ the size of the offspring population (i.e. the number of children created in the current generation). Preservative selection is notated as $(\mu + \lambda)$-strategy where the $+$ symbolizes that the new population is obtained from both current and offspring population. Extinctive selection is notated as a (μ, λ)-strategy and here we select the new μ individuals only from the $\lambda \geq \mu$ offspring.

In the traditional GA, the next population is found by producing $\lambda = \mu$ offspring and copying the whole offspring population to the next population, called generational GA (which would be a (μ, μ)-strategy in ES notation). Because the μ offspring are obtained by recombining selected parents from the current population, the new population will be quite different from the current one.

Recently, success was booked with *Steady State* EA's (SSEA) [73][77], that have as a characteristic that the population does not change much from generation to generation. This is ensured by using preservative selection and λ quite small compared to μ. The advantage is that new individuals are available immediately for use and it has been observed that SSEAs do find as good or better solutions in much less time than generational EAs[1]. In fact this is similar to what happens in nature for longer-lived species, where offspring and parents are alive concurrently and have to compete. Though for shorter-lived species, parents lay eggs and die before their offspring hatch, which is more like generational EAs.

Usually in SSEAs, λ is chosen so that the recombination operator is used exactly once per generation, which would give $(\mu + 2)$ selection in case of a reproduction operator that produces two children. Therefore in this research λ will be equal to the amount of offspring that an operator creates and a $(\mu + \lambda)$-strategy will be used. The mechanisms for selecting among the $(\mu + \lambda)$ individuals and for producing the λ offspring will be described in the next sections.

[1]It should be noted however that Goldberg and Deb [41] claim that SSEAs are not fundamentally better than generational EAs which should be able to obtain the same results with higher selective pressure.

3.2 Worst Fitness Deletion

Several mechanisms exist to select a population of size μ again from the $(\mu + \lambda)$ individuals (or from the $\lambda > \mu$ individuals in case of a (μ, λ)-strategy). Syswerda [73] suggests *random deletion*, *exponential ranking*, *reverse fitness* and *worst fitness deletion*. Random deletion could be implemented to run in $O(1)$ time but gives worse results than the others. Syswerda found worst fitness deletion and exponentional ranking the best, with exponential ranking just somewhat better than worst fitness deletion. Though exponential ranking has a parameter that depends on μ and as the number of parameters should be kept to minimum and μ will be varied often in this research, worst fitness deletion is used.

In worst fitness deletion, the λ individuals with worst fitness are deleted from the $(\mu+\lambda)$ individuals (ties are broken randomly[2]). Together with an SSEA, this ensures that the best individual is always kept in the population which is called *elitist selection*.

3.3 Tournament Selection

In [41] and [6] some mechanisms are investigated for selecting the parents that are used to produce offspring. In [6], Bäck uses the takeover time as a quantitative measure for the *selective pressure*: the force that gives fit individuals higher probability to be selected. He reports that selective pressure increases in the order *proportional selection*, *linear ranking*, *tournament selection*, (μ, λ)*-selection*. Still too much selective pressure can reduce the diversity of the population too fast and result in premature convergence, where almost no diversity is left in the population. He reports (μ, λ)-selection as the most flexible in the range of selective pressures. Though, as we work with an SSGA, the second most flexible, tournament selection, will be used in this report.

Tournament selection works by randomly drawing (with replacement) a number of individuals from the population to create a *tournament*. The winner from the tournament is the individual with best fitness (first random individual in case of a tie[3]) and is used as a parent. In this way all parents necessary for a reproduction operator are selected (with replacement).

The number of individuals, k, that are selected for the tournament, influences the *selective pressure*: the higher k, the higher the probability that individuals with good fitness are selected and the higher the selective pressure. When $k = 1$, random selection is used and to keep some selective pressure, often the lowest value for k is 2. In [50] it is noticed that "the worst fitness deletion mechanism is best combined with a low tournament

[2]In a first naïve implementation, ties were broken by taking the first individual from the (unsorted) population. It was found that the performance increases more than 10% if the ties are broken in a circular way (the next time, the individual with the next position in the population is taken). Another tested method was breaking the ties for which the increase in performance was similar randomly. Because random tie breaking is more in the spirit of EAs this is the preferred method.

[3]Because the individuals are picked randomly already it is no use to randomly pick again from these individuals.

size". With higher selective pressure the population will converge earlier and might get stuck more often in local optima. Therefore k is kept fixed to two in this research.

3.4 Representation

A problem specific part of an EA is the representation that is used to represent potential solutions in the genes of the individuals, together with a possible *decoder*. In principle each gene can have any thinkable value for this representation. The length of an individual, L, is defined as the number of genes it has. Like in most research, all individuals will have the same length. The two representations we use are *integer representation* and *order-based representation*. In integer representation each gene's domain simply consists of integer values and often each gene represents a variable from the CSP problem. Order-based representation uses an individual to store a permutation which is used in a deterministically decoded by a problem specific decoder. This decoder uses the permutation to process the variables in some order and to obtain the corresponding potential solution. Each gene specifies which variable comes at its position and the individual is constrained to have a different value for each gene, so that each individual results in a permutation of the variables. As integer representation requires less domain knowledge (i.e. no problem specific decoder), integer representation will be examined first.

3.5 Fitness Function

The fitness function, as the name suggests, defines a measure for the fitness of an individual and therefore plays a very important role in the EA. It should be noted that in the EA that is described here, the role of the fitness function is only to define a *ranking* on the individuals. Tournament based selection and worst fitness selection both only make use of the fitness in this way and therefore the EA is be said to be *rank-based*. In a traditional GA the fitness of an individual is used as an absolute value to create a probability distribution for selecting individuals. As Whitley noticed in [77], ranking avoids some problems of fitness proportionate selection. The value returned by an evaluation function is often only an approximation to the distance to the global optimum. In most cases it may not be realistic to use the value generated by an evaluation function to judge relative differences in fitness and ranking might be the best one can expect from an evaluation function. Also scaling problems are avoided, since ranking automatically induces a uniform scaling across the population.

The specifications of the fitness function highly depend on the problem and its representation and so will not be discussed here.

3.6 Reproduction

As described before, the EA in this report will use its reproduction operator only once per generation, so a minimal offspring population is produced to start selection as soon as possible. Analogously to biology, a mutation operator is executed on the offspring before evaluating them, though together with the recombination operator, we could see this as one reproduction operator. Contrary to a generational EA we do not want multiple copies of an individual in the population as they reduce the diversity. A way to reduce the probability of copies is to always use the recombination operator in each generation (so the recombination rate, p_c, is fixed to 1) and immediately after that, using a mutation operator for each of the λ offspring that possibly brings in more diversity.

3.6.1 Recombination Operators

For integer representation, we examine the two classical recombination operators: 1-point and uniform crossover, two possible generalizations of 1-point crossover: diagonal and m-point crossover, a generalization of uniform crossover: uniform scanning crossover and an adaptation of uniform crossover: HUX. For order-based representation, we examine order, order #2, partially mapped and cycle crossover. These are all described in the following sections.

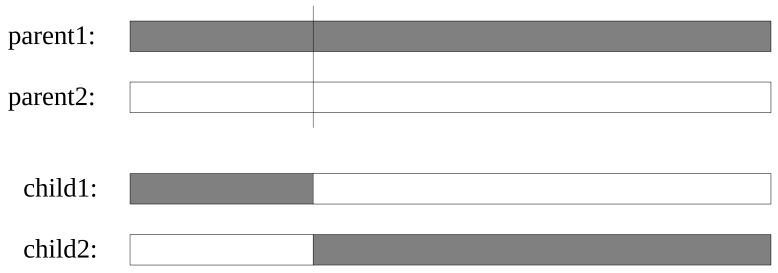


Figure 3.2: 1-point crossover.

1-Point Crossover

1-point crossover selects one crossover point by randomly selecting one of the L genes and choosing the crossover point right after it, thereby splitting the individual in two parts. The two parts are then recombined to form two children as shown in figure 3.2.

To have a more general 1-point crossover, it is allowed to choose the crossover point after the last gene, so that 1-point also includes 0-point crossover. In practice this did not really make a difference except for a possibly slightly faster implementation.

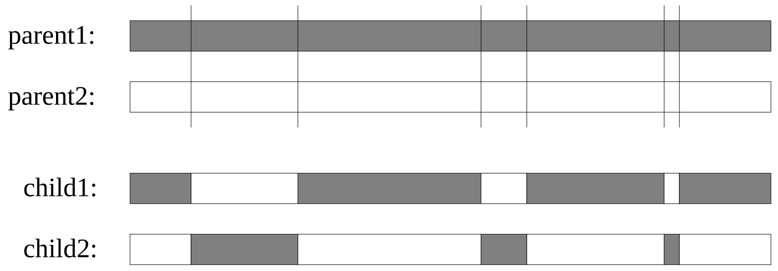


Figure 3.3: m-point crossover with $m = 6$.

m-point Crossover

A generalization of 1-point crossover can be obtained by choosing m crossover points and alternately copying the genes between two consecutive crossover points from the first parent and from the second as shown in figure 3.3.

Again a more general and slightly faster implementation is obtained when the crossover points are selected with replacement (and so are not necessarily distinct) so that an m-point operator includes all m'-point operators with $m' < m$.

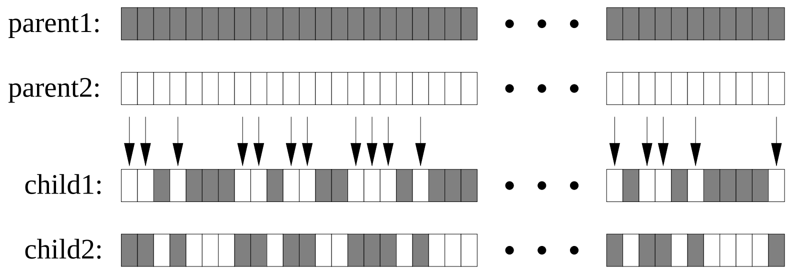


Figure 3.4: uniform crossover. An arrow at position i means the genes at position i will be crossed over.

Uniform Crossover

A further generalization of m-point crossover, is uniform crossover [71], that uses the extreme number of crossover points so that all genes separately can be crossed over. Further each gene has a probability of being crossed over, which will be fixed to 0.5 in this research. So it's easiest to see it as if parent 1 is copied to child 1 and parent 2 to child 2 and then for each gene position i a random number rnd_i in $[0, 1)$ is selected. If $rnd_i > 0.5$ then gene i is crossed over between the two children. This is graphically shown in figure 3.4.

Because uniform crossover exchanges genes instead of gene segments, it can combine features regardless their relative location. For problems where the distance between the position of two genes does not say anything about the relationship between them, this can be an advantage. For other problems uniform crossover was inferior to 2-point crossover [71].

So for each gene position we have to draw a random number to decide if it will be crossed over. Because drawing random numbers is quite costly, an optimization will be done which is possible because the probability was fixed to 0.5. As one random number consists of 32 bits, we can use the value of each bit for deciding to crossover a gene, hereby reducing the number of necessary random numbers for uniform crossover almost by a factor of 32.

HUX Operator

The HUX (heuristic uniform crossover) operator, designed by Eshelman [26], is based on uniform crossover. The idea is to have a maximally disruptive crossover operator, to exploit the search space as much as possible. He argues that crossing over always exactly half of the differing bits between two parents produces two children with maximal (Hamming) distance from these parents (bits to be exchanged are taken randomly without replacement). He designed HUX for bit representation, but a similar crossover operator can be made for integer representation by ensuring that always exactly half of the differing genes will be crossed over. On average, uniform crossover also exchanges half of the differing bits, because probability 0.5 is used to decide on crossing over. Uniform crossover, however, will have some variation in the number of exchanged bit, where HUX has no variation as it always crosses over exactly half of the differing bits.

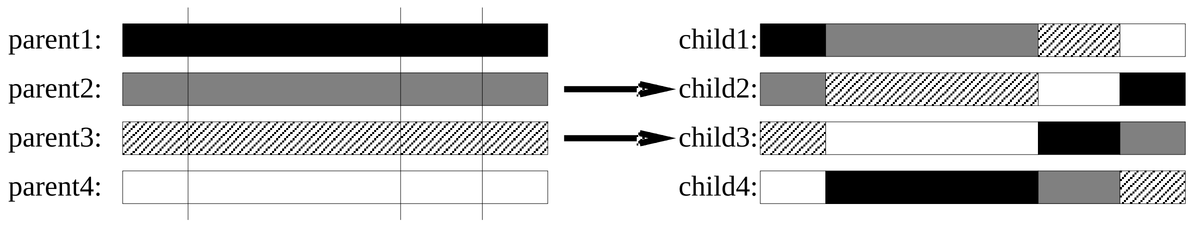


Figure 3.5: Diagonal Crossover with 4 parents.

Diagonal Crossover with m Parents

Another generalization of 1-point crossover by Eiben [22][21] is diagonal crossover (DIAG). Instead of using two parents, DIAG generalizes to using m parents. The idea is that more information from the population is combined this way and that a more disruptive operator is obtained. For DIAG, $m-1$ crossover points are selected randomly, so that each parent is divided into m parts that can be combined to create new children. As shown in figure 3.5, only the combinations along the diagonals are selected. This is to keep the

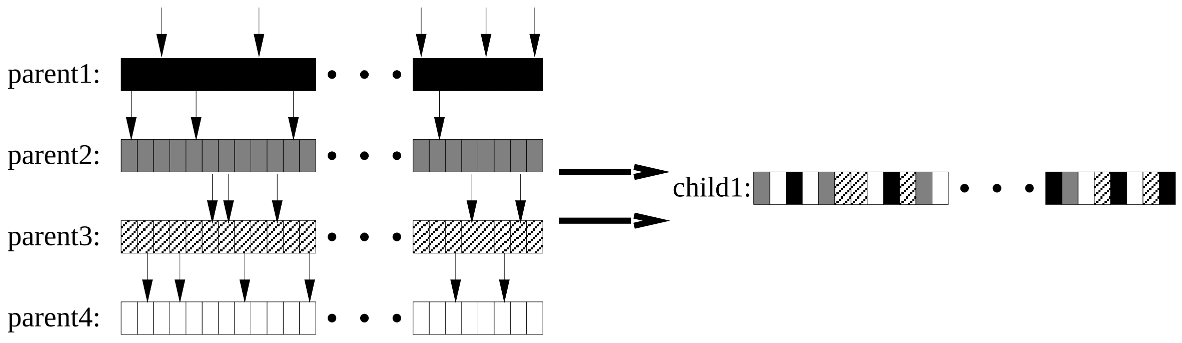


Figure 3.6: Uniform Scanning Crossover with 4 parents.

number of children acceptable as m children are created in this way. Diagonal crossover with 2 parents is equal to 1-point crossover.

The crossover points are randomly drawn with replacement and so are not necessarily distinct. Some tests have been done to see what the difference is and the results indicate that it does not really matter whether the crossover points are drawn with or without replacement. As drawing with replacement can be done somewhat faster, this is the version used in this report.

Uniform Scanning Crossover with m Parents

A generalization by Eiben [22][21] of uniform crossover is uniform scanning crossover (SCAN), that also uses m parents. SCAN only creates one offspring per m parents as more offspring is not really natural for this operator. With m parents, for each gene position i in the child, the gene at position i in each of the m parents is available. A choice is made by uniformly giving each parent probability $\frac{1}{m}$ that its gene will be selected[4].

Now for each gene we need a random real number and for each offspring we have to select m parents, which makes the SCAN operator quite costly.

Copy

Perhaps a trivial operator is the Copy operator, which takes one individual as parent and copies it exactly to create one child. This operator is necessary for an EA where only mutation is used and a child is a mutated copy of it's parent.

[4]Another possibility is occurrence based scanning crossover, where the gene which occurs most often among the parents at position i is selected. Though this operator gave inferior performance when tried in some initial tests.

$$\begin{array}{lcccc|ccc|cccc}
\text{Parent 1:} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{Parent 2:} & 10 & 0 & 6 & 5 & 1 & 3 & 7 & 8 & 9 & 2 & 4 \\
 & & & & & & & & & & & \\
\text{Child 1:} & 0 & 1 & 3 & 7 & 4 & 5 & 6 & 8 & 9 & 2 & 10 \\
\text{Child 2:} & 2 & 4 & 5 & 6 & 1 & 3 & 7 & 8 & 9 & 10 & 0
\end{array}$$

Figure 3.7: Order Crossover.

Order Crossover

Order crossover (OX), an operator that works on permutations, was developed by Davis [17] and creates two children which preserve the order and position of symbols in a subsequence of one parent while preserving the relative order of the remaining symbols from the other parent. It is implemented by selecting two random cut points and copying the genes between the cut points (including the cut points) from the first parent into the child, into the same positions as they appear in the parent. Then starting just after the second cut point, the symbols are copied from the second parent into the child, omitting any symbols that were copied from the first parent. When the end of the second parent sequence is reached, this process continues with the first symbol in the second parent until all of the symbols have been copied into the child. A second child is constructed by switching the roles of parent sequences. Figure 3.7 shows an example.

$$\begin{array}{lccccccccccc}
 & & & \downarrow & \downarrow & & & \downarrow & & \downarrow & & \downarrow \\
\text{Parent 1:} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{Parent 2:} & 2 & 5 & 0 & 9 & 7 & 3 & 8 & 6 & 1 & 4 & 10 \\
 & & & & & & & & & & & \\
\text{Child 1:} & 0 & 9 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 1 & 10 \\
\text{Child 2:} & 2 & 5 & 0 & 9 & 7 & 3 & 6 & 8 & 1 & 4 & 10
\end{array}$$

Figure 3.8: Order Crossover #2.

Order #2 Crossover

Order crossover #2 (OX2), developed by Syswerda [17], differs from OX in that several key positions are chosen randomly and the order in which these elements appear in one parent is imposed on the other parent to create the two children. So in the example in figure 3.8, child 1 is created by reordering the selected genes from parent 2, so with numbers 0,9,8,1 and 10, in parent 1 in the order given by parent 2. All the other genes are directly inherited from parent 1.

Each gene is selected with probability $\frac{1}{2}$, so that in the average case half of the genes will be selected, so that the operator is maximally disruptive in the average case[5].

Parent 1:	0	1	|	2	3	4	5	|	6	7	8	9	10
Parent 2:	2	5	|	0	9	7	3	|	8	6	1	4	10
Child 1:	2	1	|	0	9	7	3	|	6	4	8	5	10
Child 2:	0	9	|	2	3	4	5	|	8	6	1	7	10

Figure 3.9: Partially Mapped Crossover.

Partially Mapped Crossover

Partially mapped crossover (PMX), by Goldberg and Lingle [42] preserves the order and position of genes in a subsequence of one parent while preserving the order and position of many of the remaining genes from the second parent. It is implemented by selecting two random cut points which define the boundaries for a series of swapping operations. The first child begins as an exact copy of the first parent. The goal is to modify the child so that the symbols between the cut points are an exact copy of the symbols between the cut points in the second parent. This is accomplished by a series a swap operations where each symbol that must appear between the cut points is swapped with the symbol that occupies its desired position. When a gene needs to be swapped with a gene that already is between the cut points, the first gene is skipped over temporarily. When at the end the second gene has moved elsewhere, the swap is tried again. An example is given in figure 3.9.

				↓							
Parent 1:	0	1	2	3	4	5	6	7	8	9	10
Parent 2:	2	5	0	9	7	3	8	6	1	4	10
Child 1:	2	1	0	3	4	5	6	7	8	9	10
Child 2:	0	5	2	9	7	3	8	6	1	4	10

Figure 3.10: Cycle Crossover.

[5]Position based crossover, proposed by Syswerda [17], is almost the same as OX2, only now the genes that are not selected are used to impose the ordering. As we use probability $\frac{1}{2}$ to select genes, these two operators are exactly the same.

Cycle Crossover

Cycle Crossover (CX), developed by Oliver *et al.* [61], preserves the absolute position of elements in the parent sequence. A randomly chosen cycle starting point is selected. The element at the cycle starting point in the first parent is inherited by the first child. The element which is in the same position in the other parent then cannot be placed in this position so its position is found in the first parent and is inhererited from that position by the child. This continues until the cycle is completed by encountering the initial item in the unselected parent. Any elements which are not yet present in the offspring are inherited from the second parent. The second child is created by inverting the roles of the parents. An example is shown in figure 3.10.

3.6.2 Mutation Operators

For integer representation, the standard mutation operator (with $p_m = \frac{1}{L}$) and for order-based representation SWAP, RAR and Inversion mutation are described in the next sections.

Standard Integer Representation Mutation

The standard mutation used here is the classical mutation where each gene of an individual has some probability, the mutation rate p_m, of being mutated to one of the values in its domain including its current value. When mutated genes would always receive a new value, the mutation operator would be more disruptive (when p_m is fixed). Because the operators that are used are already quite disruptive, first the less disruptive version of the mutation is used. So for Graph 3-Coloring, the a gene can be mutated to its old value. For Satisfiability however, a mutated gene was always flipped to a new value as this made a better comparison with the existing algorithm possible. Also it was found for Graph 3-Coloring that the EA with only mutation performs the best. For this EA, a slightly more disruptive mutation operator could be better.

At the moment $p_m = \frac{1}{L}$ (with L the number of genes in an individual). In the literature it is suggested that this is a reasonable value in general [59]. Later the effect of the mutation rate will be examined further (see section 6.5).

A fast implementation is obtained (see section 3.11) by using the probability distribution of the number of genes that will be mutated with $p_m = \frac{1}{L}$. Using this distribution, we compute the number of mutated genes so that only a few random numbers are necessary to decide the position of these mutated genes and their values.

SWAP

SWAP is an order-based mutation that simply swaps pairs of genes. Often p_m for an order-based mutation defines the probability of swapping one random pair of genes. In this report however, p_m is used to give the probability *per gene* of using that gene as the first gene in a swapping. This is analogously to the mutation for integer representation. If a gene is used as the first gene of a swapping, then the second gene is selected randomly. In this way more than one pair of genes will sometimes be swapped per mutation, which gives a more general operator. Later we will see that it can be useful to swap more than one pair of genes per mutation. Again $p_m = \frac{1}{L}$ will be used so that the expected number of swapped pairs of genes will be one pair per SWAP operation.

In the implementation again the probability distribution for $p_m = \frac{1}{L}$ can be used to calculate the number of swappings for each SWAP operation. Then for each swapping two genes will be selected randomly (with replacement).

Inversion

The Inversion operation selects two randomly selected cut points (with replacement) and inverts the sequence (including cut points) between these points. Again several inversions per Inversion mutation can occur, by defining p_m again as the the probability per gene of selecting the gene as the first gene in an inversion operation. The second gene is selected randomly and $p_m = \frac{1}{L}$. Again in the implementation the number of inversions per mutation is calculated and two random genes for each inversion are selected.

RAR

The RAR (remove and reinsert) operation removes a randomly selected gene, shifts all subsequent genes up to (and including) a second randomly selected gene to the left and reinserts the first gene after the second. Analogously to the other order-based mutations, several RAR operations can be done in one RAR mutation with an expectation on one RAR operation per mutation.

3.7 Initial Population

The details of creating the initial population can be dependent on the problem, but always all individuals are created in a random way.

3.8 Population Size

The population size, μ, turned out to be a very important parameter with its twofold effect as described next.

In an EA that only uses a recombination operator (with two or more parents) and no mutation, it is necessary to keep *diversity* in the population, which can be exploited by the recombination operator to explore the search space. Without diversity in the population we can stop the EA, because recombining identical parents is useless. So for a recombination operator a population is essential. Now the bigger we take μ, the longer it will take before the population converges (i.e. when no diversity is left) and the more probability the EA has to find the global optimum.

On the other hand, increasing μ too much can have a negative effect on the number of necessary search steps. EAs are based on the idea of gradual evolution, meaning we expect that reproducing fit parents will give fit children. This means that the probability of producing fit children increases with the probability of selecting fit parents. The probability of selecting the fittest parents increases inversely proportional with the population size and according to this view it can be expected that more search steps are necessary with a bigger population size. So we have to strike a balance between both views to find an optimal population size.

In an asexual[6] EA, where only a mutation operator is used, we do not necessarily need a population because the mutation operator is able to explore the search space on its own force. Therefore we can expect the optimal population size for an asexual EA to be smaller than for an EA that uses recombination.

Experiments indicated that an EA with only recombination performs very poorly, so that always at least a mutation is used and in an EA with the combination of both operators we can expect a complicated trade off in the optimal population size. This will be further be discussed later in this report. A question is if this optimum is dependent on specific problem instances.

3.9 Stopcondition

An EA cannot guarantee that a solution is found if one exists, therefore it is called an *incomplete* algorithm and so we have to define a stopcondition for the algorithm. Of course, in case of a CSP we can always stop the EA as soon as a solution is found, but the problem is when to stop the EA when a solution is not yet found. Two basic methods are considered.

3.9.1 Convergence Based Stopcondition

If we consider the recombination operator to be the most important operator and mutation only as a background operator to reintroduce lost diversity, it could be considered useless to continue the EA in a converged population as only mutation would be left in the search. Therefore a natural stopcondition is to stop the EA as soon as the population is converged. We can look at the *phenotypic* diversity, the diversity in the *fitness* of the individuals in the population or at the *genotypic* diversity, the diversity in *genes* among the different individuals. The phenotypic diversity can be measured by:

- Difference in best and average fitness of the population. If this difference is zero, all individuals will have equal fitness.

[6]In fact this name is not correct as in most EAs no difference in sex is made, but as this is the name that is used in the literature it will be used in this report. Another name used in the literature is naïve evolution.

- Variance in fitness over all individuals. If the variance is zero, all individuals will have equal fitness. As a measure for the diversity, this measure is a bit more precise than the previous measure, though it's slightly more costly to compute.

If there is no phenotypic diversity left, there can still be genotypic diversity in that different individuals exist with the same fitness. This is observed to happen in practice. Therefore also some genotypic measures of the diversity are considered:

- Sum over all genes of the *variance* per gene over all individuals:

$$\sum_{i=1}^{L} \frac{1}{\mu} \cdot \sum_{j=1}^{\mu} (x_{i_j} - \overline{x_i})^2$$

  where x_{i_j} is the j^{th} gene of the i^{th} individual in the population and $\overline{x_i}$ is the average of all the i^{th} genes in the population. The minimum (zero) will be reached when all individuals are identical. The maximum will be when all gene values are as far from the average as possible, which is not to be expected in the initial population which gives a uniform distribution. An advantage is that it can also be used for problems with real valued genes.

- Sum over all genes of the *entropy* per gene over all individuals [27]:

$$E_i = \frac{-\sum_{j=1, n_{ij} \neq 0}^{|D|} (\frac{n_{ij}}{\mu}) \log(\frac{n_{ij}}{\mu})}{\log |D|}$$

$$E = \frac{\sum_{i=1}^{L} E_i}{L}$$

  where n_{ij} is the number of individuals in the population for which gene i has value j, with $1 \leq j \leq |D|$, $|D|$ the number of domain values for each gene. Then E is a normalized value in the interval $[0, 1]$. The minimum will be when all individuals are identical. The maximum will be when all gene values are uniformly distributed, as will be the expected case in the initial situation[7]. Therefore the entropy is the most precise measure of the diversity. A disadvantage is that it can only be used for a relatively small set of discrete values and is costly to compute.

It was observed that the last two measures hardly ever reach the minimum, but there is clearly a point where the diversity stays very small. A possibility is to introduce a threshold which indicates that the diversity is small enough to stop the EA, but then again a problem is what value to choose. The first measure, based on the difference in best and average fitness, does reach zero in the area where the other three measure have very small values. Therefore if this measure (also the fastest to compute) is used to represent the diversity, no extra parameters are introduced. Also this measure will always become

[7]Still an unlikely case can be imagined with only $|D|$ different individuals that has an entropy of 1.

zero at some point, so therefore we have a clear stopcondition without introducing extra parameters, which is a big advantage for the generality of the EA.

A disadvantage of using a convergence based stopcondition is that it was observed that the performance drops drastically. This because after converging often "good" mutations were able to lead to a solution and all these solutions will not be found with the convergence based stopcondition which shows the importance of the mutation operator and indicates that it might be more than a background operator, which is observed by more and more researchers.

3.9.2 Maximum Number of Evaluation Based Stopcondition

Another view is that mutation is at least as important as recombination, which is justified by the observation that many solutions for hard graph instances were found after the population had converged because of a "good" mutation that reinitiates the search. In this view it becomes important to continue the EA after it has converged and the easiest is to define a maximum (T_{max}) on the number of search steps (a search step defined as the evaluating of an individual) that can be used in one run of the EA. This is often used in the literature and gives each EA version the same number of search steps so a fair comparison seems possible.

As T_{max} can greatly influence the performance, it has to be hand-tuned for each class of problem instances and EA settings to find its optimal value, which is problematic when different algorithms have different optimal values. The problem is that EA_1 might give worse performance than EA_2 for $T_{max} = t$, whereas EA_1 could give better performance than EA_2 for $T_{max} = t^{'}$ and $t^{'} > t$. The best would be to stop when all compared algorithms have found most of the solutions that would be found if $T_{max} = \infty$, so choosing the maximal T_{max} of all compared algorithms.

Fortunately there is a way to approximate the optimal value, or more accurately to notice whether T_{max} is too small. The idea is that as an EA is a stochastic algorithm, the number of search steps that are needed to find a solution (for some EA setting), follows some distribution. If AVG is the average number of evaluations needed to find a solution and SD its standard deviation, then according to the statistical *central limit theorem*, we can approximate this distribution with the (AVG, SD)-normal distribution when enough runs are used to calculate AVG and SD. Now simple statistics tell us that for $T_{max} = AVG + 2 \cdot SD$, 96% of the solutions will be found that would be found for $T_{max} = \infty$.

Still, to calculate AVG and SD, we have to use a maximum number of evaluations because $T_{max} = \infty$ often is infeasible. But if this is too small than AVG and SD will be too small as well and so we can assume that the real AVG and σ are bigger than or equal to the calculated values. Now if we see that the T_{max} used for calculating AVG and SD, is *smaller than or close to* $AVG + 2 \cdot SD$ than we can be certain that we found less than 96% of all the solutions that would be found if we used an infinite number of evaluations. Though we cannot assume the inverse! If $T_{max} > AVG + 2 \cdot SD$, perhaps we still found less than 96% of the solutions. As an example: one time 38 solutions from the 100 runs were found with $AVG + 2 \cdot SD < 100.000$, though 54 solutions were found for $T_{max} = 200.000$ and then we saw that the new calculated AVG' and SD' for this test were bigger so that $AVG' + 2 \cdot SD' > 100.000$.

Because in initial experiments, the importance of the mutation operator was observed, in all experiments a T_{max} will be defined as the stopcondition.

3.10 Incest Prevention

Interesting in EAs is that they are inspired by biology and sometimes ideas from biology also apply to EAs. As we know from nature, it is important for a species to keep enough diversity in the population. Now mating similar individuals (like in incest or in-breeding) reduces the diversity and Eshelman designed a mechanism, called Incest Prevention (IP) to prevent similar individuals from recombining [26].

The idea is to only recombine parents that are maximally different to ensure that the children will not resemble other individuals already present in the population. The mechanism that Eshelman uses works only for two parents and a bit representation, because it uses the Hamming distance between the two parents as a measure for the difference (d). In a random population the average distance between two individuals will be half the number of bits in an individual and therefore he uses this as the initial minimal distance d_{min}. Parents are only allowed to recombine if their difference d is bigger than d_{min}. If $d \leq d_{min}$, new parents are selected. After a number of failed attempts (as 10 gave reasonable results this was the maximum number of fails), d_{min} is decreased one bit.

To generalize this mechanism to integer representation, the distance is measured just by taking the number of genes that have a different value and initially $d_{min} = \frac{|D|-1}{|D|}$ with $|D|$ the size of the domain of the genes. In a real-valued representation, the distance has to be measured in a different way.

To generalize IP to work with more than two parents, we need another difference measure as no distance is defined between three or more parents. For a set of parents we can use the two genotypic diversity measures from section 3.9.1. Two different possibilities are considered:

- Taking the sum of the measured diversity over all genes and use this as the total difference between the parents
- Count the number of genes that have a diversity above a minimal diversity threshold (δ).

For order-based representation it does not seem natural to define an incest prevention mechanism, because absolute position is not important anymore. For example, when a RAR mutation would remove a gene and reinsert it somewhere else, the parent and child would be completely different according to the above difference measures, though the obtained coloring can be very similar for parent and child. A possibly better measure for the difference between two permutations would count the number of differing edges which would be quite low in case of a RAR mutation. However this would be costly to compute, as a difference calculation will be in $O(n^2)$ instead of $O(n)$. Better measures for the difference between permutations can possibly be found, but we leave this for further research.

3.11 Implementation Characteristics

The computers that are used in the experiments are an Indigo2 (which was always used for CPU-timings) and several Indy workstations. The EA software is written in C++ and compiled with GNU C++ compiler always with optimization option –O3, which gives a speedup of more than factor two.

To generate random numbers, the MLCG (Multiplicative Linear Congruential Generator) random number generator from the GNU C++ library is used which is currently considered among the best available. It redistributes the two seeds to avoid any dependencies. The first seed is fixed to zero and the second seed is set to the number of the run.

The implementation of the population is an unsorted array of individuals as it makes adding and removing faster, though it is costly to find the worst individual. As a simple optimization, in a converged population, the worst individual is found by selecting a random individual. Although possibly a more efficient datastructure can be designed, still we cannot avoid that a bigger population size makes the EA slower.

To decrease the runtime of the EA, a fast implementation of the classical mutation is used in which the probability distribution for mutating a gene is explicitly calculated with the following formula:

$$P(k \text{ genes mutated}) = \begin{pmatrix} L \\ k \end{pmatrix} \cdot p_m^k \cdot (1 - p_m)^{L-k}$$

For $k \geq \kappa$ and some number of κ ($\kappa = 10$ was used), $P(k$ genes mutated) can be considered zero, so only κ probabilities have to calculated. When using this distribution, the *number* of genes that will be mutated and the *positions* of these genes are the only values for which a random number is needed. This reduces the time of mutating an individual to almost zero relative to the time for a recombination and speeds up the total runtime of the EA more than a factor of two. Tests have been done to ensure that the same results are obtained when this fast version of the classical mutation is used.

Generally, the evaluation time per individual takes relatively the most time so this is the place to optimize the algorithm. Realizing that many children will differ only in a few genes from their parents (especially in an almost converged population), possibly a lot of speedup can be gained by only *locally* reevaluating the fitness of an individual.

The easiest is to consider a general function *SetAllel()*, that sets a gene to a specific value and locally reevaluates the fitness of the "new" individual, by checking only the constraints that can be effected by changing this gene. Now this function can be used in recombination operators as well, by copying the parents (with their fitness) and using this function to set the changed genes. Especially in a converged population little genes will receive a new value and then this implementation will have biggest effect. When a lot of genes will receive a new value, as is the case for recombinations in a very diverse population, the new evaluation method might be more expensive, though an asexual EA that only uses a mutation operator is spectacularly faster with this implementation. For Graph 3-Coloring a speedup in runtime of about 3 to 4 has been gained by using this alternative evaluation mechanism in an EA with recombination. Unfortunately for an order-based representation it does not seem possible to locally adapt the fitness as the color of each gene depends on the colors of the genes earlier in the permutation.

Chapter 4

Measures for Comparing Algorithms

A goal in this report is to compare EAs with traditional methods and EAs with other EAs and so measures have to be defined to do this comparison upon. In this chapter some measures will be defined and discussed. As we're dealing with probabilistic algorithms, the measures have to be averaged over a number of runs to be able to say anything about their behavior in general. It was found that this number of runs is of crucial importance as differences between runs can disappear if the number of runs is increased. So a balance must be made between reliability of the results and available time.

4.1 Success Rate

Because an EA is an incomplete algorithm, we cannot guarantee that no solutions exist when the EA does not find any. To avoid this, only solvable instances are used in this research. As we work with CSPs and a solution is always forced to exist, we can compare the algorithms on *the time that they need to find a solution*. Theorems exist saying that some probabilistic algorithms are guaranteed to find a solution if one exists [20], but infinite time is needed to guarantee this and in practice we will need to define a maximum on the *computational effort* that an algorithm is allowed to take, so that not in all runs a solution will be found. As the results are averaged over multiple runs, a natural measure to compare algorithms with, is the percentage of runs in which a solution is found, the success rate (SR). Though in the case that algorithms give the same success rate, we need a second measure to compare the algorithms with. Because in practice, we only have a limited amount of resources, the effort that an algorithm takes to find a solution (if one is found) seems to be the second most important measure. The success rate remains the most important, as we have no need for an algorithm that takes very little effort, but has SR=0.0. However, when SR ¿ 0.0, restarting a fast algorithm with low SR might be preferred above a slow algorithm with high SR. This might be an argument for using the effort as the most important measure, but, as will be argued in the next two sections, using the effort to compare different kind of algorithms gives several problems. The measure that will be used for effort will be discussed in the next section.

4.2 Computational Effort

In case of equal SR, we need another measure to do the comparison with: the *effort* that an algorithm takes to find solutions (when it finds any). The most obvious measure of effort is the time[1] that the computer needs to run the algorithm in seconds, minutes or hours (CPU-time). Of course the CPU-time for an algorithm depends on the specific implementation and on the hardware that is used and so to fairly compare algorithms, the algorithms should be written in the same programming language, compiled with the same compiler (and options) and run on the same machine so that only the algorithm itself can make the difference. Though as each implementation of an algorithm will be different, we really need one standard implementation of a best algorithm in one standard language that is available to all researchers of that problem. Unfortunately often this is not the case (yet).

Another way to compare an algorithm to other algorithms for the same problem, without needing to implement the other algorithms, is by defining a basic *search step* for an algorithm that is the major factor in the algorithm's run time. Then simply the number of these steps is used as the measure of the effort, and this is independent of the exact implementation, hardware and compiler. Though only algorithms that have exactly the same search step can be compared with this measure, otherwise search steps could be combined to reduce the number of search steps. For EAs often the evaluation of an individual is seen as a search step, because it takes relatively the most CPU-time in an iteration of the algorithm and because each evaluation stands for the creation of a point in the search space. The average number (over all runs) of evaluations to find a solution (AES) will be reported to compare algorithms with similar search steps. For other algorithms, the search step will have a different nature. For backtracking (like for DSatur in section 5.4.2), a search step will be identical to a backtrack step.

Although seemingly reasonable, this causes problems when algorithms with different kind of search steps have to be compared. For EAs, another problem is when the time needed for reproducing children is in the same order of magnitude as the time needed for evaluating them and differs per EA.

Obviously we do need a way to compare algorithms that use different search steps. This can be done by multiplying AES, by the average CPU-time needed per evaluation (ATE[2]) or by just reporting ATE together with AES, which will be done when necessary. To keeps things more readable, ATE will be reported in seconds per 1000 evaluations. As noted before, to compute ATE, the different algorithms will be implemented in the same language, compiled with the same compiler (and options) and run on the same computer. The previous remark that all implementations are different will be taken into account by

[1]As lack of memory is not really a problem for the problem sizes in this report, it will not be included in the measure of effort.

[2]ATE gives the total time involved with an evaluation by dividing the total run time (without initializations) by the total number of used search steps. Because, in this report, the evaluation function locally recomputes the fitness, ATE decreases when more evaluations are used, as the difference between parent and child becomes smaller when closer to convergence.

reporting some of the implementation details and by implementing reasonably fast versions of each algorithm.

Still a low SR and very low AES can be better than higher SR and high AES and so it's not fair to look at the SR first and at AES only in case of equal SR. An attempt to solve this problem is presented in the next section.

4.3 Expected Number of Functions Evaluations

In this section a kind of combination of SR and AES [53] is presented. In a practical situation we're really only interested in the time necessary to find a solution or to say that a solution will not be found. Because EAs are incomplete algorithms we cannot say whether a solution will *never* be found. But as they are probabilistic algorithms we can give the optimal expected number of function evaluations necessary to find a solution with certainty α (EFE$_\alpha$). And possibly this can also be used to say with some certainty that no solution will be found after a specified number of evaluations.

If we know for an algorithm the probability of success when run for some specified length of time t, we can compute the number of runs r (with length t) so that at least one solution will be found with some given certainty α as follows:

$$\begin{aligned}P(\text{at least one solution in } r \text{ runs of length } t) &=\\ 1 - P(\text{No solutions in } r \text{ runs of length } t) &=\\ 1-(1-SR(t))^r\end{aligned}$$

Where $SR(t)$ is the success rate for t search steps, so a probability distribution defined on t. If we equal above probability to α, we get:

$$\begin{array}{rcll} 1-(1-SR(t))^r &=& \alpha & \Longleftrightarrow\\ (1-SR(t))^r &=& 1-\alpha & \Longleftrightarrow\\ r &=& {}^{(1-SR(t))}\log(1-\alpha) & \Longleftrightarrow\\ r &=& \frac{\log(1-\alpha)}{\log(1-SR(t))} & \end{array}$$

We can compute the total expected number of timesteps to find a solution with certainty α for each t with $r \cdot t$. By computing this for all t, we can find the optimal number of search steps before the EA should be restarted. To get a really good approximation of SR(t), we should do many runs for each value of t, for $0 \leq t \leq T_{max}$. As this would be much too costly, we approximate SR(t) by using the runs we're doing anyway to compute SR and AES. Then the runs for which a solution is found give us a sample of the distribution[3]. For example, when we do R runs and one of the runs finds a solution after t' timesteps and r other runs found their solution for $t \leq t'$, then we approximate SR(t') with $\frac{r+1}{R}$, meaning we expect to find SR=$\frac{r+1}{R}$ for $T_{max} = t'$. The better approximation of the probability

[3]It should be noted that in extreme cases only one sample can be found. For SR=0.0, EFE$_\alpha$ is not defined.

distribution SR(t) we want, the more runs of the algorithm we have to do and the higher we should choose the maximum number of search steps, T_{max}. So from all the samples we can compute an optimal t^* for which the expected number of evaluations, $r \cdot t^*$, needed to find at least one solution with certainty α, will be minimal. EFE$_\alpha$ will be equal to this minimal product $r \cdot t^*$.

Interesting about this measure is that it shows that the success rate alone is not a good measure. It can happen (and happens!) that a small increase in success rate is possible only for a big increase in t and that the optimal t^* will not increase, indicating that the smaller success rate and the higher number of restarts is preferred.

Again, because different algorithms have different defined search steps, the expected *time* necessary to find a solution with certainty α (ET$_\alpha$=EFE$_\alpha$·ATE), will be reported as well when necessary. In fact, ET$_\alpha$ would be the ideal measure for comparing different kind of algorithms. Still the earlier noted problem remains that the measure is implementation dependent. Another problem in this report is that we make an approximation of the probability distribution SR(t) which sometimes is only based on just a few runs. Making a better approximation requires many runs and would be too time consuming. Therefore, in this report, still most attention is paid to the success rate. A certainty of $\alpha = 0.9$ is used when no other value is specified.

4.4 Other Measures

Often in EAs the average fitness of the best individual in the last evaluation (AFLE) is reported. When the fitness is a sum of the violated constraints, this measure is only interesting when no solutions are found and so it is averaged over the runs where no solution is found. Though for CSPs, we're really only interested in solutions (so all constraints should be satisfied). So, this measure is not very interesting because a partial solution that only violates one constraint can still be enormously far from a real solution.

Because of the stochastic nature, averages are computed over a number of runs and standard deviations for AFLE (SFLE) and AES (SES) were computed as well.

Chapter 5

Graph 3-Coloring

Graph Coloring was selected as a testcase for EAs because it is known to be NP-complete and because it is a direct instance of a more general class of problems: random Constraint Satisfaction Problems (CSPs), that have n variables, a number of possible values per variable and a set of binary constraints on pairs of variables, meaning variables are supposed to have different values [74]. Graph Coloring restricts each of the variables to have the same domain values and the results that are found in this research might be applicable to other CSPs as well. To reduce the space of possible problem instances we restrict ourselves to Graph 3-Coloring.

First a problem description will be given, then the problem instances that are used in the experiments will be discussed and in the last sections the traditional algorithms that are important in this report are described.

5.1 Problem Description

Given an undirected graph $G = (V, E)$, coloring each vertex $v \in V$ with one of three colors so that no two vertices connected by an edge $e \in E$ are colored with the same color is known as the *Graph 3-Coloring* Problem. Several variations exist, like finding the least number of colors that is needed to color the graph, or to find the largest subgraph in G that can be colored with the given number of colors. All of these problems are known to be NP-complete [33], so it is unlikely that a polynomial time algorithm exists that solves any of these problems. And if we assume $P \neq NP$ for the optimization variants, there is an $\epsilon > 0$ such that no polynomial time approximation algorithm for these problems can find a solution that is guaranteed to be within a ratio of $|V|^\epsilon$ of optimal [1]. Still there are many applications like register allocation [10], timetabling [76], scheduling and printed circuit testing [34] and although no good polynomial algorithm exists for general graphs, an algorithm can still exist which has good expected performance for a specific class of graphs.

Existing algorithms for Graph k-coloring are: an $O(n^{0.4})$-approximation algorithm by Blum [8], the simple Greedy algorithm [54], DSatur from Brélaz [9], Iterated Greedy(IG) from Culberson [14], XRLF from Johnson *et al.* [47].

Both XRLF and DSatur (and some variants) are often considered the best existing algorithms. In [47] it was reported that XRLF finds sometimes better solutions than DSatur, but takes far more time and that a clear winner between the two was not found, though this analysis was for finding the minimal k in random graphs. Only DSatur was implemented of the two because we only want to compare EAs with a good existing algorithm, no clear winner showed up between DSatur and XRLF, DSatur can easily be extended to a backtrack version (making a comparison with the EA more fair) and DSatur has shorter running time. IG was also implemented as it will be used in a combination with the EA.

5.2 Problem Instances

In the literature do not exist so many benchmark 3-colorable graphs and therefore new graphs are created with the graph generator[1] written by Joe Culberson [14]. This generator creates various classes of k-colorable quasi-random graphs. The classes that will be examined are the following:

- *arbitrary 3-colorable graphs*, where vertices are randomly assigned one of the 3 colors uniformly and independently. This is a class that has widely been investigated.
- *equi-partite 3-colorable graphs*, where the 3 partitions are as nearly equal in size as possible (the smallest partition's size being one less than the largest's). Because the partitions are almost equal in size, an algorithm has less information to make use of.
- *flat 3-colorable graphs*, where also the variation in degree (number of neighbors) for each node is kept to a minimum, so even less information is available to the algorithm than for equi-partite graphs.

For the first two classes of graphs, once the pre-partitioning in three color sets has been done, a vertex pair v, w is assigned an edge with fixed probability p, provided that v and w do not have the same color. So there is some variation in the degree for each vertex. p is called the *edge connectivity* of a graph instance. The generator creates a graph $G_{n,p,s}$ for a specific graph class with parameters n (number of nodes), p (edge connectivity) and s (the seed for the random generator). More information on these classes can be found in [14].

Because experiments on all of the three classes is too time consuming (sometimes a couple of days), only the equi-partite class of graphs will be used in the experiments that optimize the EA, though all three classes will be used in a final comparison between the best EA and DSatur.

[1]source code in C is available via ftp://ftp.cs.ualberta.ca/pub/joe/GraphGenerator/generate.tar.gz

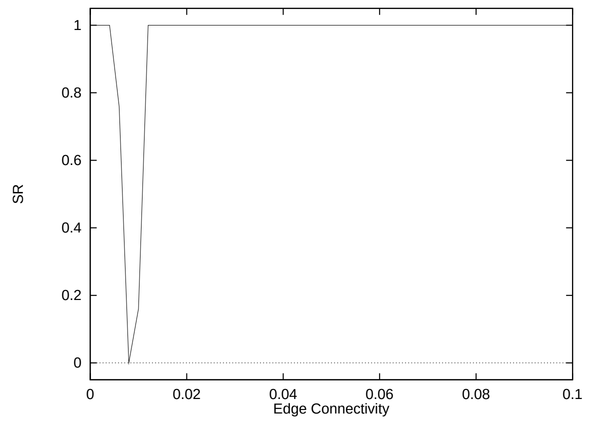


Figure 5.1: Phase transition for DSatur for $n = 1000$.

5.3 The Phase Transition

It is known that for many NP-complete problems, typical instances can be very easy to solve. Turner found that many k-colorable graphs are easy to color [75]. Because any algorithm can color them easily, comparisons between algorithms based on these graphs are not very meaningful and we should look for harder problem instances that pose some challenges for candidate algorithms to overcome. Cheeseman *et al.* [63][2] found that NP-complete problems have an "order" parameter and that the hard problems occur at a critical value of this parameter, called the *phase transition*. This phase transition will be at the point where the problem changes from underconstrained to overconstrained. When the problem is underconstrained, many possible solutions will exist so that almost any algorithm will find one of them. When the problem is overconstrained, almost all paths will be dead ends and the algorithm will soon have tried all possible paths. Also in the case that a solution is always generated in an instance, most algorithms will easily find this solution among the many dead ends. At the phase transition it will be much harder to find solutions, because many local optima will exist. Also a second phase transition exists with extremely hard instances but which is much harder to notice because only a few problem instances will be extremely hard and many instances have to be generated to find them.

Hogg did a lot of research about the phase transition and developed some theory around it [11]. They used Walsh polynomials to model an arbitrary CSP and to predict the cost function for simple backtrack search. Then they used this function to predict where its peak will be, giving the following formula:

$$\nu_{crit} = \frac{-\ln 2}{2^{k-1}\ln\left(1-2^{-k}\right)} \cdot N$$

[2]Cheeseman *et al.* did their experiments on *reduced* problem instances. Because one does not know much anymore about the problem instance after reducing, the analysis is much harder and therefore no reduced graphs are used in this research.

where N is the number (in bits) of variables in a search state, k is the size of the constraints (order of Walsh functions) and ν_{crit} the number of distinct constraints and terms in the Walsh polynomial. They empirically found that this formula describes the behavior of a variety of other search methods as well and so it can be used to derive another formula that gives the location of the phase transition expressed in p for equi-partite graphs.

If a variable can take $|D|$ values, then per variable $\lceil {}^2\log |D| \rceil$ bits are needed. If we use binary constraints, then $k = 2 \cdot \lceil {}^2\log |D| \rceil$. So for Graph 3-Coloring, $k = 4$ and $N = 2 \cdot n$ bits (if n is the number of nodes) and therefore $\nu_{crit} \approx 2.68 \cdot n$. Now if we divide ν_{crit} by the total number of possible distinct constraints, we get p. The total number of possible constraints (or edges) in an equi-partite graph is $\frac{n}{3} \cdot \frac{2n}{3} + \frac{n}{3} \cdot \frac{n}{3} = \frac{n^2}{3}$, so $p = \frac{\nu_{crit}}{n^2/3} \approx \frac{8}{n}$. This formula will not give the exact location, but can be used to get the approximate location of the phase transition for different n and seems to work quite well in practice, although empirically it was found that sometimes instances with $p = \frac{7}{n}$ or $p = \frac{7.5}{n}$ were closer to the phase transition as they were harder. In figure 5.1, the success rate for instances near the phase transition is plotted.

5.4 Traditional Algorithms

5.4.1 The Greedy Algorithm

A simple well known algorithm for many problems is the greedy algorithm, which for Graph Coloring implies:

> For some ordering of the nodes of a graph, color the nodes in this order with the minimum color (for some ordering) that is possible without violating constraints.

This ordering of the nodes often is random, but some heuristics can be used to determine the ordering, which is the case for DSatur. Grimmet and McDiarmid [43] have shown that for almost all random graphs (in the usual model), the greedy algorithm uses no more than about twice the optimal number of colors. Several studies showed that in practice and in theory, the greedy algorithm performs poorly [75][54] and for our Graph Coloring problem where we only allow, 3 colors, the greedy algorithm with random permutations is only useful for very easy problem instances, far away from the phase transition. Some empirical observations showed that indeed the algorithm performs very poorly, even quite far from the phase transition and so it will not be included in the comparison. However, the Greedy Algorithm is as a basis for DSatur, IG and the EA with order-based representation.

5.4.2 DSatur

DSatur from Brélaz [9] uses some heuristics to dynamically create an ordering of the nodes and then greedily colors the nodes as follows:

- node with highest *saturation degree* (number of differently colored neighbors) is selected and given the smallest possible color.
- in case of a tie, the node with highest *degree* (number of neighbors) in the uncolored subgraph is selected.
- in case of a another tie, a random node is selected.

Because of the random tie breaking, DSatur becomes a stochastic algorithm and just like the EA, we need to do several runs to obtain useful statistics.

The datastructure of Turner [75] who proposed heaps to dynamically keep the nodes ordered by saturation degree has been implemented[3]. Because we want to do fair comparison between the EA and DSatur, we have to allow DSatur to continue for a number of search steps. In this research this is done by creating a backtrack version of DSatur by backtracking to the lastly evaluated node that still has available colors to try. Often, backtracking is added to DSatur as it combines well with the dynamic datastructures. Now a search step is defined as the expanding of a node with a new color. Note that this is not equal to creating a new complete coloring. Thus the search step differs from the one used for EAs.

The DSatur heuristics perform very well: far from the phase transition no backtracking is used at all and solutions are found very quickly. Closer to the phase transition, the algorithm needs the backtracking but still performs remarkably well though it has problems when the graph instances get larger ($n = 1000$). So it is for these instances that EAs and other time consuming methods get a chance.

5.4.3 Iterated Greedy

Joe Culberson's Iterated Greedy (IG) [14], makes use of the greedy algorithm described earlier and tries to find the minimum number of colors to color a graph, so a valid solution always exists. It starts with some (random) permutation and on each iteration, it uses the coloring that is found by the greedy algorithm to generate a new permutation which is constrained to have vertices with the same color adjacent in the permutation. This ensures

> "that the new coloring will use no more colors than the previous coloring. Assuming that a coloring has been generated using the greedy coloring, it follows that for colors $c2 > c1$ every vertex with color $c2$ is joined by an edge to some vertex of color $c1$. However, the converse is not true. If the vertices are rearranged so that the vertices of color $c2$ precede those of color $c1$ and the greedy algorithm is applied, it is possible that some of the vertices from the $c1$ set will receive the same color as those in $c2$. An accumulation of such partition changes may result in a reduced coloring."

[3]Although the special search structure that efficiently stores the available colors for each node, is omitted because the algorithm is only used for $k = 3$.

He uses several heuristics to rearrange the vertices given a coloring of a graph. Assume that the current coloring uses k colors. The vertices of each color set are placed as a group and within each group in the order of the previous permutation.

- **Reverse:** Place the vertices of i^{th} color set as group $k - i + 1$.
- **Random:** Place the groups in random order.
- **Largest First:** Place the groups in order of decreasing size.

Sets of equal size are ordered by decreasing color class. He reports that "the ratio 50:50:30 for **Largest:Reverse:Random** is just as good as any setting over a wide range of graphs from various classes". Each iteration of the algorithm is (naturally) considered to be a search step. Instead of initializing IG with a random permutation, as Culberson suggests, it might be advantageous to initialize IG with a permutation that is better than the average random permutation and DSatur (without backtracking) could be used for this. He reported that the effect of adding DSatur to IG is not clear. And as this was found here as well, random initial permutations will be used for IG.

Chapter 6

Evolutionary Algorithms on Graph 3-Coloring

This chapter contains the major part of the work and results in this research. The SSEA, that is described in chapter 3, will be compared with the DSatur algorithm for the Graph 3-Coloring CSP, which was described in the previous chapter. In section 6.1, an EA will be presented that uses the integer representation and for which good parameter settings will be obtained. In section 6.2, an EA with order-based representation will be presented. Then the algorithms will be compared and improvements to the EA will be suggested and tested and finally a big comparison between the best EA and DSatur will be done in section 6.7. To get an idea about the scalability of the EA and its parameter settings for most experiments, tests for $n = 200$ and $n = 1000$ have been done. The results are averaged over 100 runs for $n = 200$ and 25 runs for $n = 1000$ when not stated otherwise. Because of the time consuming nature of the experiments, mostly only tests for one graph instance were done for each n. When there was reason to doubt whether the results generalize to other instances of the same class, sometimes extra experiments were done for other instances.

6.1 EA with Integer Representation

With *integer representation*, each gene of an individual represents a node in the graph and its value can be one of the three colors. The corresponding fitness of an individual is the number of *constraints* that are violated in its coloring. Here we see each *edge* as a constraint, constraining the nodes that it connects to have different colors. For zero fitness, no constraints are violated and a solution is found. So we deal with a minimization problem.

Another option for the fitness of an individual would be to count the number of *nodes* that violate constraints. Because it just ignores a lot of information, it is expected to perform worse. As this was also observed empirically, the fitness function that counts the number of violated constraints will be used.

Operator	SR	AFLE	SFLE	AES	SES	EFE$_\alpha$	ATE	ET$_\alpha$
1-point	0.50	53	54	103655	41420	646290	0.30	191
2-point	0.49	50	52	99497	48693	682816	0.29	200
uniform	0.59	42	52	92554	52667	485170	0.31	151
HUX	0.53	45	51	85039	48259	478040	0.30	146
DIAG 2-10P	0.52	47	51	84650	43574	512113	0.43	220
DIAG 11-20P	0.58	41	50	75539	44584	379729	0.45	171
DIAG 21-30P	0.58	40	50	74987	45209	373644	0.46	172
DIAG 31-40P	0.61	30	52	70658	44366	330541	0.48	159
SCAN 2-10P	0.60	38	49	70503	44717	328895	0.63	207
SCAN 11-20P	0.61	38	49	68236	45005	314412	0.68	214
SCAN 21-30P	0.61	37	48	70692	43804	331617	0.72	239
SCAN 31-40P	0.63	36	48	66051	40308	301627	0.76	229
1-9point	0.53	47	52	92462	44378	549721	0.31	170
10-19point	0.58	42	50	86204	45041	454830	0.32	146
20-29point	0.55	43	50	82431	43655	453264	0.33	150
30-39point	0.56	42	50	81050	44959	436713	0.35	153

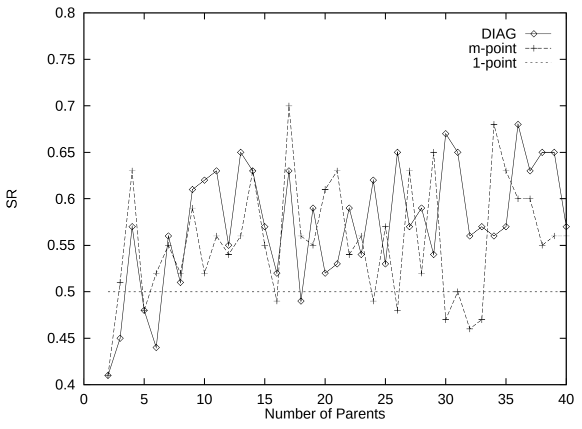
Table 6.1: Performance of recombination operators for $G_{eq,n=200,p=0.08,s=5}$, $\mu = 200$, $T_{max} = 200.000$ and no IP.

A totally different representation, is to let the EA find some input for a deterministic hillclimber and to evaluate the output of this algorithm with one of the above methods for example. This representation will be described later in section 6.2, because more domain knowledge is needed to find a good hillclimber and first the EA with the integer representation, that uses a minimum of domain knowledge, is tested.

In the next sections, the best operators and their optimal population size will be examined and the Incest Prevention mechanism will be tested. A first try with the EA (without optimized parameters) at the phase transition for $n = 200$ gave negative results and no solutions were found at all. Another try for $n = 200$ and $p = 0.5$, gave a success rate of 100% for all considered operators, indicating that this region is too easy. To get some usable results to compare different EA settings, graph instances close to the phase transition will be used.

6.1.1 Comparing the Operators

In this section will be examined how the recombination operators with multiple parents (SCAN and DIAG, which are described in section 3.6.1) behave when the number of parents is varied and how they compare with the standard two parent operators. As all use an unoptimized population size of 200, we can only make a comparison under this assumption.



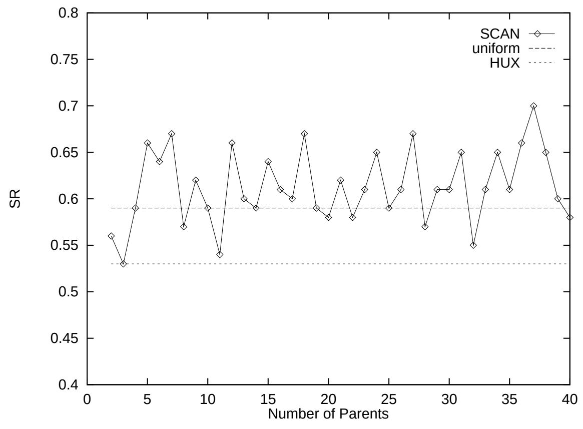


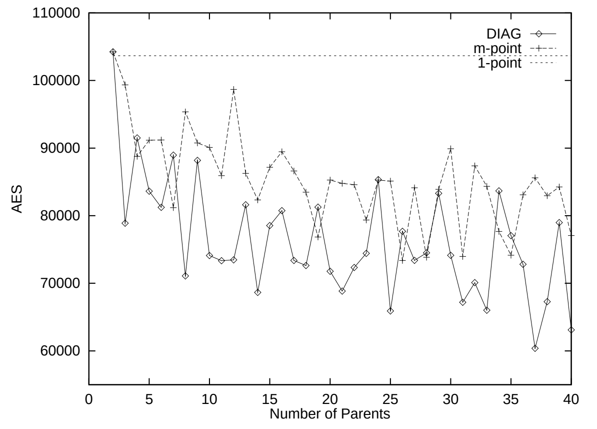
Figure 6.1: Effect of multiple parents on SR for $n = 200$

The results shown in table 6.1, are averaged over groups of parents for operators with multiple parents. See chapter 4 or appendix A for explanations of the abbreviations. A first observation concerning T_{max} is that, using the statistical test described in section 3.9.2, the numbers for AES and SES do not indicate that a maximum of 200.000 evaluations is too small (though for 1- and 2-point and uniform it could be too small), although still more evaluations would possibly have a positive effect[1]. It can be seen that the SCAN-operator has best SR and AES everywhere, though for a high number of parents the difference with DIAG is small. To get a better idea, some of the measures have been plotted in figures 6.1 to 6.4[2]. The following observations can be made for measures SR, AES and EFE_α:

- For SCAN, using more than four or five parents does not give any increase in performance anymore.
- For DIAG, using up to 15 to 20 parents gives a big increase in performance. After that the performance keeps slightly increasing up to about 35 parents.
- Also for m-point, using more than about 15 crossover points does not improve the performance much anymore.
- Using extra parents seems to be advantageous (up to a limit of parents)! And using multiple parents seems to be better than multiple crossover points when comparing DIAG and m-point for SR and AES.

[1]With hindsight it would probably be better first to do a really big run for some operators to get good information about AES and SES and to be able to make a better guess about the optimal maximum number of evaluations.

[2]For m-point the number of parents in the graphs is one less than the number shown to able to compare better with DIAG



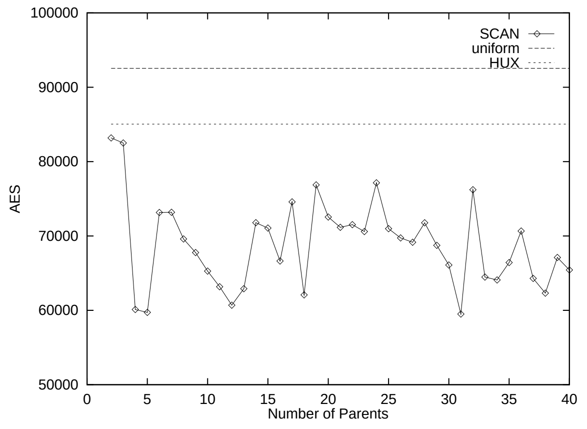
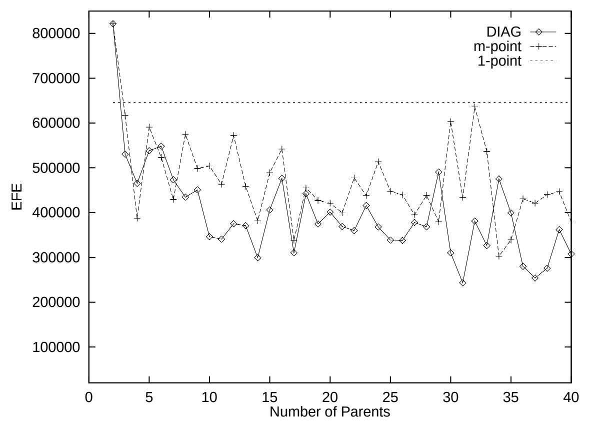


Figure 6.2: Effect of multiple parents on AES for $n = 200$



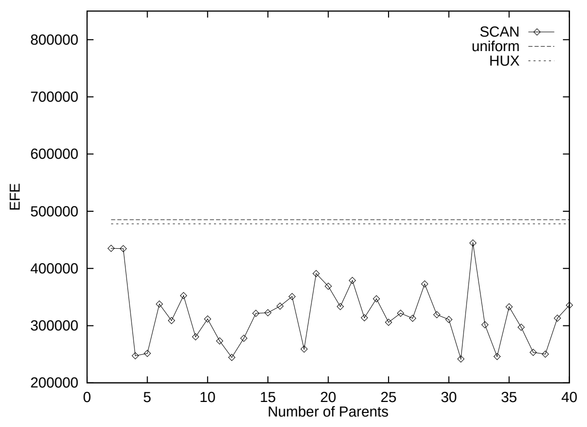
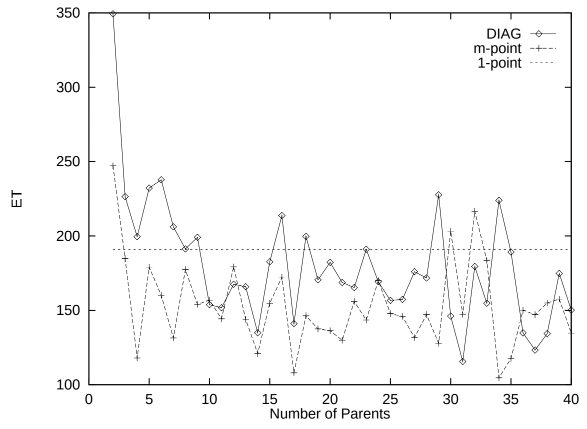


Figure 6.3: Effect of multiple parents on EFE_α for $n = 200$

- AES decreases when more parents are used. At first this may seem counterintuitive because more parents give more diversity, later convergence and therefore the solutions are expected to be found later (which would give a *higher* AES). This is true: the solutions that were found before convergence will now be found later, but because of the importance of mutation, still a lot of solutions are found after converging. Now probably a lot of these solutions will be found earlier, because of the delayed convergence, and will make AES decrease.

- The performance of DIAG for more than about 15 parents and for SCAN everywhere is better than that of uniform, 1-point, 2-point crossover and m-point crossover. So the multiple parent operators perform better than the classical ones, with SCAN being the best of the two.



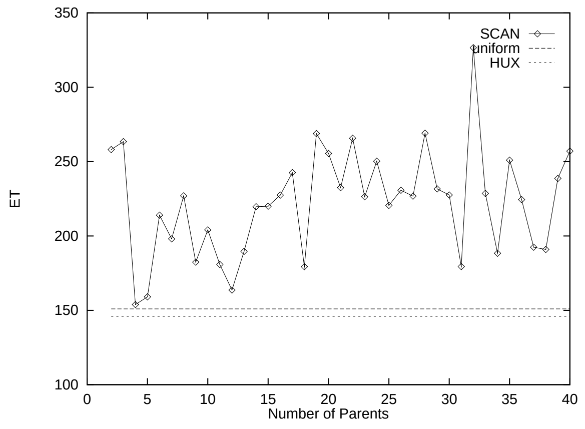


Figure 6.4: Effect of multiple parents on ET_α for $n = 200$

- HUX does not really perform better than uniform crossover.
- The ruggedness of the curves (though maybe a little exaggerated by the scale of the y-axis) indicates that the number of runs is not enough. The difference in performance between m-point and $(m + 1)$-point and DIAGm and DIAG$(m + 1)$ can be as much as 14%! We should keep this in mind when comparing performance.

What is important in practice is how long has to be waited to find a solution and when we look at the CPU-time involved with the operators (ATE and ET_α), the picture gets different:

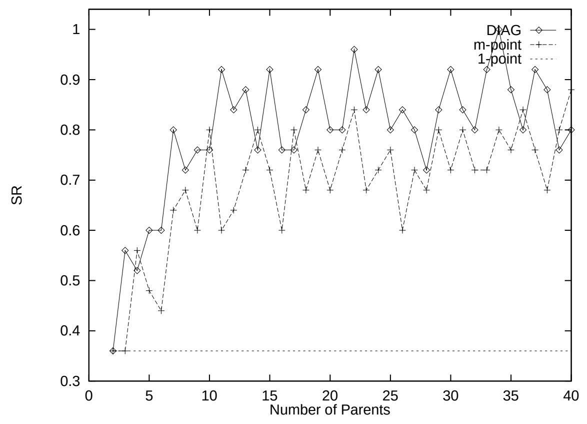
- Still 1-and 2-point have worst results, though the differences get much smaller because 1- and 2-point are the fastest operators. Intuitively this can be expected somewhat because to some extent the speed and the performance are interchangeable for an operator.
- The other operators have almost equal performance, though perhaps m-point (for more than about 15 parents) is the best. SCAN shows to be very costly.

Although these results are consistent for other instances of size $n = 200$, we have to see how everything scales up and therefore experiments for $n = 1000$ have been done for which the results are shown in table 6.2. Concerning T_{max}, AES and SES do not guarantee us that 250.000 evaluations are enough and probably less than 96% of the solutions that would be found with an infinite number of evaluations is found (only for SCAN the number of evaluations seems enough). Though what is most important now is the number of parents and we can draw some conclusions about that. Again the results are also plotted in figures 6.5 to 6.8. The following conclusions can be made for $n = 1000$:

- SCAN has similar effect to the number of parents as for $n = 200$ and therefore scales up quite nicely.

Operator	SR	AFLE	SFLE	AES	SES	EFE$_\alpha$	ATE	ET$_\alpha$
1-point	0.36	510	636	222121	20374	1482480	0.65	964
2-point	0.36	349	513	210578	27670	1431852	0.65	931
uniform	0.84	119	415	188734	33947	445908	0.80	357
HUX	0.84	141	399	171066	40164	444440	0.78	347
DIAG 2-10P	0.63	365	609	190966	30007	707899	1.69	1196
DIAG 11-20P	0.84	163	419	163282	35282	358630	1.93	692
DIAG 21-30P	0.84	170	452	157311	40262	357992	2.10	752
DIAG 31-40P	0.86	175	440	152295	39624	351822	2.39	841
SCAN 2-10P	0.88	110	348	143015	38132	289719	2.21	640
SCAN 11-20P	0.92	100	318	138141	40945	265027	2.39	633
SCAN 21-30P	0.92	74	253	131973	38115	239444	2.51	601
SCAN 31-40P	0.90	127	371	132569	41313	292572	2.57	752
1-9point	0.55	378	577	200299	27182	876468	0.70	614
10-19point	0.70	269	522	182722	34034	556655	0.74	412
20-29point	0.73	244	515	184252	35928	535544	0.76	407
30-39point	0.78	190	458	180715	35699	473285	0.79	374

Table 6.2: Performance of recombination operators for $G_{eq,n=1000,p=0.030,s=5}$, $\mu = 200$, $T_{max} = 250.000$ and no IP.



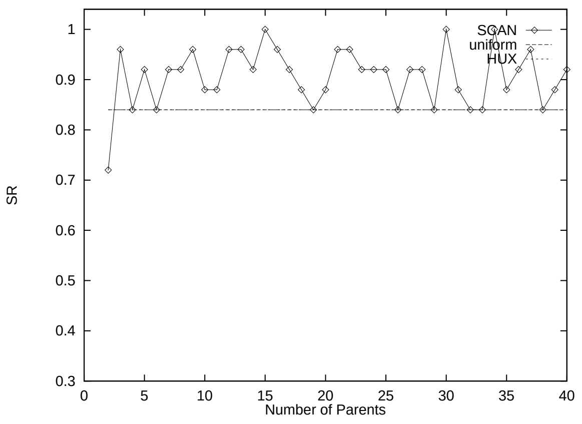


Figure 6.5: Effect of multiple parents on SR for $n = 1000$.

- Again DIAG benefits a lot from extra parents up to 15 to 20 parents. After that still a slight increase in performance is seen up to about 35 parents which the same as for $n = 200$. With about 35 parents DIAG comes close to the performance of SCAN, though SCAN is somewhat better for this bigger instance.

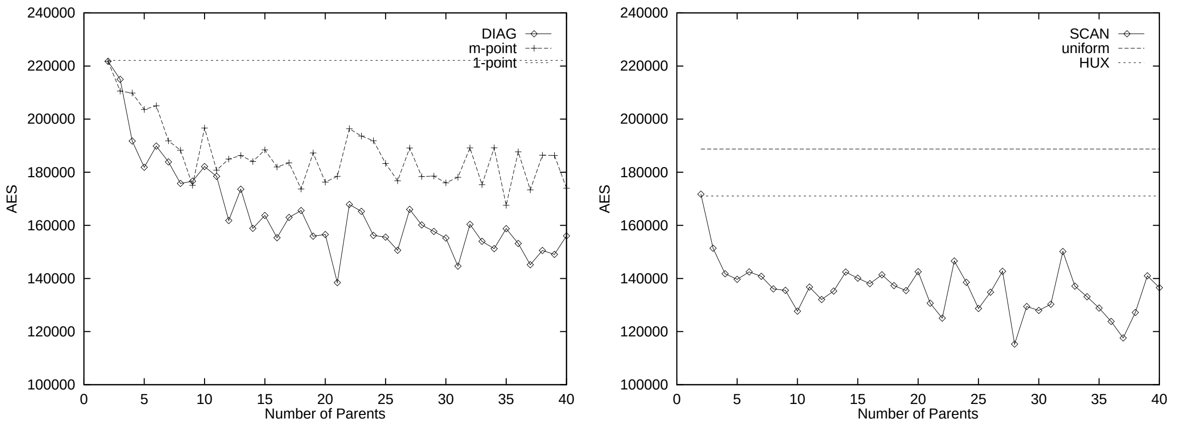


Figure 6.6: Effect of multiple parents on AES for $n = 1000$.

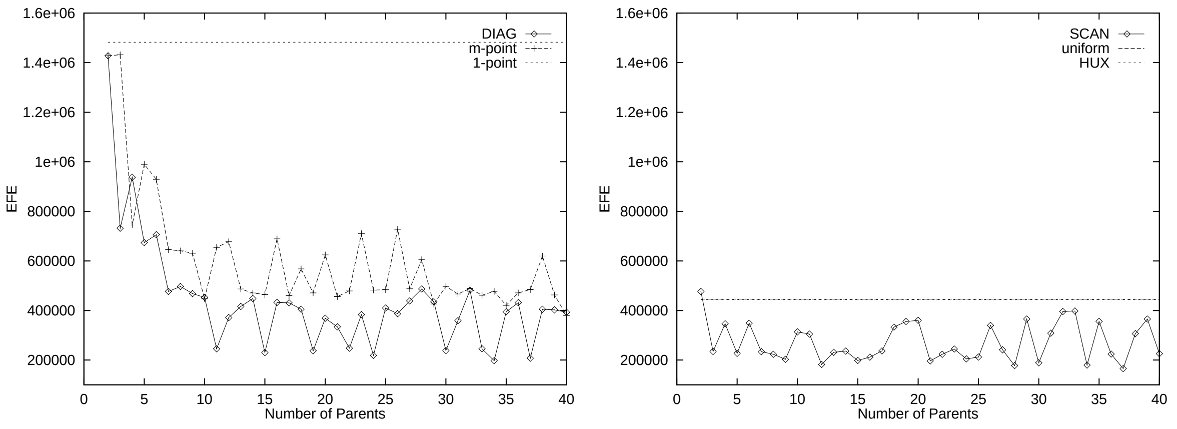


Figure 6.7: Effect of multiple parents on EFE$_\alpha$ for $n = 1000$.

- Also for m-point it is advantageous to use more cross overpoints now and for up to about 20 crossover points, the performance keeps increasing.
- Using more parents still seems to be better than using more crossover points.
- Classical uniform crossover still performs quite good if we only consider SR. If we also look at AES then SCAN and DIAG perform better, with SCAN the best operator.
- The classical 1- and 2-point crossover are not very really competitive with the other operators and therefore will not be used anymore.
- If we also consider the times involved with each operator, the picture gets quite different again and now m-point and uniform crossover are the best, showing that in

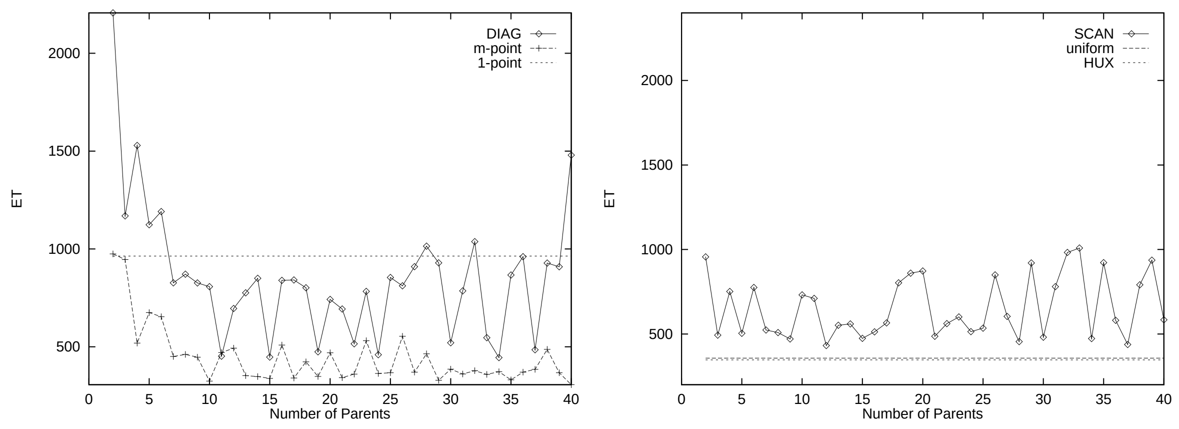


Figure 6.8: Effect of multiple parents on ET_α for $n = 1000$.

a practical situation it might be better to select an operator with lower SR.

- The conclusions for $n = 1000$ are quite similar to those for $n = 200$ and one could say that the EA is quite robust concerning scalability.

Because using more parents makes an operator more expensive, in further tests, the the least possible number of parents that still gives best performance will be used. For DIAG this is about 35 parents and for SCAN this is about 5 parents. For m-point crossover, using more crossover points is not too costly and therefore 34 crossover points will be used to be able to compare it with diagonal crossover with 35 parents.

Probably uniform crossover is superior to 1- and 2-point for this problem and representation because no building blocks are present in the individual as the position of a gene is totally irrelevant. Uniform does not try to preserve building blocks with high order and is therefore better suited to this representation. Though if we start using more parents, DIAG, a more general version of 1-point, will have more crossover points which makes it more disruptive and therefore probably better suited to this representation. For m-point, which is another generalization of 1- and 2-point, the same holds. Interesting is that SCAN, a generalization of uniform crossover, is not able to benefit from many parents.

So to conclude this section, SCAN with 5 parents, DIAG with 35 parents, 34-point crossover and uniform crossover, seem to be the most promising operators and will be used in the next sections.

6.1.2 Preventing Incest

As table 6.3 shows, the results with incest prevention (IP, see section 3.10) were spectacular for the two parent operators (though ATE increased) and obviously the IP mechanism and its underlying idea work. AES also decreased when IP was used.

	$n = 200$		$n = 1000$	
	SR	ATE	SR	ATE
Uniform	0.54	0.31	0.52	0.56
Uniform + IP	0.84	0.47	0.84	1.25
34-point	0.63	0.36	0.56	0.64
34-point + IP	0.84	0.49	0.80	1.00

Table 6.3: Effect of IP on uniform and 34-point with $\mu = 200$ for $G_{eq,n=200,p=0.080,s=5}$, $T_{max} = 200.000$ and for $G_{eq,n=1000,p=0.025,s=5}$, $T_{max} = 300.000$.

To see what the effect is for the multiple parent operators SCAN and DIAG, versions with a difference measure based on entropy (see section 3.10) were tested because for discrete values this is a better measure for the diversity than the variance measure. A first version uses a threshold, δ, that determines per gene the minimum diversity and sums the number of genes with enough diversity (IPTE). A second version just sums the diversity over all genes (IPSE).

The entropy for 5 parents that are not all the same, will always lie between 0.94 (2,2,1 distribution, most uniform) and 0.46 (4,1,0 distribution), so values for the threshold between $\delta = 0.94$ and $\delta = 0.46$ should be checked, lower values will have equal SR as for $\delta = 0.46$, higher values will have performance less or equal than with $\delta = 0.94$. The entropy for 35 parents will always lie between 0.99 (12,12,11 distribution, most uniform) and 0.06 (34,1,0 distribution).

To see whether the high number of offspring for DIAG35 makes a difference, also a version with one offspring was tested (DIAG35_{O1}). The effect of the threshold is shown in table 6.4. For DIAG, the effect of the threshold value in IPTE is not very clear and in case of an improvement, the improvement is not so much and not very significant and so the threshold method for DIAG will not be used anymore. Also there does not seem to be a significant difference between DIAG with 35 offspring and one offspring and therefore 35 offspring will be used in further tests as this is faster. For SCAN there does seem to be a clear effect of the threshold value and optimal performance is found with $\delta = 0.4$ or $\delta = 0.0$. $\delta = 0.0$ means that when not all parents have the same value for a gene, the parents are considered diverse enough for this gene.

Table 6.5[3] shows that for SCAN for the entropy measure, the Sum of Entropies (IPSE) is better than the Threshold of Entropies (IPTE). It also shows that IPSE is better than IPSV, that sums the variance over all genes, which is to be expected as the entropy is a better measure for the diversity than the variance.

Table 6.6 shows that IPSE makes SCAN again competitive with uniform and m-point crossover. For DIAG and $n = 200$, IPSE always gives equal or better results whereas for $n = 1000$, IPSE sometimes has a big positive effect and sometimes has a small negative

[3]More seeds were used because of contradictory results of IPSE.

	$n = 200$			$n = 1000$	
	SCAN5	DIAG35	DIAG35$_{O1}$	SCAN5	DIAG35
No IP	0.58	0.40	0.39	0.60	0.68
$\delta = 0.8$	0.58	0.45	0.47	0.68	0.64
$\delta = 0.7$	0.58	0.41	0.50	0.68	0.64
$\delta = 0.6$	0.61	0.46	0.48	0.72	0.64
$\delta = 0.5$	0.61	0.45	0.43	0.72	0.52
$\delta = 0.4$	0.64	0.40	0.42	0.72	0.44
$\delta = 0.3$	0.64	0.45	0.48	0.72	0.64
$\delta = 0.2$	0.64	0.44	0.43	0.72	0.68
$\delta = 0.1$	0.64	0.43	0.40	0.72	0.56
$\delta = 0.0$	0.64	0.43	0.40	0.72	0.56

Table 6.4: Effect of the threshold on SR for SCAN and DIAG for $G_{eq,n=200,p=0.080,s=5}$, $T_{max} = 100.000$ and for $G_{eq,n=1000,p=0.025,s=5}$, $T_{max} = 300.000$.

	$s = 0$	$s = 1$	$s = 2$	$s = 3$	$s = 4$	$s = 17$
SCAN NoIP	0.51	0.55	0.57	0.47	0.63	0.46
SCAN + IPTE ($\delta = 0.0$)	0.63	0.65	0.63	0.68	0.74	0.58
SCAN + IPSE	0.73	0.75	0.74	0.79	0.76	0.68
SCAN + IPSV	0.64	0.63	0.66	0.56	0.80	0.59

Table 6.5: Comparison on SR of different IP versions for SCAN and $G_{eq,n=200,p=0.08,s}$.

	$s = 0$	$s = 1$	$s = 2$	$s = 3$	$s = 4$	$s = 17$	ATE
SCAN ($n = 200$)	0.51	0.55	0.57	0.47	0.63	0.46	0.63
SCAN + IPSE	0.73	0.75	0.74	0.79	0.76	0.68	1.79
SCAN ($n = 1000$)	0.68	0.44	0.52	0.68	0.60	0.60	2.07
SCAN + IPSE	0.76	0.80	0.72	0.88	0.72	0.88	6.73
DIAG ($n = 200$)	0.45	0.50	0.44	0.47	0.61	0.50	0.47
DIAG + IPSE	0.58	0.64	0.58	0.53	0.65	0.50	0.55
DIAG ($n = 1000$)	0.72	0.60	0.64	0.56	0.56	0.52	1.34
DIAG + IPSE	0.88	0.56	0.60	0.52	0.64	0.64	1.98

Table 6.6: Effect of IPSE on SR for DIAG and SCAN for $G_{eq,n=200,p=0.080,s}$ and $G_{eq,n=1000,p=0.025,s}$.

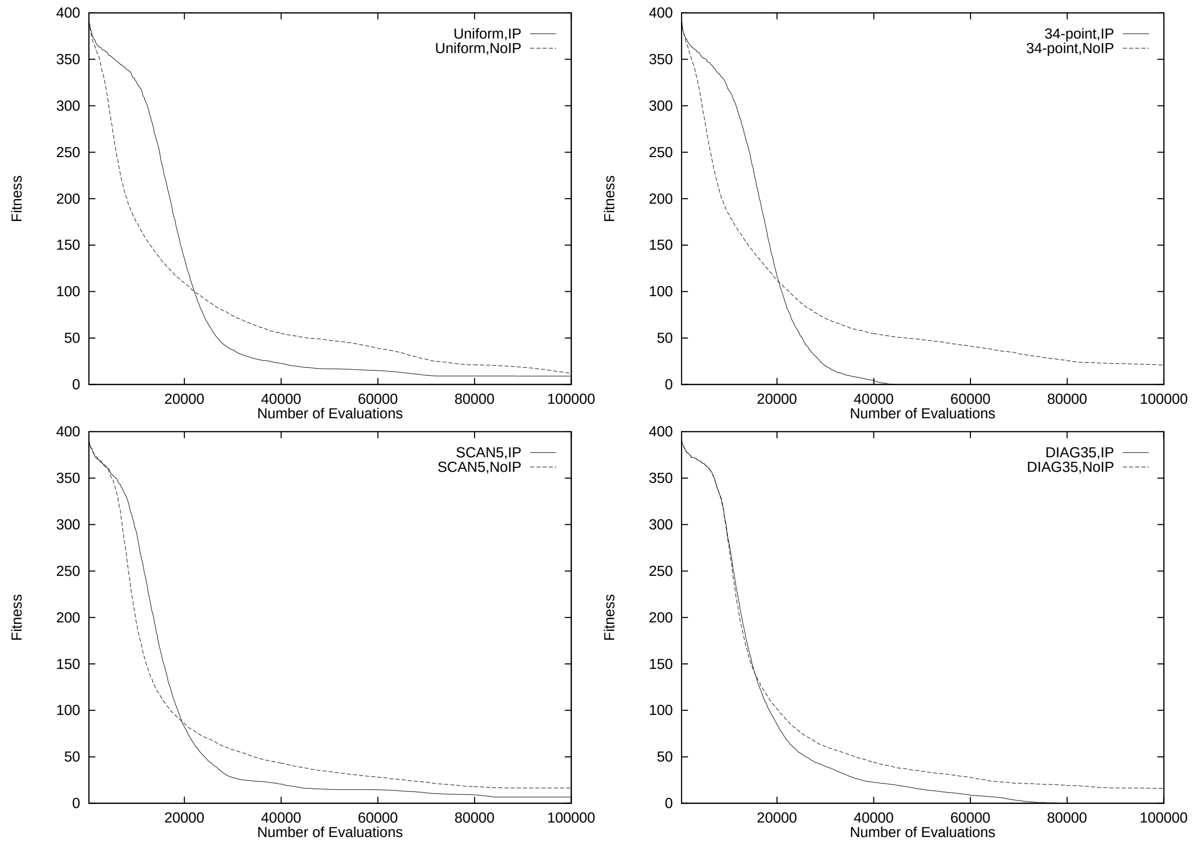


Figure 6.9: Effect of incest prevention on fitness curve averaged over 25 runs.

effect than when not used and therefore or DIAG also the IPSE mechanism will be used.

If we also consider the increase in time because of IP, it might be better in a practical situation not to use IP. For each recombination, all the genes of all parents have to be examined and so the time per recombination will increase with an amount almost proportional to $n \cdot \frac{\#\text{parents}}{\#\text{offspring}}$, which explains why IP costs so much more for SCAN (which takes five parents to produce *one* offspring) than for DIAG. Still IPSE is used for DIAG and SCAN and a decision whether to use it or not is deferred until later.

The lower performance of DIAG compared to SCAN seems strange because we saw in previous statistics that from about 35 parents diagonal crossover has almost equal performance to uniform scanning crossover. But the other seeds show that it's not just a sampling error. Perhaps with 35 parents often the diversity among the parents is already high enough and the incest prevention mechanism does not bring in much more diversity before the parents are recombined.

In figure 6.9 the average fitness of the best individual in the population is plotted against the evaluation number for the four recombination operators with and without in-

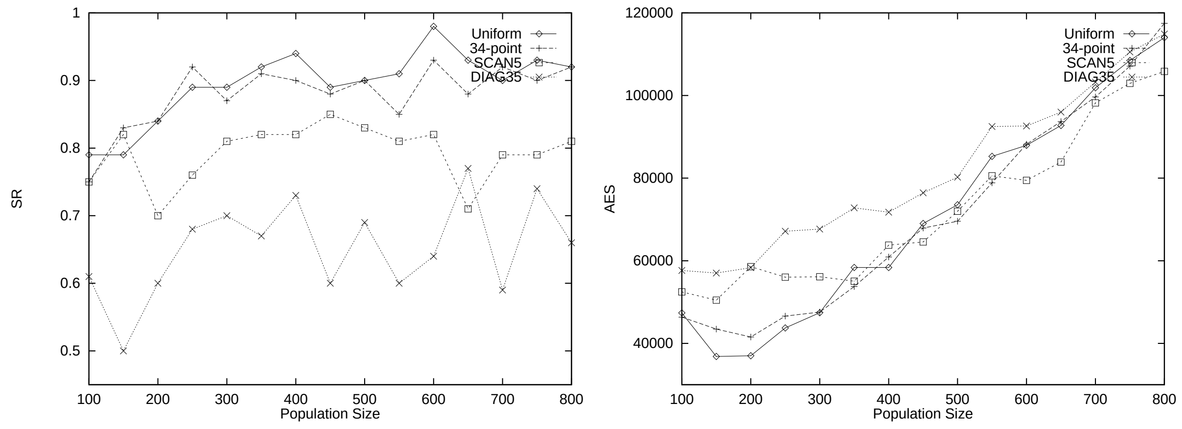


Figure 6.10: Effect of population size on SR and AES for $G_{eq,n=200,p=0.080,s=5}$, normal IP for uniform and 34-point, IPSE for SCAN5 and DIAG35, $T_{max} = 200.000$.

cest prevention. For uniform, 34-point and SCAN5 crossover we can see that the incest prevention mechanism slows down the rate at which the EA finds better individuals (although a bit less for SCAN5), which is of course a side effect of not crossing over parents that are similar. The convergence can be located at the point where the steepness of the curve begins to decrease and we see that for uniform, 34-point and SCAN5 crossover with the incest prevention mechanism, the population converges later what is exactly what we want to achieve. But for DIAG35 the point of convergence does not change when incest prevention is used. Again, this could be because of the 35 parents that already give high enough diversity among the parents so that incest prevention does not add anything. Only after the point of convergence, incest prevention does seem to have some positive effect on the performance of DIAG35.

Another strange phenomenon is that uniform+IP seems to be better than SCAN5+IPSE, but SCAN5 alone is better than uniform alone. Perhaps there is a number of parents smaller than 5 for which the combination of SCAN with IPSE is even better than uniform + IP. Perhaps it's caused by the fact that SCAN produces only one parent and uniform two, but this is left for further study.

So to conclude this section, IP is an effective method to increase SR and the best generalization for multiple parents is IPSE, though especially IPSE can greatly increase ATE and with this decrease its practical usefulness.

6.1.3 Population Size

Up to now the population size was fixed to 200. But increasing the population size often has positive effect on the performance [24] and because we cannot be sure that the change in population size has the same effect on the performance for each operator, the effect of changing the μ will be examined for uniform, SCAN5, DIAG35 and 34-point crossover.

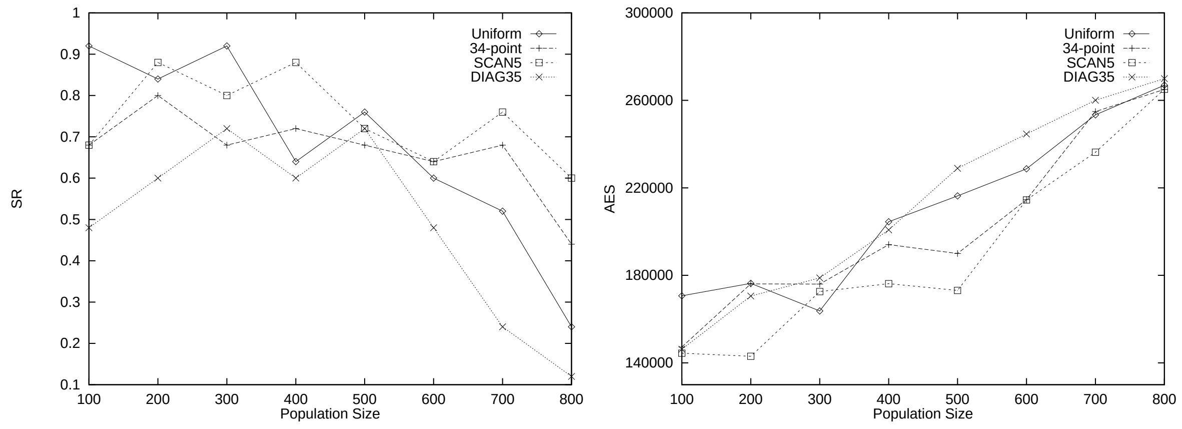


Figure 6.11: Effect of population size on SR and AES for $G_{eq,n=1000,p=0.025,s=5}$, $T_{max} = 300.000$ with IP(SE).

In figure 6.10, which shows the performance for varying population size, SR increases fastly up to $\mu = 250$, slower between $250 \leq \mu \leq 600$ and stops increasing after $\mu = 600$ (500 for DIAG35 and SCAN5). The reason that SR stops increasing is that the maximum number of evaluations is too small for this population size. A higher μ takes more time per evaluation, because of a longer search for removing the worst individual (or for inserting in an ordered list). For $n = 200$, ATE increases from 0.27 for $\mu = 100$ to 1.28 for $\mu = 800$ and for $n = 1000$, ATE increases from 1.15 for $\mu = 100$ to 4.74 for $\mu = 800$. Also for a higher μ more evaluations are needed: AES is increasing monotonically for $\mu \geq 200$. Because of this a population size (for all operators) of 250 is selected to have a higher success rate and still a low AES and low ATE.

Figure 6.11 shows how this scales up to $n = 1000$. It can be seen that for $\mu > 300$, μ gets too large in combination with a maximum of 300.000 evaluations, but if we take in account that evaluating 300.000 individuals for $n = 1000$ already takes about 10 minutes, we do not really want to use much more evaluations. Some experiments with more evaluations prevented the curve from dropping, but still after about 250 no significant increase in SR was obtained.

A reason for the increase in AES probably is the twofold effect of μ (as noted before in section 3.8): it will increase the diversity so that it will take longer for the EA to converge and so more solutions can be found before the EA is converged. Though after converging we rely more on mutation to change the "right" gene of the "right" individual and increasing μ will reduce the probability of selecting the "right" individual. Of course if we took the maximum number of evaluations high enough we would eventually pick the "right" individual and the "right" gene, but this is costly.

Figure 6.11 shows SR increases up to and AES increases fastly after $\mu = 250$ and so for $n = 1000$, $\mu = 250$ also seems to be a good choice for the population size.

6.1.4 Comparison of the Best Recombination Operators

In this section we'll see how good each recombination operator performs near the phase transition, and how it compares with DSatur. DSatur uses the same T_{max} as the EA. Both use about the same amount of time per search step for $n = 200$ and for $n = 1000$, though DSatur needs far less search steps.

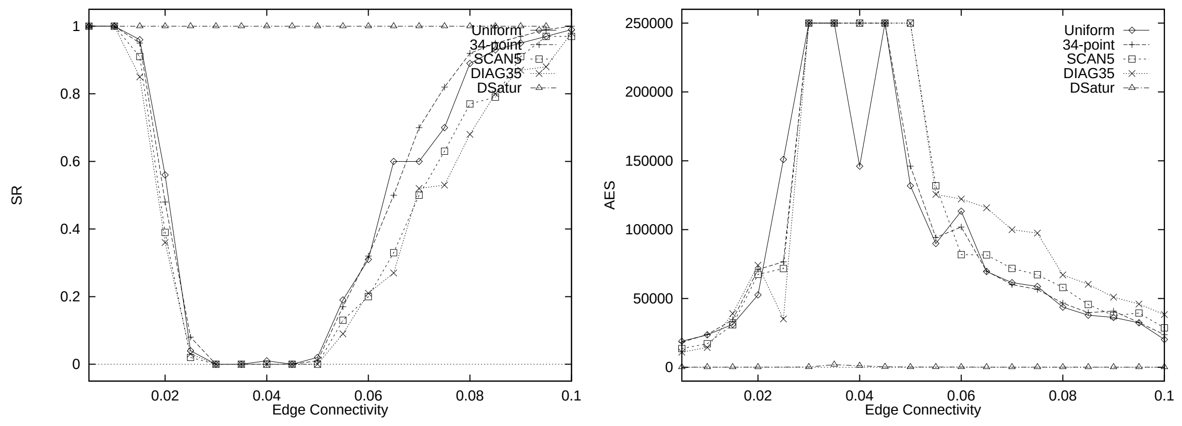


Figure 6.12: Performance of operators near phase transition for $G_{eq,n=200,p,s=5}$, $T_{max} = 250.000$. The EA uses incest prevention and $\mu = 250$.

For $n = 200$, figure 6.12 shows that clearly DSatur is far more efficient than the EA: DSatur only needs a little more than 200 steps at the phase transition and has 100% success everywhere. For the EA most striking is that there's not too much difference between the operators although uniform and 34-point with IP are somewhat better. Because of the difference in performance between the operators is not so big, uniform is fastest and a little better than m-point, for $n = 1000$ only uniform crossover was tested.

For $n = 1000$ in figure 6.13, the problem instances seem to be hard enough to give DSatur problems at the phase transition, though it still outperforms the EA in AES and SR.

6.1.5 Asexual Reproduction

In GAs, traditionally mutation is seen as a "background" operator that only introduces new diversity in the population. Recently however, discussions are going on whether it's better only to use an asexual mutation operator instead of the traditional way of using a recombination operator followed by a mutation operator. At first glance it seems hard to believe that only mutation will be able to find any solutions at all and surely the recombination intuitively seems to be better because it combines found information. But if we realize that in nature one of the most successful organisms (although not very complex), bacteria, reproduce asexually, we should be more open to the idea of just using mutation.

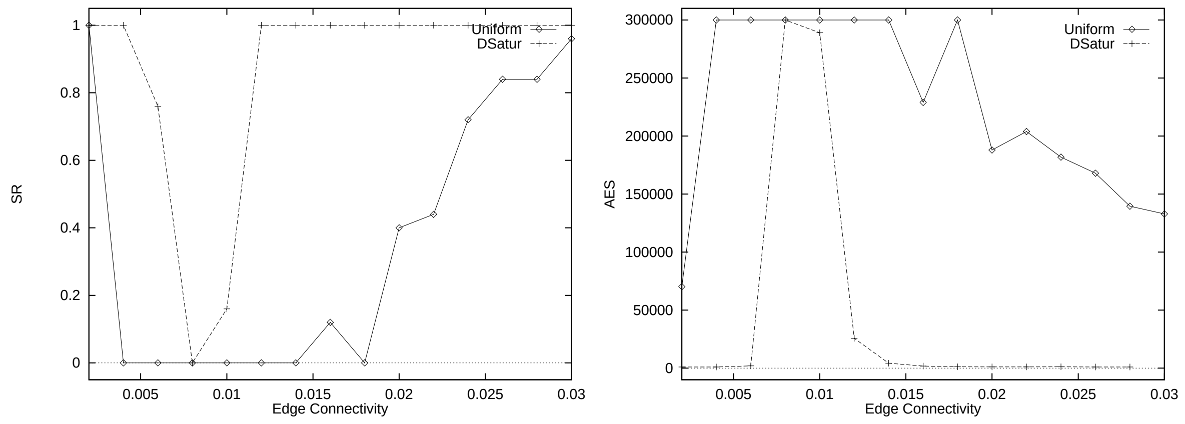


Figure 6.13: Performance of uniform and DSatur near phase transition for $G_{eq,n=1000,p,s=5}$, $T_{max} = 300.000$.

	$p = 0.14$		$p = 0.30$	
	SR	AES	SR	AES
Uniform (no Mutation)	0.0	250000	0.04	3147
Uniform+Mutation	0.18	47145	0.92	8266
Mutation	0.32	77315	0.91	20060
Uniform+Mutation+IP	0.59	15028	0.99	9477
Uniform+Heuristic Mutation	0.34	26693	0.92	4635
Heuristic Mutation	0.39	62488	0.93	5567
Uniform+Heuristic Mutation+IP	0.69	16471	0.99	7748

Table 6.7: Effect of different reproduction schemes for $G_{eq,n=50,p,s=5}$, $T_{max} = 250.000$ and $\mu = 200$.

Biologists even see mutation as the main source for evolutionary change. In [16] several good results of "naïve evolution" (as it is called in [66]) are reported. Eiben *et al.* [24] compared several asexual operators for the Graph 3-Coloring problem with a version that used uniform crossover, a standard mutation and μ fixed to 200 for $n = 50$. They found that the best operator was an asexual operator that mutated a number of genes and used an heuristic to determine the value for each mutated gene. This operator is very similar to the mutation used here combined with an heuristic that simply picks the color that gives best results.

Table 6.7 shows that close to the phase transition, mutation alone is better than uniform crossover and mutation and far from the phase transition the combination of uniform crossover and mutation is better than mutation alone (uniform crossover alone does not

	$n = 200$		$n = 1000$	
	SR	AES	SR	AES
$\mu = 1$	0.17	84803	0.76	171491
$\mu = 2$	0.08	86171	0.24	221857
$\mu = 3$	0.06	85632	0.12	246051
$\mu = 4$	0.04	149972	0.12	229381
$\mu = 5$	0.06	163421	0.04	242962
$\mu = 10$	0.02	210328	0.06	263054
$\mu = 15$	0.01	247573	0.0	300000
$\mu = 20$	0.02	230876	0.0	300000
$\mu = 25$	0.02	155004	0.0	300000

Table 6.8: Effect of population size on mutation only, for $G_{eq,n=200,p=0.05,s=5}$ (closer to the phase transition to see differences better) , $T_{max} = 250.000$ and for $G_{n=1000,p=0.016,s=5}$, $T_{max} = 300.000$.

work good at all). Though if we add IP (which does not work with mutation alone) to uniform crossover, we get the best performance everywhere. But because of the different effect of μ on a mutation operator and on a recombination operator (see section 3.8), we first have to find the optimal μ for mutation alone to be able to do a fair comparison.

Table 6.8 shows that clearly the optimal value for μ is 1 which is the smallest population size that still has some form of selection! Still Goldberg already gives some theory that predicts a really small population size of 3 (which still makes sense in connection with reproduction and crossover according to him) to be optimal for a serial EA [40]. And although figures 6.10 and 6.11 show the advantage of a big population size, for the order-based representation later in this thesis, this really small population size will be tested as well.

Figures 6.14 and 6.15 show the performance of uniform crossover (with mutation) against that of mutation alone. Clearly asexual reproduction is far better for the Graph 3-Coloring problem, which of course could imply that the problem is not complex enough, but as we're testing on an NP-complete problem close to the phase transition this is not the case. Added advantage of using only mutation is that it is about 10 times faster than a combination with uniform crossover (implying that it is also about 10 times as fast per search step as DSatur, though still DSat is faster in total). The question remains whether an EA with only mutation and $\mu = 1$ still remains a EA. Perhaps a better name would be Random Hill-climber, but this will be discussed further at the end of this thesis.

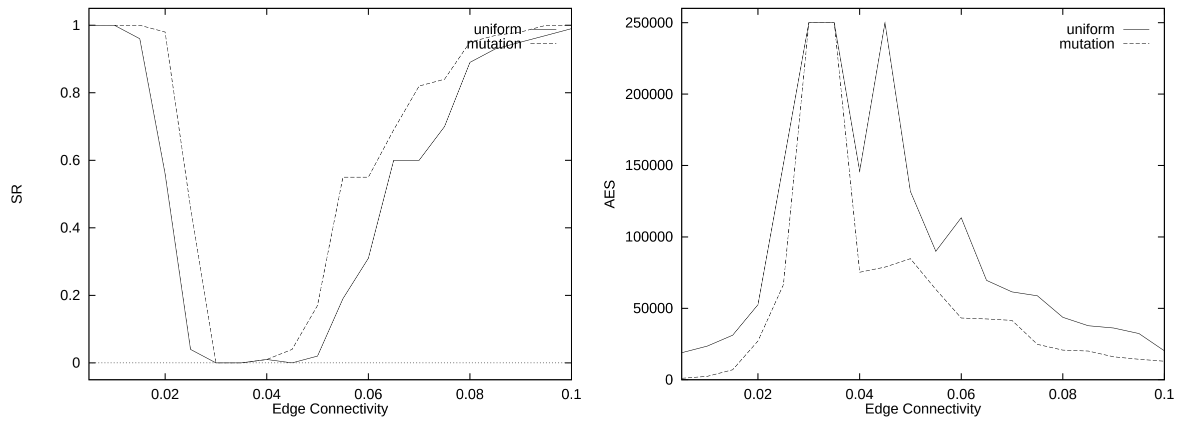


Figure 6.14: Performance of asexual EA against EA with recombination for $G_{eq,n=200,p,s=5}$, $\mu = 250$ for uniform and $\mu = 1$ for mutation alone. Uniform crossover also uses IP.

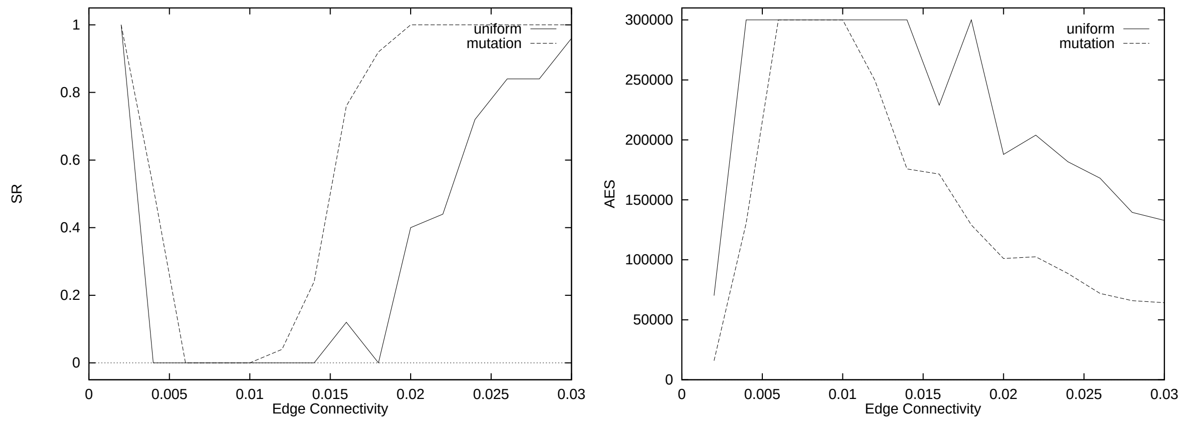


Figure 6.15: Performance of asexual EA against EA with recombination for $n = 1000$.

6.2 EA with Order-based Representation

Because Eiben *et al.* had very good results for the *order-based* representation in [24], although only tested for small problems, this will be experimented with in this section.

In order-based EAs the individuals are permutations and special operators are used to recombine and mutate permutations (see section 3.6) and so the fitness function now has to evaluate permutations. We can use a coloring algorithm as a *decoder* that deterministically colors a graph for a given permutation of the nodes. One such an algorithm is the Greedy Algorithm (see section 5.4.1), which colors a node with the lowest color that does not violate constraints (leaving nodes uncolored when no colors are left[4]).

The Greedy Algorithm that uses random permutations does not work so well, but when information about the order of the nodes is supplied, the performance can increase dramatically (e.g. DSatur). In theory if the right order is given to the Greedy Algorithm, it will color a graph optimally. So now the idea is to let the EA find a good ordering and use the Greedy Algorithm as an intermediate decoder of this ordering, so that after decoding the permutation, the evaluation function can calculate the corresponding fitness value. The most simple way of evaluating a decoded permutation is just to use the number of uncolored nodes[5] as the fitness value and this will be the fitness function that will be used for now.

A characteristic of order-based representation with the specified decoder is that a small change in an individual can have a big effect in the resulting coloring. This can be positive for escaping local minima.

As valid individuals for the EA have to be permutations of length n, the search space for the order-based EA is greater than for the integer representation: $n!$ instead of 3^n. Because at most $3^n/3!$ different possible colorings exist when three colors are used, it is obvious that a lot of the points in the order-based search space will lead to the same coloring and therefore redundancy exists. Both things are often considered harder conditions for a search algorithm. As EAs are designed to work well for big search spaces, they might not be affected so much by this increase in search space.

In all tests that follow in the rest of this chapter, 100 runs with $T_{max} = 250.000$ for $n = 200$ and 25 runs with $T_{max} = 300.000$ for $n = 1000$ will be done, except when stated otherwise.

6.2.1 Comparing the Operators

It seems there are at least two different kind of sequencing problems: TSP-like problems (where the edges are more important) and scheduling-like problems (where the position in the sequence is more important). Both types of problems need different kind of operators and several studies have been done to compare operators on both types of problems ([70], [30] and [56]). For the Greedy Algorithm it is obvious that it is not so important that two nodes are next to each other in a permutation, it is far more important what relative position in the permutation a node has and therefore solving the Graph Coloring problem with the Greedy Algorithm is considered a problem of the second type.

In [70] the most important order-based operators are described and compared and it shows that for scheduling-like problems order crossover #2 (OX2) by Syswerda [72] is the best operator. This can also be seen in the study of Manderick and Spiessens [56] that

[4]Another possibility is to give nodes just any color if no other colors are left so that they constrain future nodes in their colors. Important is that this decoder will color deterministically and uses some fixed criterium to decide on the color. This is left for further research.

[5]As for the integer representation, it is better to count the number of violated constraints, this was also tested for the order-based representation. Surprisingly, this gave far worse results and so was not used. Probably this is due to the vaguer relationship between fitness function and representation.

	$n = 200$		$n = 1000$	
Operators	SR	AES	SR	AES
OX + No Mutation	0.24	60641	-	-
OX + RAR	0.89	77021	0.04	274816
OX + SWAP	1.00	21195	0.16	220271
OX2 + No Mutation	0.88	17858	-	-
OX2 + RAR	1.00	13265	0.88	65131
OX2 + SWAP	1.00	8364	1.00	41614
OX2 + Inversion	1.00	19205	0.36	73312
PMX + SWAP	1.00	18839	0.36	206613
CX + SWAP	1.00	13585	0.76	173590
Uniform + Mutation	0.84	51879	0.84	176351

Table 6.9: Performance of various order-based operators for $G_{eq,n=200,p=0.08,s=5}$ and $G_{eq,n=1000,p=0.025,s=5}$ and $\mu = 200$.

shows that a position-based operator (which is in this research identical to OX2) is the best operator for a scheduling problem. Though both studies analyze the operators in isolation of a mutation operator.

In this section OX, OX2, PMX, CX and SWAP, RAR and Inversion mutation (which are described in section 3.6.1) are examined and compared. Note that the order-based mutations use a similar distribution as the mutation for the integer representation and p_m means the probability *per gene* of performing a SWAP, RAR or Inversion operation on it (with the second gene chosen randomly).

As table 6.9 shows for $\mu = 200$, clearly the use of order-based representation with the right operators outperforms the integer representation, especially the number of evaluations needed to find a solution drops spectacularly (though ATE gets about two times bigger). The best operator combination clearly is OX2 together with SWAP[6]. Interestingly, the analysis of Manderick and Spiessens predicts that RAR instead of SWAP should be the best mutation operator, although it should be noted that their analysis treats mutation without recombination[7], they use the flow shop scheduling problem and they do not give information about the hardness of their test instances.

In the rest of this research, only the the OX2 recombination operator and the SWAP mutation will be used.

[6]Although later results (see section 6.2.3), like for integer representation, showed that SWAP alone with the right value for μ is very good as well.

[7]Though also tests with SWAP and RAR alone showed that SWAP alone performs far better.

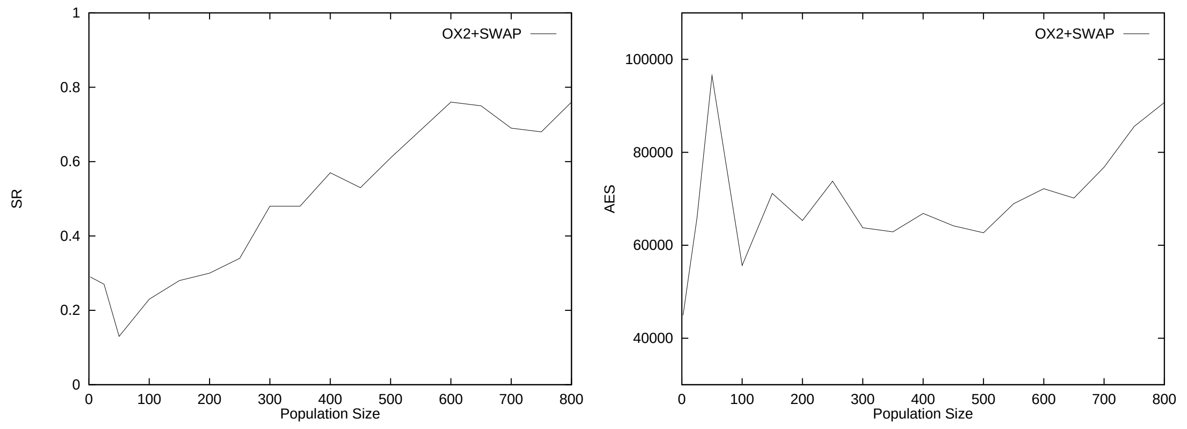


Figure 6.16: Effect of population size on SR and AES for $G_{n=200,p=0.04,s=5}$.

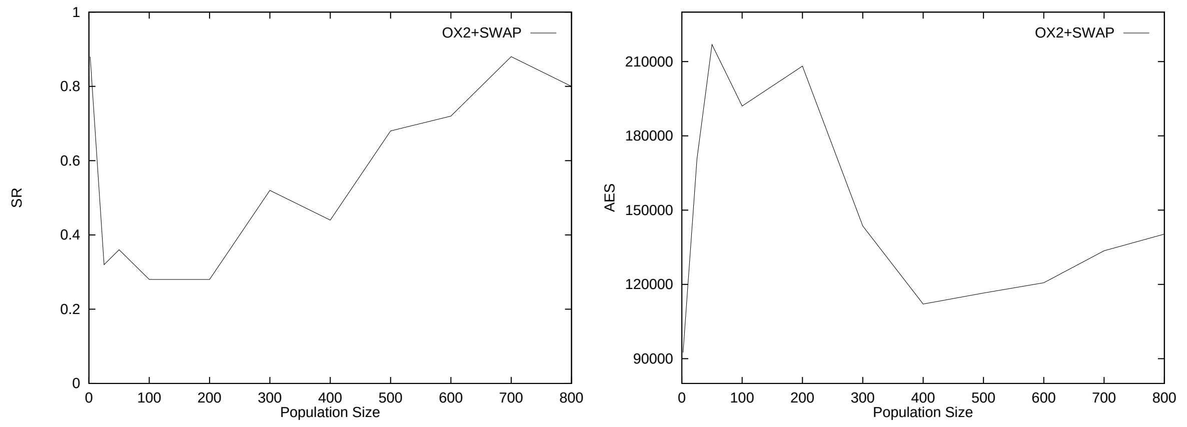


Figure 6.17: Effect of population size on SR and AES for $G_{n=1000,p=0.016,s=5}$.

6.2.2 Population Size for OX2

In this section the optimal population size is searched for OX2 together with SWAP and figure 6.16 shows that for $n = 200$, $\mu = 600$ seems the best choice. It should be noted that in a first test only $\mu > 100$ was tested and if one extrapolates the curve in figure 6.16 this seems reasonable. But because of the results for the asexual EA and because of Goldberg's theory predicting $\mu = 3$ to be optimal [40], also some really small values for μ were tested and this gave some unexpected results. If μ is decreased below $\mu = 50$, the performance starts *increasing* again with optimal performance for $\mu = 2$, which is the smallest possible population size for an EA with recombination. This indicates that below a certain value for μ it is not useful to have a population at all and only above this minimal μ it is useful to have a population. This phenomenon can be seen both for $n = 200$ and $n = 1000$: the

	$n = 200$				$n = 1000$			
	$s = 5$		$s = 0$		$s = 5$		$s = 0$	
μ	SR	AES	SR	AES	SR	AES	SR	AES
1	0.26	79441	0.78	45061	0.92	87425	0.92	82743
2	0.11	114222	0.75	59575	0.72	100848	0.64	108054
3	0.11	94552	0.58	54381	0.48	164233	0.64	147417
4	0.16	104053	0.64	69218	0.64	145030	0.56	158897
5	0.15	121065	0.59	94299	0.24	181856	0.52	146200
10	0.04	121438	0.41	101268	0.20	155116	0.32	197867
15	0.05	155686	0.32	108714	0.16	154075	0.28	224129
20	0.04	78313	-	-	-	-	-	-
25	0.03	144522	-	-	-	-	-	-

Table 6.10: Effect of population size on mutation only for $G_{n=200,p=0.035,s}$ and $G_{n=1000,p=0.014,s}$. Because of some unexpected results for $s = 5$ also results for $s = 0$ are shown.

minimum μ is about 50 for $n = 200$ and about 200 for $n = 1000$. A reason can be that when μ is too small, the EA will converge too early and will rely heavily on mutation after converging and we saw for the integer representation that for mutation alone a very low value for μ is best.

Figure 6.17 shows that for $n = 1000$, the performance for the EA with $\mu = 700$ is similar to that with $\mu = 2$, but for $n = 200$, the performance with $\mu = 600$ is far better then that with $\mu = 2$, showing a population size is useful here! $\mu = 600$ and $\mu = 700$ will be used for resp. $n = 200$ and $n = 1000$ for the order-based EA with recombination from now on.

6.2.3 Asexual Reproduction

Because the results for the integer representation showed that asexual reproduction was best, SWAP[8] alone was tested as well to see if this conclusion also holds for a different representation.

Table 6.10 shows that without doubt we can conclude that again $\mu = 1$ is optimal both for $n = 200$ and $n = 1000$, so that for an asexual (serial) EA a minimal population is most efficient.

Just like for the asexual EA, in Evolution Strategies (ES) [67] mutation is the main operator, the representation is real-valued and deterministic selection is used, where in a (pure) GA, crossover is the main operator, the representation is binary-valued and probabilistic selection is used. If we realize this, then the EA with only SWAP perhaps could

[8]RAR alone was tried as well but showed inferior results.

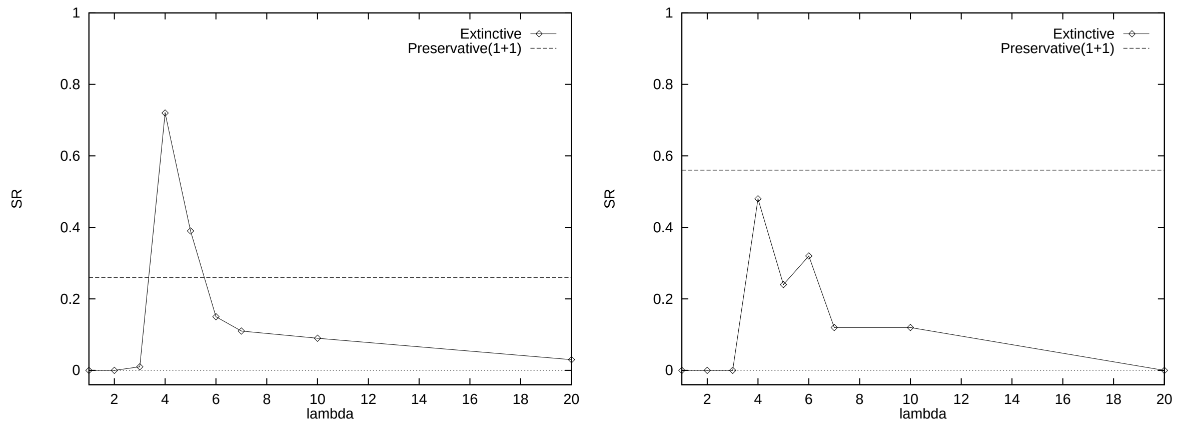


Figure 6.18: Effect of λ in (1,λ)-strategy for $G_{eq,n=200,p=0.035,s=5}$ (left) and $G_{eq,n=1000,p=0.012,s=5}$ (right).

better be called an ES with (1+1)-strategy. If we keep $\mu = 1$ which was found to be optimal then it's obvious that $\lambda = 1$, because $\mu = 1$ means that we only use the best individual to produce offspring. Now in the same spirit, if $\lambda > 1$ and one of the first offspring is better, then we should continue with this offspring to produce new offspring again otherwise we would not use the best individual for producing offspring anymore. To continue with the best offspring would be equal to the (1+1)-strategy. In a chapter 7, this is also verified empirically.

In ESs, Schwefel introduced the (μ,λ)-strategy as a model for the *limited life span* that is found in biology. Here a population of μ individuals produces $\lambda > \mu$ offspring, but now *extinctive* selection is used to select the μ best individuals only amongst the λ offspring. This extinctive scheme is the one that is mostly used in ESs nowadays and it would therefore be worthwhile to test the (1,λ)-strategy on the Graph 3-Coloring problem. Schwefel reports $\lambda/\mu \approx 7$ should be used.

Figure 6.18 shows the effect of the (1,λ)-strategy for several values of λ, comparing it to the (1+1)-strategy. Firstly it shows how important the value of λ is. If $\lambda < 4$ then there's not enough offspring to ensure that the parent will be replaced by an equally good or better individual. And if $\lambda > 10$ then too much resources are wasted to create unuseful offspring and not enough evaluations remain to find solutions. This high sensibility for the value of λ is a disadvantage of extinctive selection. Secondly, figure 6.18 shows that extinctive selection is better for $n = 200$ but worse for $n = 1000$, so it's not clear which selection method would be better and looking at the λ-dependency of the extinctive selection, preservative selection seems more robust and will be used. Also some tests with other values for μ gave poor results, similarly to what was seen with the (μ+1)-strategy.

So now we can compare the asexual EA with (1+1)-strategy to the combination of OX2 and SWAP with $\mu = 600$ and $\mu = 700$ for resp. $n = 200$ and $n = 1000$, the results for which are shown in figures 6.19 and 6.20. We can see that for $n = 1000$, the asexual

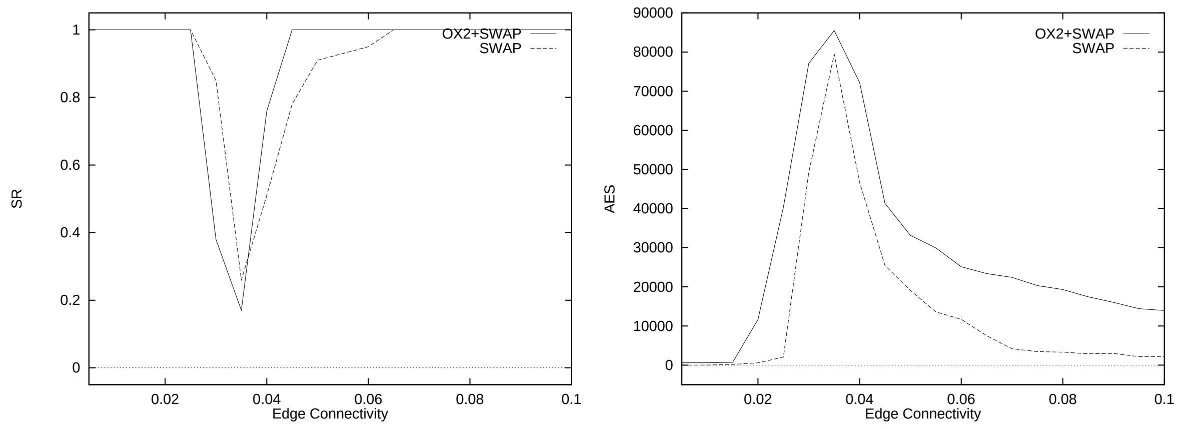


Figure 6.19: Effect of using only mutation for $n = 200$.

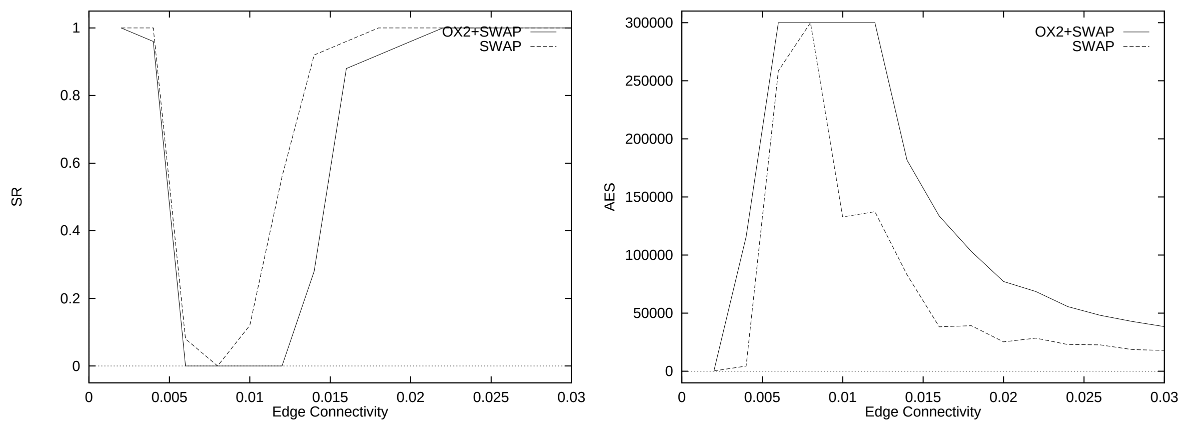


Figure 6.20: Effect of using only mutation for $n = 1000$.

EA is clearly better than the EA with recombination. Though for $n = 200$ a region exists where the EA with recombination is better than the asexual EA. This shows that for some instances (and perhaps in general for some problems), the EA is able to benefit from using a crossover operator *and* a population, because tests for the population size showed that in this region $\mu = 600$ gave better results than $\mu = 2$. Though this is only for this instance and $n = 200$, other instances showed really the asexual EA performs better.

A reason for the better performance of the asexual EA with $\mu = 1$, could be that a population can be a hindrance after converging. After convergence, mutation itself is able to bring in new diversity. Though each time a new hill is found to climb, the "weight" of the population prevents the EA from climbing upwards! Each time that a better individual is found, a lot of evaluations are wasted before converging by fiddling with "old" individuals.

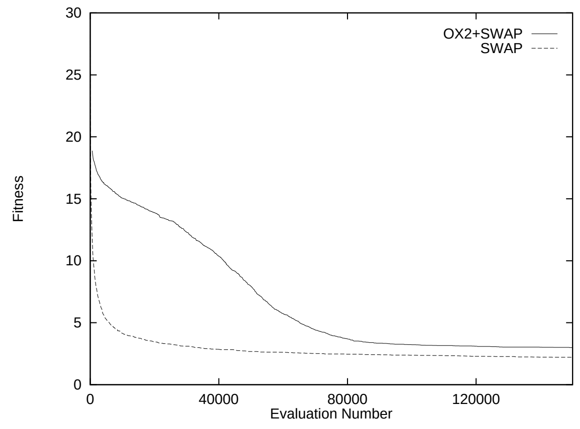


Figure 6.21: Fitness functions of OX2+SWAP and SWAP alone for $G_{n=200,p=0.035,s=0}$.

This is not a problem when μ is very small as is the case for the asexual EA. Figure 6.19 shows that OX2 with a population performs better for easier problems as for $n = 200$ where perhaps the solutions are found before convergence.

Also interesting is the difference between OX2 and SWAP against SWAP alone when we plot the fitness value of the best individual against the evaluation number. As shown in figure 6.21, the weight of the population for OX2, slows the EA down in (hill)climbing upwards[9]. This symbolizes the superior performance of the asexual EA with $\mu = 1$ compared with the EA with recombination and a $\mu \gg 1$. Not only does it have a higher success rate, but also smaller AES, smaller ATE and reduced theoretical and empirical complexity.

6.3 Comparison of the Algorithms

In this section, we compare the performance of the four algorithms: asexual EA with integer representation, asexual order-based EA, DSatur and IG near the phase transition for $n = 200$ and $n = 1000$.

For $n = 200$, DSatur is the best algorithm, followed on some distance by the order-based EA and then IG together with the integer representation EA. DSatur hardly has any problems at the phase transition where the order-based EA has more troubles but does find solutions there! Both the EA with integer representation and IG are not able to find solutions for the hardest instances.

For $n = 1000$, the conclusions become somewhat different. Now the EA with integer representation is better than IG if we look at the success rate. Again the order-based EA is better than the EA with integer representation. But now close to the right side of the phase transition the order-based EA has about the same performance as DSatur!

[9]Really, because we're dealing with a minimization problem, the EA is climbing *downwards*.

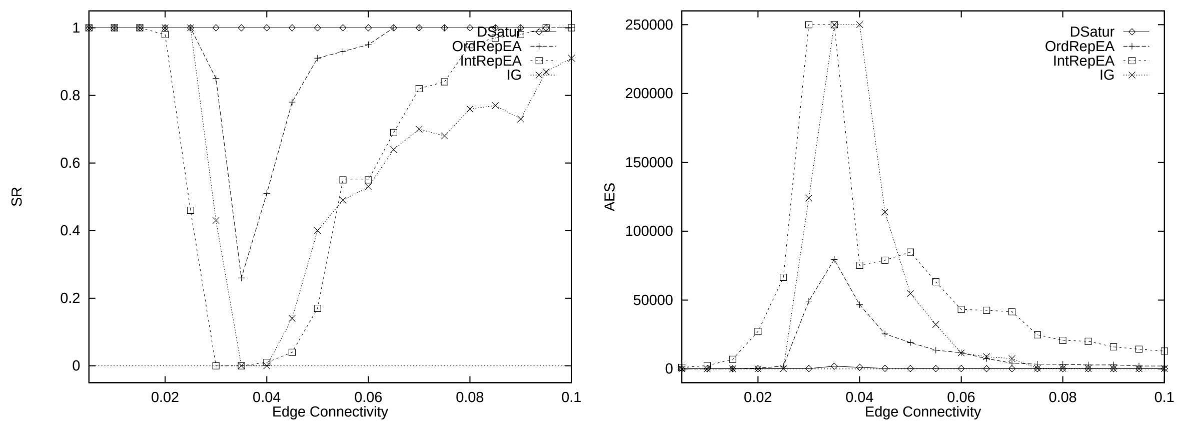


Figure 6.22: Performance of different algorithms for $G_{eq,n=200,p,s=5}$. The EAs both use only mutation and $\mu = 1$.

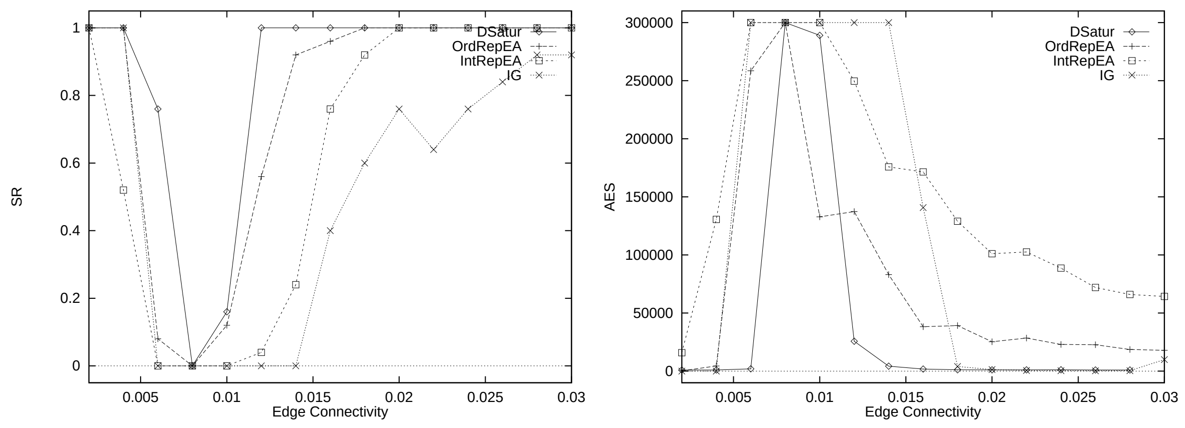


Figure 6.23: Performance of different algorithms for $G_{eq,n=1000,p,s=5}$.

We cannot compare EFE_α for algorithms with different defined search steps and so in figure 6.24 only ET_α is shown. Now if we consider time, a bit further from the phase transition, DSatur is fastest, though for $p \geq 0.1$, IG has similar performance. The order-based EA takes relatively far more time than the other algorithms to find a solution with a probability of 90%. But close to the phase transition CPU-time gets less important as it's more important to *find* solutions there, and so it's interesting to try improving the order-based EA as will be done in the next sections.

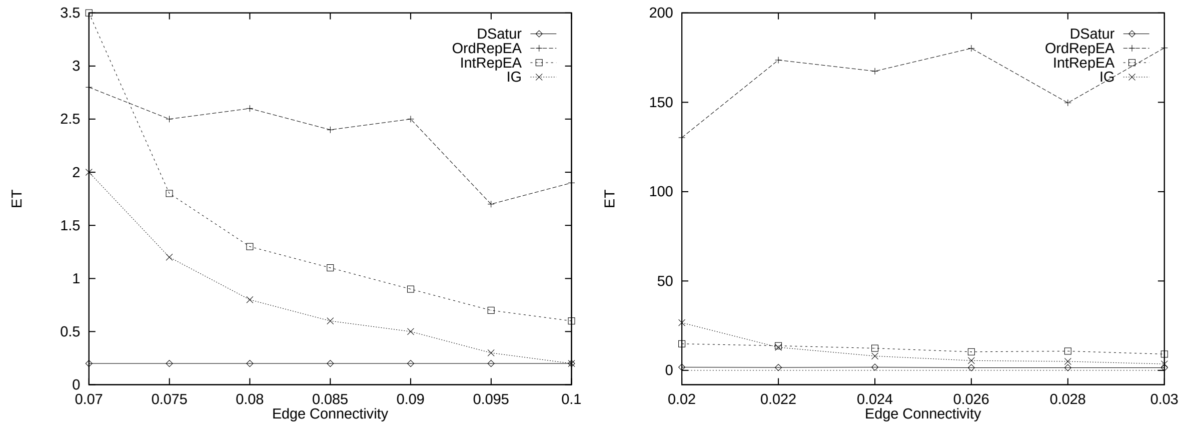


Figure 6.24: ET_α for different algorithms for $n = 200$ (left) and $n = 1000$ (right). Values of p are selected where all algorithms do find a reasonable amount of solutions, so that the expected search time fits in one graph.

6.4 Hybrid Evolutionary Algorithms

Often combining algorithms or parts of them, has a positive effect on the performance. Therefore in the next sections some combinations of the EA with DSatur and IG are examined.

6.4.1 A Combination of IG and EA

The IG heuristics have as a characteristic that they never increase the number of used colors after a reordering which is important when we are looking for the minimum number of colors to color a graph. But although IG never increases the number of colors, it can increase the number of *uncolored nodes* after a reordering and in this respect it is less useful to the EA. On the other hand, where the heuristics can increase the number of uncolored nodes for a permutation, they might just as well decrease it, so that they could be used as an asexual operator in the EA.

There are several ways to combine the IG algorithm with the EA. The first that was tried is to use the IG heuristics instead of the reproduction step to produce new individuals. The number or ratio of evaluations that is used for IG then is a parameter of the method. For an IG step, two (so same number of children is produced each generation) individuals are chosen with tournament selection and for both apart the color classes are computed and one IG heuristic is chosen (with the same probabilities as before) to create a new child for each individual. The results for this combination were not so good as expected. For $n = 200$, some combinations are shown in table 6.11. For all combinations with IG, SR decreased. Also the number of times that an IG heuristic improves the best individual is 0 or 1 per run where a normal recombination (OX2 + SWAP) makes about 20 or more

	SR	AES
EA	0.36	70683
IG + EA	0.16	93065
IG + Reverse	0.32	65387
IG + LargFirst	0.09	83422
IG + Random	0.19	76751

Table 6.11: First combination of IG and EA for $G_{n=200,p=0.030,s=5}$, $T_{max} = 175.000$ and $\mu = 600$. When all three heuristics were used, the ratio in used evaluations, EA:IG, was 1:1, when just one heuristic was used EA:IG was 3:1.

improvements of the best individual, showing this way of combining IG and EA does not work.

Culberson's IG uses only one individual (no hillclimbing) and perhaps the heuristics need to be used several times at the same individual to increase the probability that the IG heuristics will create a better individual. One way to achieve this to let the IG heuristics work on one individual for some number of steps once in a while. Another way is to use the heuristics only for the current best individual and the same scheme as before. The last method was tried and gave the results that are shown in table 6.12. This shows that far

	$p = 0.08$		$p = 0.050$		$p = 0.040$	
	SR	AES	SR	AES	SR	AES
EA	1.00	19319	1.00	33167	0.74	68707
IG + EA	1.00	3666	0.57	40519	0.05	95147
IG	0.64	373	0.38	5471	0.00	175000

Table 6.12: Second combination of IG and EA for $G_{eq,n=200,p,s=5}$, $T_{max} = 175.000$ and $\mu = 600$.

from the phase transition, combining IG with EA has a positive effect on the success rate for IG and on AES for EA, so both profit from the combination. Though not very efficient as IG would have even better results if it would just be restarted a number of times, than for the IG/EA combination. So just using IG instead of the combination works better.

Closer to the phase transition, the IG heuristics are not able to improve on the performance of the EA, but on the contrary decrease it thoroughly, showing that this combination of IG and EA again does not really work[10].

A possible explanation for the failure of using IG heuristics in the EA could be that

[10]This shows how important it is to know what kind of problem instance one is working at, because depending on that the conclusions can be completely different.

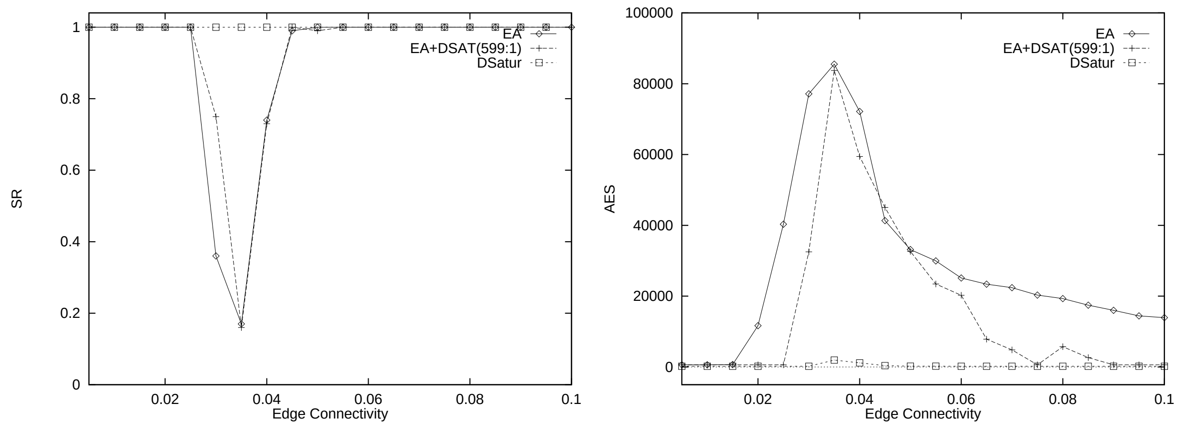


Figure 6.25: Performance for combinations of the EA with recombination and DSatur for $G_{eq,n=200,p,s=5}$, $T_{max} = 150.000$ and $\mu = 600$.

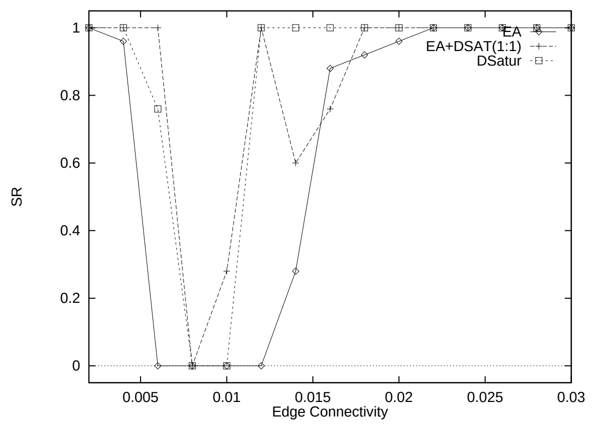
the problems near the phase transitions are the hardest problems with lots of local optima and the IG heuristics might lead the EA to a local optimum where it gets stuck. Still other heuristics might be able also to find solutions near the phase transition as is the case for DSatur. But because the combination of IG and EA is not very fruitful, the IG heuristics will not be included in the EA.

6.4.2 A Combination of DSatur and EA

Because both DSatur and the order-based EA work with permutations, a combination should be possible. A simple combination is to initialize the EA with the permutation that DSatur found without backtracking (leaving uncolored nodes uncolored). Then the amount of the EA population that is initialized with DSatur is a parameter. Experiments were done that initialized the EA with DSatur for one individual in the population, a quarter of the population, half of the population and all of the population[11]. In fact, in the combination we replace the backtracking mechanism of DSatur with the EA. When we have $\mu = 1$ as for SWAP alone, using DSatur to initialize the only individual will perhaps have too much impact on the part of the search space that the EA will examine and indeed this combination did not give consistent results (i.e. sometimes better performance, sometimes worse). Therefore DSatur is only used to initialize the EA that both uses OX2 and SWAP, because then μ is 600 or 700.

In figure 6.25 we see that far from the phase transition it is beneficial to use DSatur, because there DSatur (without backtracking) alone will find solutions and this combination ensures that the solution will never be worse than the solutions of DSatur alone. As can be seen, using DSatur to initialize *one* individual of the population has a positive effect on the

[11]Though in the figures only the graph for the optimal value for this parameter is plotted.



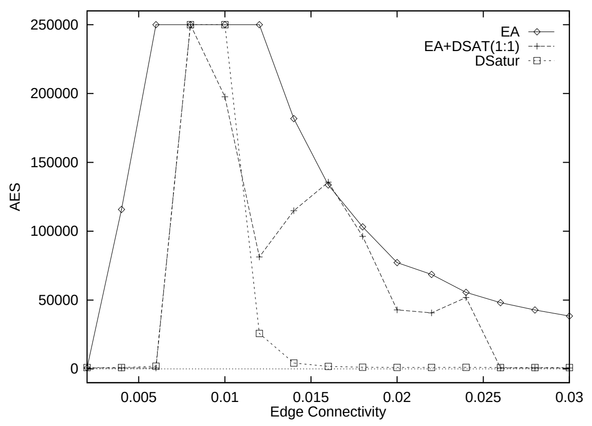


Figure 6.26: Performance for combinations of the EA with recombination and DSatur for $G_{eq,n=1000,p,s=5}$, $T_{max} = 250.000$ and $\mu = 700$.

	p	$s=0$ SR	$s=0$ AES	$s=1$ SR	$s=1$ AES	$s=2$ SR	$s=2$ AES
DSatur	0.010	**0.08**	125081	0.00	300000	0.00	300000
	0.011	1.00	19525	1.00	5305	**0.00**	300000
	0.012	1.00	4725	1.00	8453	1.00	56889
EA+DSAT(3:1)	0.010	**0.28**	226537	0.00	300000	0.00	300000
	0.011	0.60	189587	0.88	119493	**0.92**	144644
	0.012	0.88	158179	1.00	44310	0.84	145492
EA+DSAT(1:1)	0.010	0.00	300000	0.00	300000	0.00	300000
	0.011	0.24	191678	1.00	154543	**0.52**	177337
	0.012	0.88	160619	0.88	126950	0.76	169220

Table 6.13: Performance for combinations of the EA and DSatur for $G_{eq,n=1000,p=0.010,s}$, $T_{max} = 300.000$ and $\mu = 700$.

performance of the EA: SR increases and AES decreases, although at the phase transition using DSatur did not have any advantage. Still for $n = 200$, it's better to use backtracking instead of the EA, because DSatur (with backtracking) has SR=1.00 everywhere. Though things are different for $n = 1000$ where backtracking is not sufficient anymore.

As we see for $n = 1000$ in figure 6.26, further from the phase transition, DSatur (with backtracking) clearly has best performance, but very close to the phase transition for $p = 0.010$, the combination of the EA and DSatur is better than both algorithms alone!

To see whether this result is not just a sample error, some results for other seeds are shown in table 6.13 and they support the same conclusion: for a large problem instance

	$p = 0.035$		$p = 0.050$		$p = 0.080$	
	SR	AES	SR	AES	SR	AES
SWAP	0.26	79441	0.91	19070	1.00	3320
SWAP+degree	0.07	106023	0.56	72301	1.00	14279
OX2+SWAP	0.17	85508	1.00	33167	1.00	19319
OX2+SWAP+degree	0.32	96173	1.00	30545	1.00	17288

Table 6.14: Results for degree weights for $G_{eq,n=200,p,s=5}$ and $\mu = 1$ for the asexual EA and $\mu = 600$ for the EA with recombination.

	$p = 0.010$		$p = 0.014$		$p = 0.020$	
	SR	AES	SR	AES	SR	AES
SWAP	0.12	257040	0.92	83118	1.00	25284
SWAP+degree	0.08	247272	0.60	111132	1.00	59389
OX2+SWAP	0.00	300000	0.28	181730	0.96	77207
OX2+SWAP+degree	0.12	208313	0.80	136781	1.00	67802

Table 6.15: Results for degree weights for $G_{eq,n=1000,p,s=5}$ and $\mu = 1$ for the asexual EA and $\mu = 700$ for the EA with recombination.

close to the phase transition, using the EA is better than using backtracking! Still a problem is to optimally choose the ratio of individuals that is initialized with DSatur, because this does not seem to scale up.

Degree Weights

Up to now all nodes were treated equally and all have a weight of one in the fitness function: if a node cannot be colored, it increases the fitness with a unit of one. If we realize that DSatur gives higher priority to nodes with a higher *degree* and in this way uses the degree of a node as extra information about its importance, we can use the degree of a node for its weight in the EA's fitness function. In fact this is the same as a penalty function [65][62], using some domain knowledge to set the weights. This gives the results for $n = 200$ and $n = 1000$ shown in tables 6.14 and 6.15.

The conclusions are clear: for SWAP the degree weights have a very negative effect both for $n = 200$ and for $n = 1000$. But for the combination of OX2 and SWAP, using degree weights gives very good results, especially for $n = 1000$[12]. With the degree weights the EA with recombination seems to be competitive with the asexual EA (without degree weights) again. This complete difference between a EA with and without recombination

[12]The same observations were found for graph instances with other seeds

	$p = 0.006$		$p = 0.010$		$p = 0.016$	
	SR	AES	SR	AES	SR	AES
DSatur(BackTr)	0.76	1993	0.16	288954	1.00	1784
EA (OX2+SWAP)	0.00	300.000	0.00	300000	0.88	133607
EA/DSAT(OX2+SWAP)	1.00	700	0.44	171012	1.00	809
EA (SWAP)	0.04	262145	0.08	257949	0.96	73678
EA/DSAT (SWAP)	1.00	480	0.08	98299	1.00	31805

Table 6.16: Results for DSatur-based evaluation function for $G_{eq,n=1000,p,s=5}$ and $\mu = 1$ for the asexual EA and $\mu = 700$ for the EA with recombination.

could be a result of the difference in population size. With only one individual the degree weights perhaps constrain this individual too much.

DSatur Decoder

Another way of combining EA and DSatur is found when you realize that in fact DSatur uses the degree as an *ordering* to break ties. Now perhaps this ordering is not the best one (it's just an heuristic) and we could use the EA to try to find better orderings which DSatur can use to break ties. Another way of seeing it is that we use the coloring mechanism of DSatur as a new decoder for the EA. So this DSatur decoder keeps using the (dynamically found) saturation degree to select nodes and colors them with the first available color, but now breaks ties (when two nodes have equal saturation degree) by using a permutation of the EA for the ordering of the nodes. Like before, DSatur leaves the uncolorable nodes uncolored and again the number of uncolored nodes will give the fitness of each individual.

By using this method for $n = 200$, the new EA has SR=1.0 for SWAP and for OX2+SWAP for all edge connectivities, which shows that this coloring mechanism is obviously a better one than the greedy one. So good perhaps that for $n = 200$ the ordering of the nodes is not too important so that it's easy for the saturation heuristic to find a good ordering. But this new decoder takes about four times as much time to compute a fitness value and more importantly, DSatur (with backtracking) alone takes less steps to find solutions. Also it is interesting to note that for the EA with recombination, all solutions were found in the initialization phase and because the initial individuals are random permutations, this indicates again that perhaps this ordering for breaking the ties is not too important and often a random ordering will lead to a solution. On the other side, perhaps $n = 200$ is simple enough for this saturation based color mechanism and $n = 1000$ might be more interesting.

The results for $n = 1000$ in table 6.16 suggest that the combination with the new decoder is far better than both algorithms apart, especially for the EA that uses recombination. This would be true if the new decoder would not be about 20 times slower for problem instances of this size than the previous one. Still the results could be interest-

	$s = 0$		$s = 1$		$s = 2$		$s = 3$	
	SR	AES	SR	AES	SR	AES	SR	AES
DSAT	0.08	125081	0.00	300000	0.00	300000	0.80	155052
EA(SWAP)	0.24	239242	0.00	300000	0.00	300000	0.12	205643
EA/DSAT (OX2+SWAP)	0.44	192434	0.24	198748	0.00	300000	0.64	114232

Table 6.17: DSatur decoder results for other seeds, $G_{eq,n=1000,p=0.010,s}$ and $\mu = 1$ for the asexual EA and $\mu = 700$ for the EA with recombination and DSatur decoder.

ing because the best combination uses a big population and an algorithm for a parallel computer might take advantage of that[13]. Further we see that for bigger graph instances, the order of the nodes *is* important and the degree based ordering is not necessarily the best ordering. This suggests that perhaps the degree weights for the EA (as described in the previous section) might also not be the best setting for the weights (as will be shown in section 6.6). Table 6.17 shows the conclusions are similar for different seeds as well. It also shows the EA/DSAT-combination is a more *robust* method: whenever DSatur finds solutions, the EA/DSAT-combination finds solutions and sometimes the EA/DSAT-combination finds more solutions. Interesting is the difference between the asexual EA and the EA with recombination when the DSatur decoder is used. Probably this is because for the DSatur decoder, a small change in an individual hardly has any effect on the resulting coloring as the individual is only used for breaking ties. In the asexual EA, the SWAP operator possibly is not able to direct the search in a significant way or to move away from local minima.

Because the EA with the new evaluation function is so slow, it will not be used anymore as in the same time it could be better to restart DSatur or EA more often.

6.5 Mutation Rate

6.5.1 Optimal Mutation Rate

Up to now the mutation rate was $\frac{1}{L}$. This is based on some theory by Mühlenbein for the *counting one* problem [59] which showed $p_m = \frac{1}{L}$ (which gives an expectation of one mutated gene for each mutated child), was optimal for that problem. This value also gave good results for a variety of NP-hard combinatorial problems [4][51][52]. Though it should be noted that these results were not for an order-based mutation like SWAP, where in fact two genes are swapped and changed as a minimum. Still the optimal mutation rate could be different from $\frac{1}{L}$ as was used up to now and therefore other values for p_m are tested for $n = 200$ and $n = 1000$.

[13]Because in this research a SSEA is used that produces one new individual per generation, the EA is not easy to parallelize. Though the results might be similar for a generational EA.

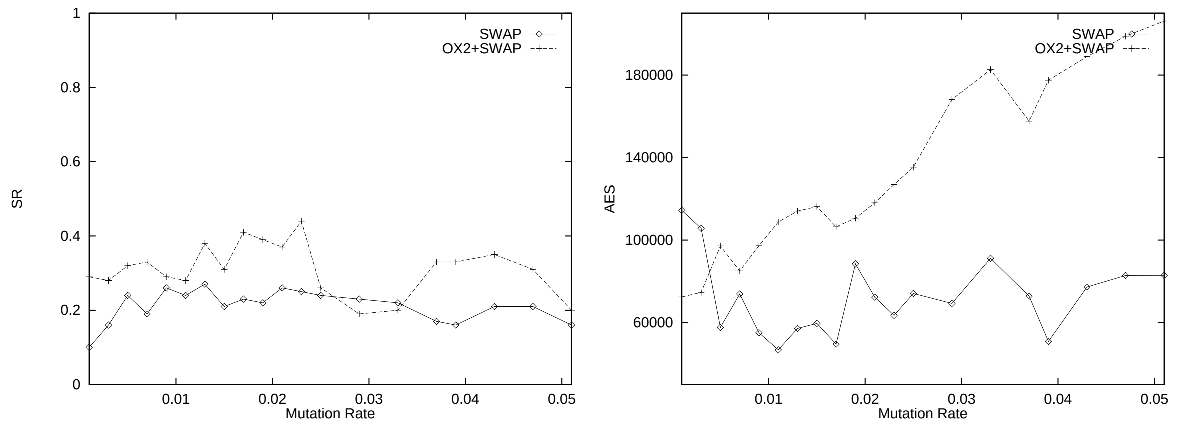


Figure 6.27: Performance for varying p_m for $G_{eq,n=200,p=0.035,s=5}$. $\mu = 1$ for the asexual EA and $\mu = 600$. The OX2+SWAP combination also uses the degree weights (see section 6.4.2).

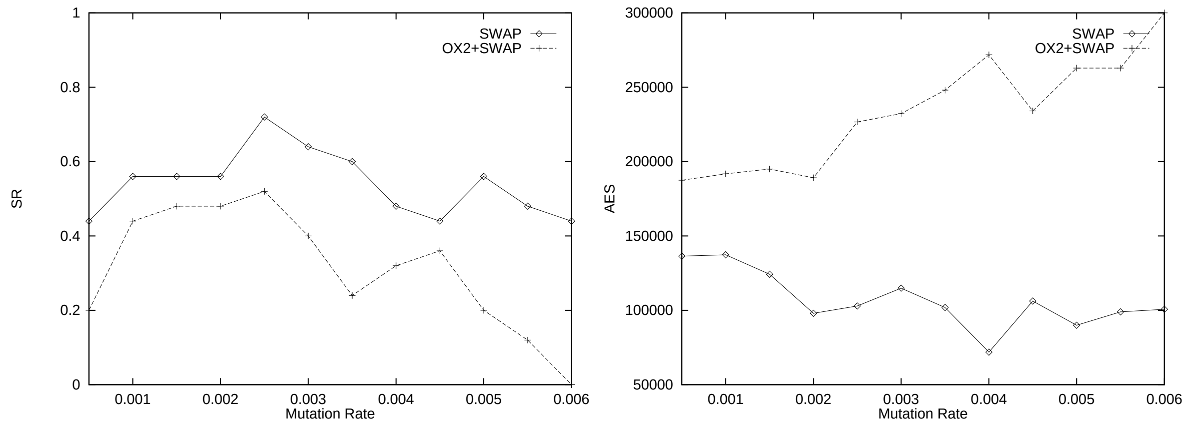


Figure 6.28: Performance for varying p_m for $G_{eq,n=1000,p=0.012,s=5}$. $\mu = 1$ for the asexual EA and $\mu = 700$.

Again an unexpected difference shows up between the results for the EA with and without recombination, as can be seen in figures 6.27 and 6.28. AES clearly increases with p_m for the OX2+SWAP combination, which might be because a lot of evaluations are wasted because the mutation operator is too disruptive. For SWAP alone, increasing p_m hardly seems to have any effect on the AES (though for $p_m < \frac{1}{L}$ the performance drops). The SR seems to be optimal both for SWAP and the OX2+SWAP combination when p_m is chosen around or just above $\frac{1}{L}$, but really it does not change that much when p_m is chosen higher. So for the OX2+SWAP combination $p_m < \frac{1}{L}$ or $p_m > \frac{1}{L}$ gives worse results and $p_m = \frac{1}{L}$ seems to be the appropriate choice here. For SWAP alone, p_m should be higher

	$n = 200$				$n = 1000$			
	$s = 5$		$s = 0$		$s = 5$		$s = 0$	
	SR	AES	SR	AES	SR	AES	SR	AES
SWAP	0.26	79441	0.78	45061	0.92	87425	0.92	82743
OneSWAP	0.26	72645	0.91	44367	0.88	70566	0.92	71367
OX2+SWAP	0.17	85508	0.61	58566	0.28	181730	0.52	146190
OX2+OneSWAP	0.11	94248	0.54	57263	0.24	165231	0.52	167629

Table 6.18: Effect of always swapping exactly one pair of genes for $G_{eq,n=200,p=0.035,s=5}$ and for $G_{eq,n=1000,p=0.014,s=5}$.

than $\frac{1}{L}$, which is logical because a change less than one swap per mutation means no change and a wasted evaluation. Though the exact value for p_m does not matter so much (even $\frac{10}{L}$ is good), which means that for SWAP alone an expected change of one swap per mutation is not really necessary. Because it is not really clear which value for p_m for SWAP alone is best, $\frac{1}{L}$ will be kept.

A value of $p_m = \frac{1}{L}$ implies that an *expectation* of one swapped pair of genes per mutation is "good". But although the expectation is one swap, there still exists some variation in the expectation so that $P(0 \text{ swaps}) \approx 0.36$ for $p_m = \frac{1}{L}$. This can be good when recombination is used before mutating, because then the recombination is not affected by the mutation, but if we use only mutation and $\mu = 1$ (and preservative selection!), then it's completely useless to produce a new individual which is exactly the same as its parent. Also we saw, with higher values for p_m, that more swaps per mutation did not give an improvement, therefore it would be interesting to always do exactly one swap operation per individual (OneSWAP). Results for $n = 200$ and $n = 1000$ are shown in table 6.18, which shows that for the asexual EA it gives a slight advantage to use OneSWAP, but for the EA with recombination it is disadvantageous to always perform a swap, showing it is useful to recombine without mutating. Perhaps a mutation that changes *at least* one pair of genes could be interesting as well. In chapter 7 this has been tried for the Satisfiability problem. To be able to compare the asexual EA and the EA with recombination, the OneSWAP operator will not be used unless stated otherwise.

6.5.2 Changing the Mutation Rate Externally

Up to now, the mutation rate has been kept fixed, but probably changing p_m during a run can have a positive effect on the performance, which has already been suggested by Holland [46]. This can especially be interesting as we saw that the mutation operator turned out to be so important and so mechanisms that can enhance the performance of the mutation operator are important as well. It has been argued that p_m should decrease over time ([45]

and [28])[14]. Hesser and Männer [45] used a function of the form

$$p_m(t) = \frac{b}{e^{-ct}}$$

with b and c constants and t the evaluation number, to control the decrease of p_m. Because $p_m < \frac{1}{L}$ is not considered useful, this is kept as a lower bound for p_m. Similarly $p_m > 0.5$ is not very useful, because then most of the information of the parents is lost, which is against the spirit of an EA and in the beginning the EA starts with random permutations so that a higher p_m is useless. Therefore $b = 0.5$ so that $p_m(0) = 0.5$. The other constant, c, determines for which t, $p_m(t) = \frac{1}{L}$. If $c = \frac{\ln b \cdot L}{t_c}$, with L the number of genes in an individual, then t_c is the evaluation number so that $p_m(t_c) = \frac{1}{L}$, so t_c is a parameter of the method. We get

$$p_m(t) = \begin{cases} \left(2 \cdot e^{-\frac{\ln L/2}{t_c} \cdot t}\right)^{-1}, & \text{if } 0 \leq t \leq t_c \\ L^{-1}, & \text{if } t_c \leq t \leq T_{max} \end{cases} \tag{6.1}$$

Another function to control the decrease of p_m is by Bäck and Schütz [7], which is of the form $p_m(t) = (b + c \cdot t)^{-1}$. They constrain $p_m(t)$ so that $p_m(0) = 0.5$ and $p_m(T_{max}) = \frac{1}{L}$, so they do not use a parameter like t_c. This yields:

$$p_m(t) = \left(2 + \frac{L-2}{T_{max}} \cdot t\right)^{-1}, \text{if } 0 \leq t \leq T_{max} \tag{6.2}$$

Figure 6.29 shows $p_m(t)$ for the two functions when $t_c = T_{max}$. The third curve shows that function 6.1 approximates function 6.2 when $t_c \approx 50.000$. Figures 6.30 to 6.33 show the performance for the two functions for $n = 200$ and $n = 1000$ where t_c is varied for function 6.1.

For SWAP alone, function 6.1 is able to increase the SR (especially for $n = 1000$). As AES is increasing with t_c, we should keep t_c as small as possible. It seems that for $t_c > 50.000$ the SR does not clearly increase anymore and AES starts increasing, so $t_c \approx 50.000$ seems the best value for function 6.1.

Also function 6.2 shows better performance for both problem instances. If we compare the performance of the two functions with $t_c = T_{max}$, function 6.2 is clearly better, but if we compare them with $t_c = 50.000$, they have similar performance. This could be expected because for $t_c = 50.000$ the functions approximate each other as can be seen in figure 6.29. This shows that the EA with SWAP alone is able to benefit from a dynamically changing mutation rate and it indicates that $p_m(t)$ should decrease quite fast in the first part of the run.

[14]Though in their observations p_m expresses the probability *per gene* of mutating it, so that a changing value for p_m changes the expected number of mutated genes. Often for order-based mutations, p_m gives the probability *per individual* of just one mutation in the individual and the effect of dynamically changing p_m will probably be different there. However here, a similar probability distribution for SWAP as for the mutation for integer representation was used and therefore possibly the results are applicable for this mutation operator as well.

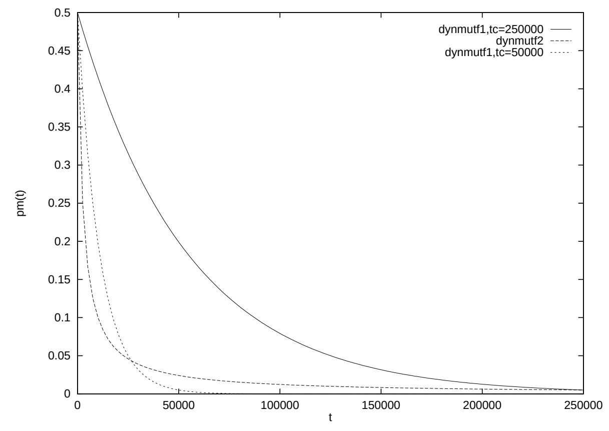


Figure 6.29: Difference between dynamical mutation functions.

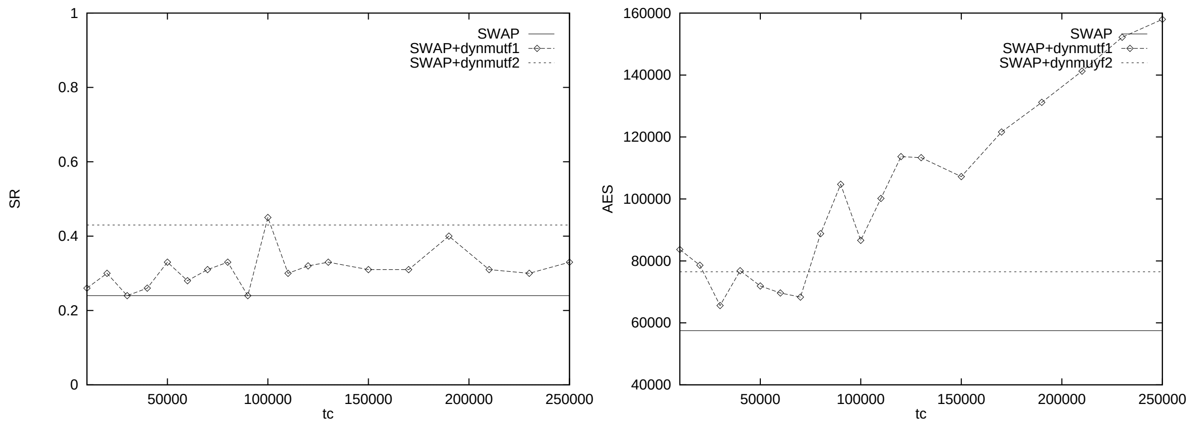


Figure 6.30: Performance of dynamical mutation rates for SWAP and $G_{eq,n=200,p=0.035,s=5}$ and $\mu = 1$.

Figures 6.32 and 6.33 show that the EA with recombination has degraded performance if a dynamically changing mutation rate is used. For $n = 200$, the performance is not significantly better and for $n = 1000$ the performance is far worse for both functions. Again the difference between an EA with and without recombination shows up, though it could also be the difference in population size[15]. A reason for the negative results for the EA with OX2 could be that for a high μ there is no need to bring in new diversity because diversity already exists in a big population and a high mutation rate will only

[15]Another difference with the asexual EA in the setup, is that for the EA with recombination degree weights are used. Still this only changes the fitness function and should not make a difference in the functioning of the dynamically changing mutation rates.

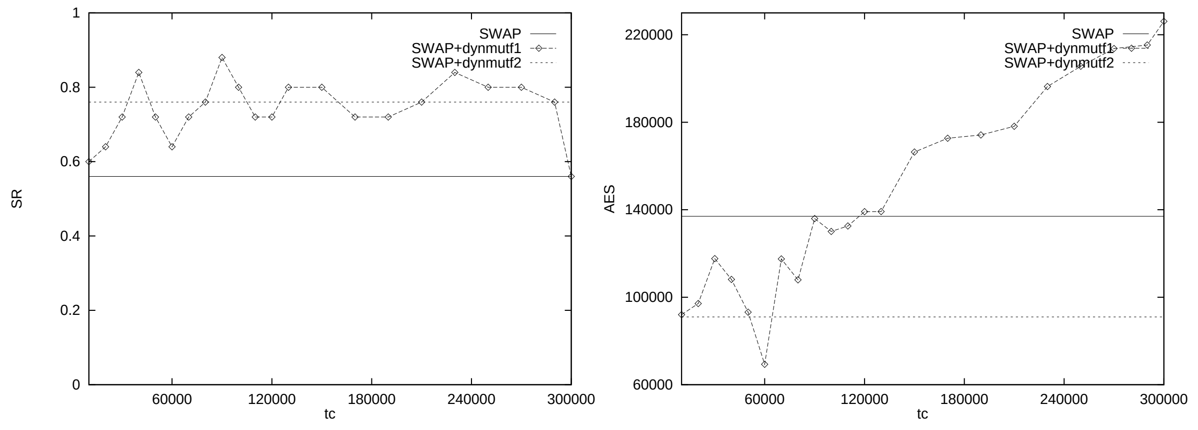


Figure 6.31: Performance of dynamical mutation rates for SWAP and $G_{eq,n=1000,p=0.012,s=5}$ and $\mu = 1$.

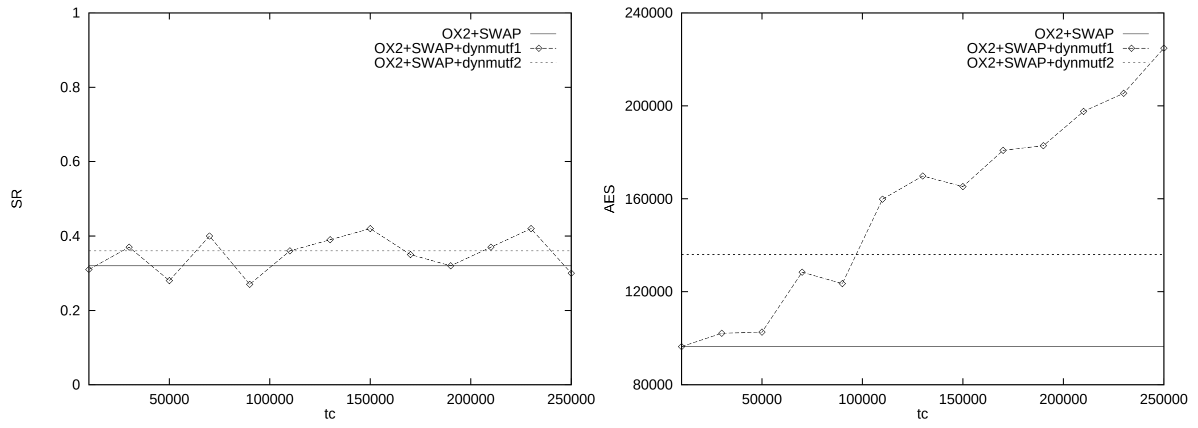
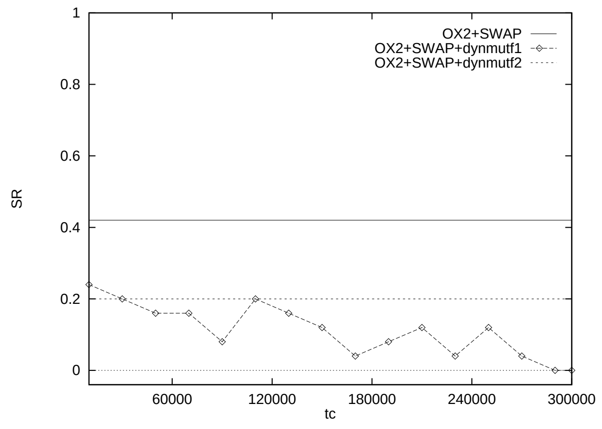


Figure 6.32: Performance of dynamical mutation rates for OX2+SWAP for $G_{eq,n=200,p=0.035,s=5}$ and $\mu = 600$. Also the degree weights are used.

ruin the exploitation of the recombination operator and the exploration could be directed to a small part of the solution space when p_m has settled to $\frac{1}{L}$. For a small μ where not much diversity exists in the population, it is interesting to bring in more disruptiveness or exploration by a high mutation rate in the beginning. Bäck uses some theory in isolation of a recombination operator to observe that dynamic mutation rates can be useful for multimodal fitness functions [5]. He argues that a 1-point crossover could possibly be used together with the dynamic mutation rate technique as the effect of 1-point crossover is expected to be small and so he also notices a difference between the effect of dynamic mutation rates for asexual EA and EA with recombination.



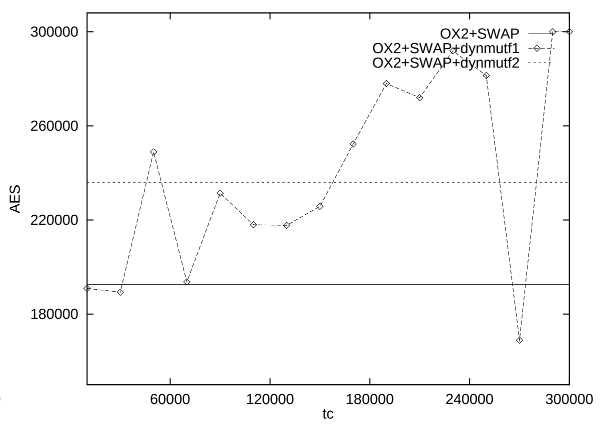


Figure 6.33: Performance of dynamical mutation rates for OX2+SWAP for $G_{eq,n=1000,p=0.012,s=5}$ and $\mu = 700$.

Another interesting technique for decreasing the mutation rate is the one with a self-adapting mutation rate by Bäck [2][3]. Here the mutation rates are incorporated into the genetic representation of the individuals and the mutation rates are also subject to mutation and selection. So no external, deterministic control is needed for changing the mutation rate. Though in personal communication he said the mechanism in [3] did not work so well and he introduced another scheme where the mutation rate is dependent on the fitness value. In [7] he compared the self-adapting mechanism with the deterministic mechanism for changing p_m which is described above and concluded that the deterministic mechanism gave better results when the fitness function is not changing over time. Therefore the self-adapting mechanism is not tested here.

Concluding this section, we can say that a dynamically changing mutation rate increases the performance for the asexual EA and that the mutation rate should decrease quite fast in the beginning of the run as is the case with function 6.2. For the EA with the OX2 operator, the dynamically changing mutation rate does not seem to work.

6.6 Stepwise Adaptation of Weights

Because the degree weights (described in section 6.4.2) had such good performance when applied in an EA that used the recombination operator, it is interesting to examine the idea of changing the weights of the fitness function a bit further. The idea of the degree weights was that the degree of a node was a measure for the *hardness* of a node for the EA to color it. But what is a more direct way for deciding on the hardness of nodes than using the EA itself?

Often for a CSP, the fitness function returns the number of wrongly instantiated variables or the number of violated constraints. For now, without loss of generality, we assume it's the uninstantiated variables that are counted in the fitness function. Then it is natural to see these uninstantiated variables as hard to satisfy and the idea is now just to increase the weights for these variables to make them more important. Now the exact mechanism to increase these weights and the use of these weights can have various forms. For example, it should be decided which part of the population will determine the hardness of the variables. It's possible to use a percentage of the population or just the best individual to determine the weight updates. Eiben *et al.* [23][25] increased the weights of the unsolved constraints of the *best individual* after each run of the EA (for the Zebra problem) and then restarted the EA with these new weights. They used the obtained weights in the next run to get better weights after each run. A problem with this is that you only know that these weights are good if solutions are found with them, but in a practical situation, after finding solutions there's no need to repeat the run with these good weights anymore.

The idea here is to increase the weights *during* a run of the EA with a period T_p by coloring the best individual and increasing the weights in the fitness function with a constant value $\triangle w$ for those nodes that were not colored by the greedy algorithm.

So the fitness function is changed with period T_p and each time the population has to be evaluated again[16]. The EA is supposed to learn *on-line*, step by step, the weights for the fitness function and therefore this mechanism is named *Stepwise Adaptation of Weights* (SAW) mechanism. The results for the EA with SAW were spectacular, as shown in the following experiments, and showed that the SAW-mechanism works for the Graph 3-Coloring problem, a reason to test the parameters T_p and $\triangle w$ thoroughly.

As can be seen in figures 6.34 and 6.35, varying $\triangle w$ (with T_p fixed), has hardly any effect on the performance. Similar graphs were found for other values for T_p and surprisingly results for $\triangle w = 300$ were *identical* to results for $\triangle w = 30$, showing that it has no use to choose higher values for $\triangle w$ than 20 or 30 and figure 6.34 shows that the exact value for $\triangle w$ does not have a significant effect on the performance. For no specific reason $\triangle w = 1$ will be used and so we can omit a parameter.

The effect of the period, T_p, is that for a fixed total number of evaluations, a big period will result in little updates of the weights and therefore less chances for the EA to learn good weights and from this point of view we should choose the period small. A small period, however, means that the EA does not have time to find a good individual for the current set of weights. Also for each weight update all individuals have to be reevaluated because the fitness function changed. This can be costly if the population is big. This means we should not choose T_p too small. Especially for the EA with a big μ, it will probably take some time for the population to converge or to find a good individual for

[16]It could be argued that this reevaluation of the population should be included in the total number of evaluations so that in fact the number of evaluations left for search is reduced. Currently the evaluations for reevaluating the population are not included in the total cost. Though for the EA with $\mu = 1$ this hardly matters if T_p is not too small. Also for a given permutation, after a weight update, its coloring does not change. Only the corresponding fitness value will change and so the decoder, which is most time consuming, is not necessary. All that should be done is recalculate the fitness which can be done very fast if the number of uninstantiated variables would be stored with each individual. Further, these reevaluations do not generate new points in the search space.

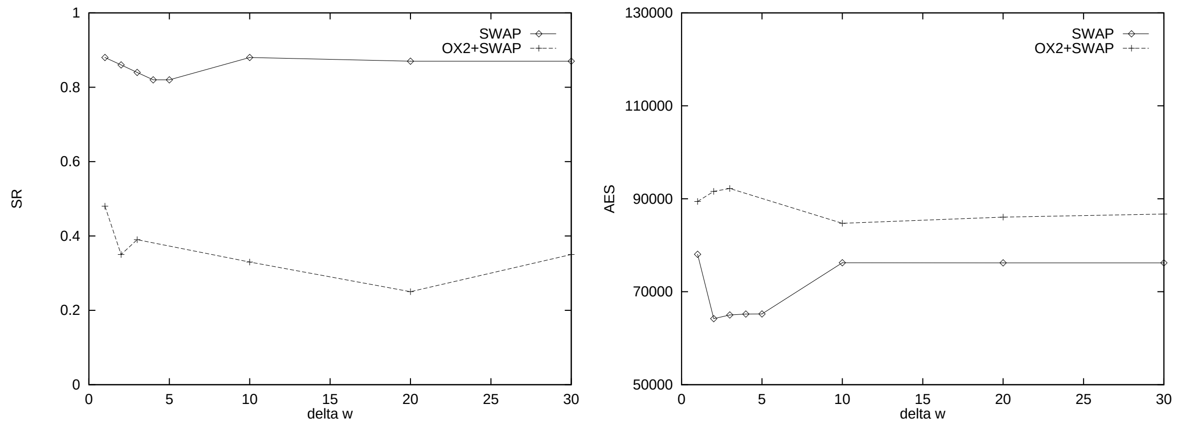


Figure 6.34: Effect of varying $\triangle w$ for $G_{eq,n=200,p=0.035,s=5}$. OX2+SWAP uses $\mu = 600$ and $T_p = 2500$. SWAP uses $\mu = 1$ and $T_p = 250$.

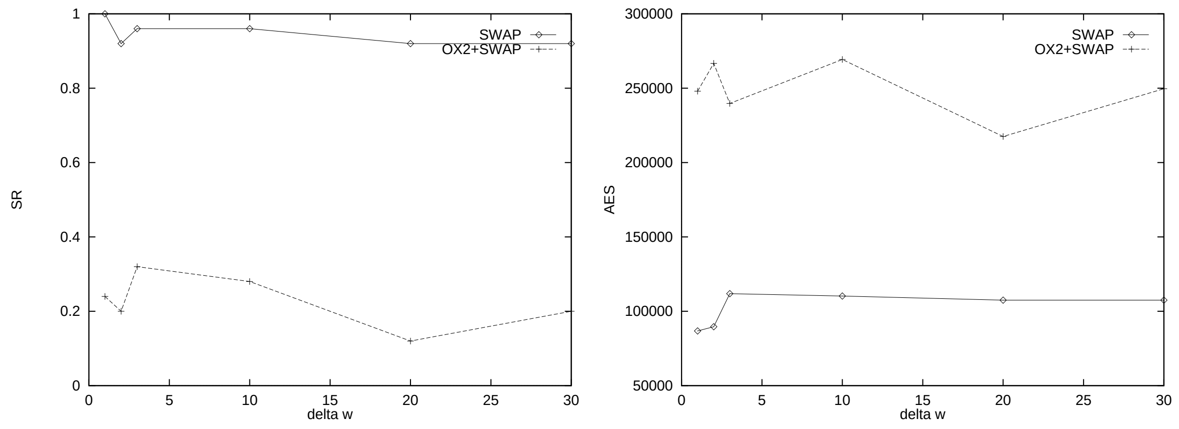


Figure 6.35: Effect of varying $\triangle w$ for $G_{eq,n=1000,p=0.010,s=5}$. OX2+SWAP uses $\mu = 700$ and $T_p = 10000$. SWAP uses $\mu = 1$ and $T_p = 250$.

the new weights and we can expect a higher value for T_p if μ is higher. Figures 6.36 and 6.37 show the performance for various values for T_p for $n = 200$ and $n = 1000$.

Surprisingly for SWAP alone $T_p = 1$ seems to be just as good as higher values for T_p which means the fitness function is adapted after *every* mutation. This can be explained by the fact that for mutation alone often the offspring is worse than the parent (i.e. the best individual if $\mu = 1$) so the same weights will be increased again. And if the offspring is better, the difference will often be small so similar weights will be increased. This is similar to using a higher value for $\triangle w$ and a higher value for T_p and as we saw earlier the value of $\triangle w$ does not really matter. So we can expect that for SWAP alone the value for

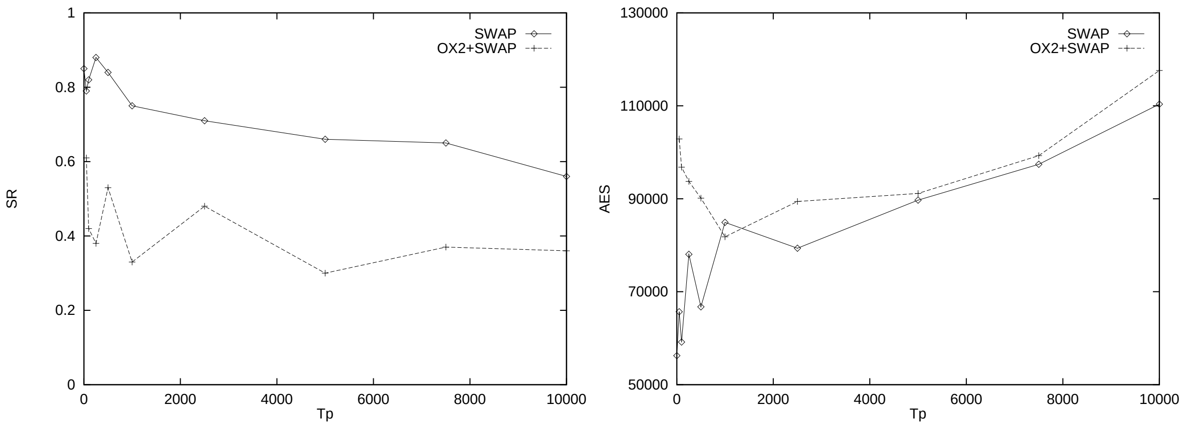


Figure 6.36: Effect of varying T_p for $G_{eq,n=200,p=0.035,s=5}$.

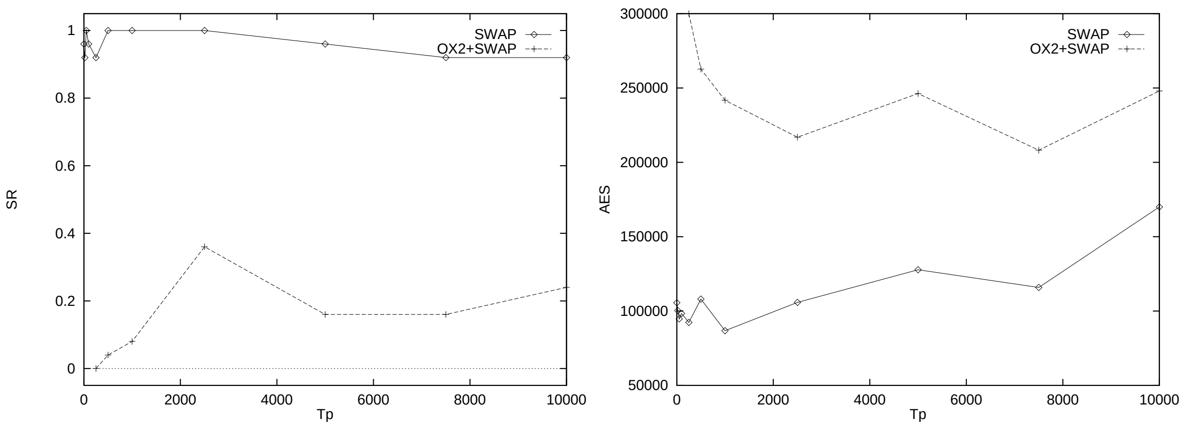


Figure 6.37: Effect of varying T_p for $G_{eq,n=1000,p=0.010,s=5}$.

T_p does not really matter as long as T_p is small (i.e. when enough updates remain). We can see from the graphs that T_p should be between 1 and 2500. For $T_p > 2500$ we see the performance degrades somewhat and this is not strange because then less than 100 weight updates remain. Still the performance stays good for all values of T_p showing the *robustness* of the SAW mechanism. For no specific reason $T_p = 250$ is chosen both for $n = 200$ and $n = 1000$.

After a weight update the whole population will be reevaluated and for a big population, as is the case for the EA with recombination, often a new best individual will be found in the existing population and so for the next weight update the weights will be increased in a different way. This means choosing T_p small for an EA with a big μ *does* have some effect. We can see this for $n = 1000$ and OX2+SWAP where $T_p < 2500$ gives degraded

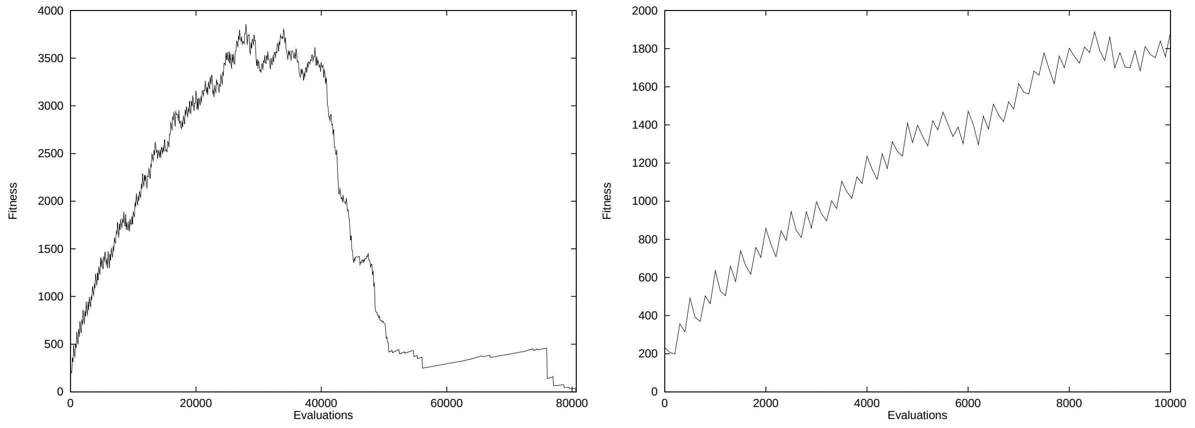


Figure 6.38: Fitness curve for the SAW mechanism for SWAP, $n = 1000$ and $p = 0.010$.

performance. We saw before that for a big population it's more efficient to choose T_p high, as less reevaluations are necessary, and therefore $T_p = 2500$ for $n = 200$ and $T_p = 10000$ for $n = 1000$ will be used.
Very interesting is the fitness curve for a run of the EA with the SAW mechanism when a solution is found. Figure 6.38 shows there is a phase where the mechanism gradually guides the EA through the search space and where the fitness increases a lot because of the increased weights. And there is a second phase where the best individual has been guided through the hardest parts of the search space and now sort of sees the exit of the maze.

Another interpretation could be that with the SAW mechanism, the EA is able to automatically learn a good set of weights or penalties in the first phase that make the problem easier for the EA to solve in the second phase. Then one might expect that these learned weights can be fixed in a next run for the same instance, so that the EA can immediately use them to solve the problem, starting with a new random population. Some tests however, showed that this gave a spectacular decrease in performance which was also found by Frank [32]. This shows that the adaptivity is crucial in two ways in the SAW mechanism. The individual adapts itself to the weights (by mutation and selection) and the weights adapt to the best individual (by SAW) and one cannot be used without the other. It would be interesting to know whether both adaptivities are still needed in the second phase. A test could be done in which the weights are reset to 1 after the first phase to see if the individual still needs to adapt to that specific set of weights. And another test in which the weights are kept but are not adapted anymore after the first phase, could show whether the weights still have to adapt to the individual. But this is left for further research. Also interesting is the zoom-in in figure 6.38 that shows that within each period the fitness (error) drops as the EA finds a better individual and then sharply rises when the weights are updated, giving the image of a *saw*! The long first phase also shows how important it is to continue the EA long enough to enable it to guide the individuals past the hardest local minima.

	$s = 0$		$s = 1$		$s = 2$		$s = 3$	
	SR	AES	SR	AES	SR	AES	SR	AES
DSAT	0.08	125081	0.00	300000	0.00	300000	0.80	155052
EA/DSATdecoder	0.44	192434	0.24	198748	0.00	300000	0.64	114232
EA(SWAP)	0.24	239242	0.00	300000	0.00	300000	0.12	205643
EA(SWAP)+SAW	0.96	76479	0.88	118580	0.92	168060	0.92	89277
EA(OX2+SWAP)	0.00	300000	0.00	300000	0.00	300000	0.00	300000
EA(OX2+SWAP)+SAW	0.44	216834	0.20	256511	0.04	269408	0.20	284292

Table 6.19: SAW results for other seeds, $G_{eq,n=1000,p=0.010,s}$. The asexual EA uses $\mu = 1$ and the EA with recombination uses $\mu = 700$.

In table 6.19 the performance of the SAW mechanism is checked for graph instances with different seeds and compared with DSatur and it shows the spectacular results for the SAW mechanism which enables the EA to beat DSatur on the hardest graph instances.The EA with SAW is also superior to the EA that used the decoder based on DSatur (as described in section 6.4.2). Also important, the table shows the amazingly robust behavior of the SAW mechanism. Even on instances that seem very hard for the other algorithms the SAW EA has very good performance, with not only a high SR, but also a low AES! Further the SAW EA was able to find solutions for $n = 1000$ and $p = 0.008$ which is almost at the phase transition, where DSatur finds no solutions at all. As SAW hardly uses domain knowledge, is robust with respect to its parameters and not very costly, it is a very promising technique for CSPs in general.

6.6.1 SAW and Degree Weights

The results for the EA with degree weights (see section 6.4.2) show that the degree of a node does give a reasonable approximation of the hardness of the nodes and therefore it could be interesting to investigate a combination of the SAW mechanism with the degree weights.

A first method is to use the degree weights as initial weight and then use the SAW mechanism to adapt these weights. This gives the results shown in table 6.20. For SWAP, there's no significant difference when the degree weights are used to initialize the weights. For OX2+SWAP and $n = 1000$, the degree weights seem to degrade the performance although the difference is in fact four solutions and this is maybe not very significant, so the degree weights do not really seem to have any effect, showing again the robustness of SAW.

Another option to include the degree weights is found when realizing that using a separate value for $\triangle w$ for each node is more general than the fixed weight increase that has been used up to now. The idea now is to use the degree weights as $\triangle w$ to make the weights of "important" nodes increase faster. This gives the results shown in table 6.21.

	$n = 200$		$n = 1000$	
	SR	AES	SR	AES
SWAP+SAW+degree	0.71	80675	1.00	105590
SWAP+SAW	0.88	78046	0.92	92379
OX2+SWAP+SAW+degree	0.37	100496	0.08	220332
OX2+SWAP+SAW	0.35	101653	0.24	247978

Table 6.20: Results for first combination of degree weights and SAW for $G_{eq,n=200,p=0.035,s=5}$ and $G_{eq,n=1000,p=0.010,s=5}$.

	$n = 200$		$n = 1000$	
	SR	AES	SR	AES
SWAP+SAW+degree	0.72	75461	0.92	93675
SWAP+SAW	0.88	78046	0.92	92379
OX2+SWAP+SAW+degree	0.29	87939	0.24	228500
OX2+SWAP+SAW	0.35	101653	0.24	247978

Table 6.21: Results for second combination of degree weights and SAW for $G_{eq,n=200,p=0.035,s=5}$ and $G_{eq,n=1000,p=0.010,s=5}$.

Again this combination of SAW and degree weights is not really fruitful what could perhaps be expected as figure 6.34 shows that the value for $\triangle w$ is not really important.

6.6.2 Population Size Again for Asexual EA

With SAW we saw that the population size has influence on the period T_p, therefore it's also possible that the optimal values for μ have changed and so it's necessary to test for the effect of the population size again if SAW is used.

Figure 6.39 shows that for $n = 200$ and SWAP alone the optimal μ is still 1 and a similar result was found for $n = 1000$, so that the conclusion about a minimal population for an asexual EA, does not change under the SAW mechanism.

Figure 6.40 shows the results for extinctive and preservative selection when SAW is used. Firstly, we see that the optimal value for λ shifts from $\lambda = 4$ for $n = 200$ to $\lambda = 7$ for $n = 1000$, so the optimal value of λ does not scale up. This shows again the high sensibility of the (1,λ)-strategy. Secondly, if we compare figure 6.40 to figure 6.18, that gives the performance of extinctive selection without the SAW mechanism, we see that SAW also works for extinctive selection! Thirdly, if λ is chosen optimal then extinctive and preservative selection have similar performance, but preservative selection seems more robust and therefore the (1+1)-strategy will be kept.

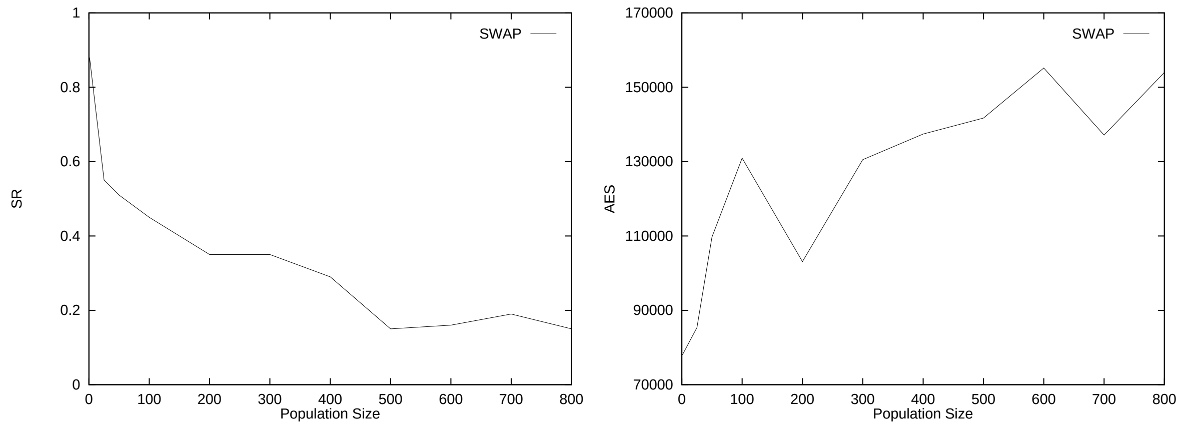


Figure 6.39: Effect of population size on performance for SAW for $G_{eq,n=200,p=0.035,s=5}$.

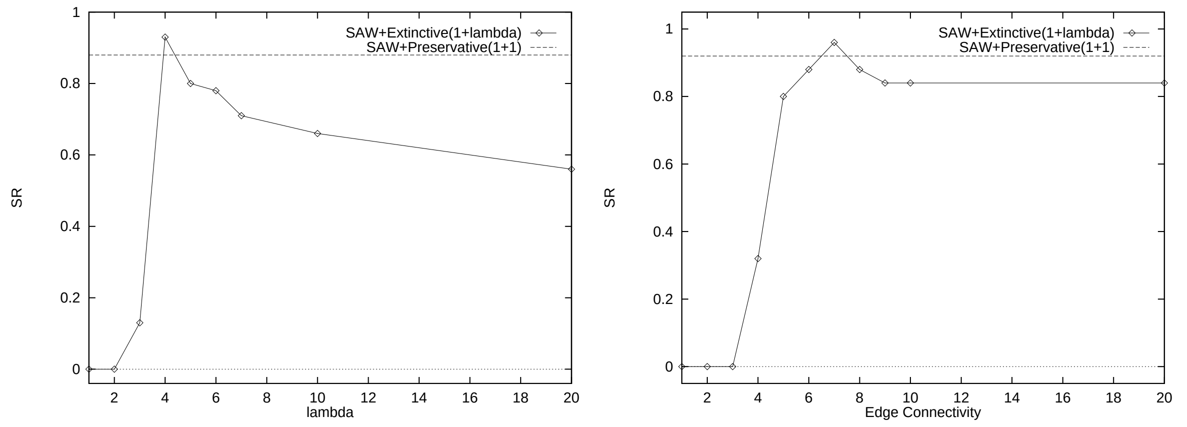


Figure 6.40: (1,λ)-strategy with SAW for $n = 200$ (left) and $n = 1000$ (right).

6.6.3 Population Size Again for EA with Recombination

We saw that the population size and/or recombination had effect on the period of the SAW mechanism, but then the other way around SAW could also have effect on the optimal population size and this is tested again for $n = 200$ and $n = 1000$ in this section.

Figures 6.41 and 6.42 show that for OX2, still a minimum value for μ exists below which it's useless to have $\mu > 1$, like we saw in section 6.2.2. Above this minimum value, $\mu = 600$ still seems the best choice and again below the minimum value, $\mu = 2$ is optimal. But now with SAW the performance is superior with $\mu = 2$!

An explanation could be that the tradeoff between a long period (so a good individual for the new weights can be found) and many weight updates (so enough samples are taken to find good weights) is more of a problem for a big μ, because a long period is necessary

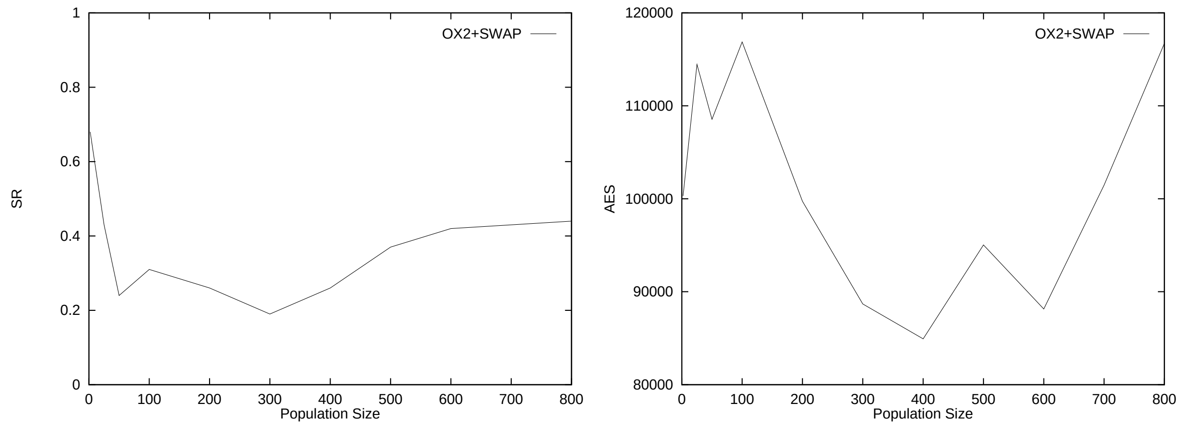


Figure 6.41: Effect of changing the population size for OX2 with SAW and $G_{eq,n=200,p=0.035,s=5}$, $T_p = 2500$.

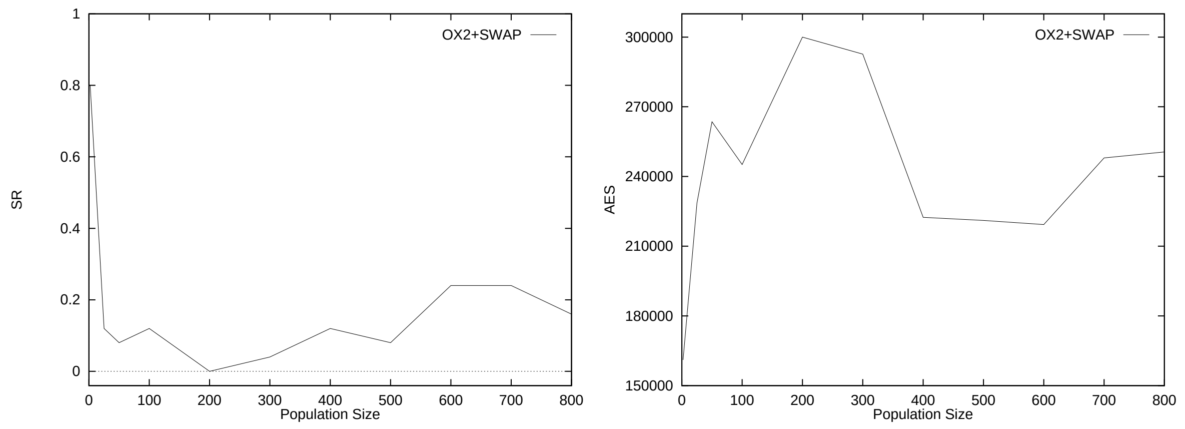


Figure 6.42: Effect of changing the population size for OX2 with SAW and $G_{eq,n=1000,p=0.010,s=5}$, $T_p = 10000$.

to allow the population to converge and therefore less weight updates remain. Possibly the value of T_p is not big enough to allow the EA to find good individuals again, but cannot be chosen higher as not enough weight updates would remain. This is not a problem anymore with $\mu = 2$.

A problem with this explanation is that it predicts that a higher period with the same number of weight updates should give better results. This should be because figure 6.42 shows that for the same T_p, $\mu = 2$ gives better results indicating that the number of updates is enough and therefore according to the tradeoff, T_p would be too low for $\mu = 600$. But runs (both for $n = 200$ and $n = 1000$) with T_p doubled and the total number of evaluations doubled (so the number of weight updates stays equal) did not give better results where runs with only the total number of evaluations doubled did give better results so the original T_p was high enough. Still the tradeoff remains but it does not hold anymore that with a big μ the same results can be obtained as for a small μ by increasing T_p.

A better explanation can be found by focusing on the real difference between a big and a small population. When a big population has converged, no (genotypic or phenotypic) diversity is left and when the fitness function changes because of a weight update *after* the population has converged, no diversity is left to start an exploration for this new fitness function. As discussed before, after converging, a population is a hindrance and many evaluations are wasted for converging again each time that a better individual is found or when the fitness function is changed. This is not a problem when μ is very small! A solution would be to stop using a population as soon as it is converged and then only use mutation (or start using mutation only then) or to reinitialize the population after convergence or after a weight update as was done by Eshelman in [26][17]

Still for $\mu = 2$ such a long period as before is probably not necessary and as we decrease T_p, more weight updates can be done. For $n = 200$ the optimal period now is found for $2 \leq T_p \leq 250$ and for $n = 1000$ it is found for $2 \leq T_p \leq 1000$ and again for no specific reason $T_p = 250$ is chosen both for $n = 200$ and $n = 1000$.

With this new settings, the EA with recombination can compete with the asexual EA again, but if we really want to compare and see what the effect is of using recombination, we have to do tests for a (2+2)-strategy both for OX2+SWAP and for SWAP alone. When this was done for $G_{eq,n=1000,p=0.010,s=5}$ and $T_p = 2$, OX2+SWAP gave SR=0.96 and AES=115508 and SWAP alone gave SR=0.88 and AES=101141, which does not show a significant difference, so we could reason that crossover is useless here, even more because it constrains us to use $\mu = 2$ where $\mu = 1$ is far better for SWAP alone as can be seen in figure 6.39.

6.6.4 SAW and Integer Representation

To see whether the SAW mechanism also works for another representation, it was tested for the EA with integer representation.

As shown in table 6.22, SAW has very good results again both for the asexual EA and for the EA that also uses uniform crossover. For the EA with uniform crossover, the results show that when SAW is used it does not matter very much if IP is used or not, showing again the robustness of the SAW mechanism. Although the order-based EA is still far better than the EA with integer representation, this result shows that SAW is generally applicable.

[17]This was implemented as well, but as the first results were not very encouraging, it was not investigated any further.

	$n = 200$		$n = 1000$	
	SR	AES	SR	AES
Mutation	0.17	84803	0.76	171491
Mutation+SAW	1.00	56844	1.00	116411
Uniform+Mutation+IP	0.02	131861	0.12	228938
Uniform+Mutation+SAW+IP	0.86	106633	0.72	254022
Uniform+Mutation+SAW	0.90	122485	0.56	240921

Table 6.22: Results of the SAW mechanism for the integer representation for $G_{eq,n=200,p=0.050,s=5}$ and $G_{eq,n=1000,p=0.016,s=5}$. Mutation alone uses $\mu = 1$ and the combination uniform+mutation uses $\mu = 250$.

6.7 The Final Tests

In this section a final comparison is done between the asexual EA with SAW and DSatur on arbitrary 3-colorable, equi-partite 3-colorable and flat 3-colorable graphs (as described in section 5.2) for $n = 200$, $n = 500$ and $n = 1000$ for edge connectivities around the phase transition where the hardest graphs are found. Resp. 100, 50 and 25 runs are done for $n = 200$, $n = 500$ and $n = 1000$. For all instances, $T_{max} = 300.000$, $T_p = 250$, $s = 5$ and (1+1)-strategy is used. Also because no recombination is used, the version of SWAP that always swaps exactly one pair of genes, OneSWAP, is used as the mutation operator (and so parameter p_m is not used anymore). The conclusions are obvious. For $n = 200$, DSatur is best for all p (although for flat graphs DSatur has some troubles near the phase transition and the EA is better as can be seen in figure 6.49). But for $n = 500$ and $n = 1000$, close to the phase transition, the EA beats DSatur consistently: often the EA is able to find solutions where DSatur does not find any! This could be because the instances with $n = 200$ are small enough to permit backtracking to get out of local optima and find solutions which is not possible anymore for $n = 500$ and $n = 1000$ where the solution space becomes too big.

For all graphs and sizes, further from the phase transition, DSatur beats the EA in time. For example, for an instance where both algorithms have SR=1.00, like $G_{eq,n=1000,p=0.016,s=5}$, DSatur takes about 1.5 second for 1000 node expansions and so the expected time to find a solution with 90% certainty is about 1.5 seconds. The EA uses about 3.6 seconds per 1000 evaluations and has an expected time of about 70 seconds to find a solution with 90% certainty. For these instances, the DSatur heuristics are good enough to find a solution in one shot where every node only has to be considered once. This can only be beaten by smarter implementations of the DSatur algorithm, because the number of search steps is already minimal and the EA has no chance of beating DSatur. Still the EA is able to find solutions far from the phase transition showing again the robustness of the EA, although far from the phase transition the number of edges in the graph grows fastly and the computation time of the evaluation function becomes more important now so that the

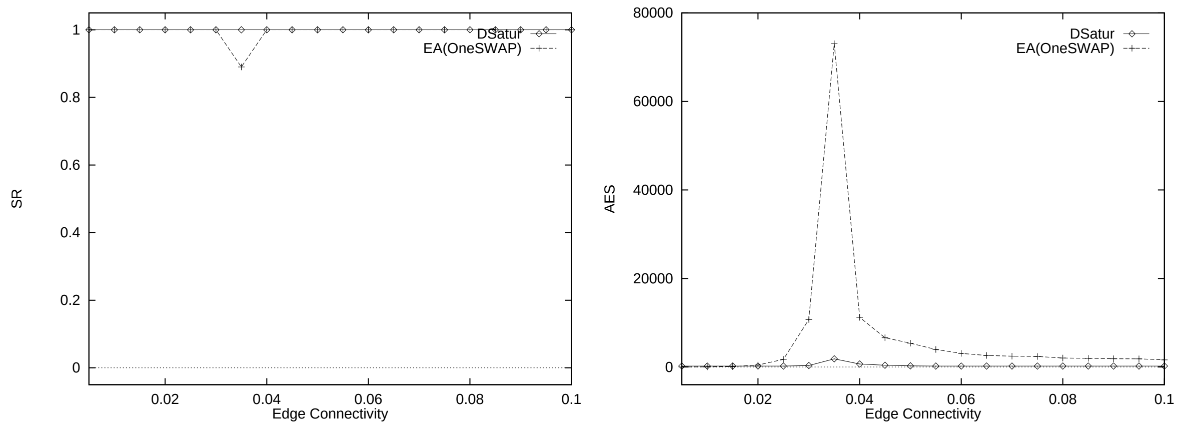


Figure 6.43: Comparison around phase transition for 3-colorable graphs and $n = 200$.

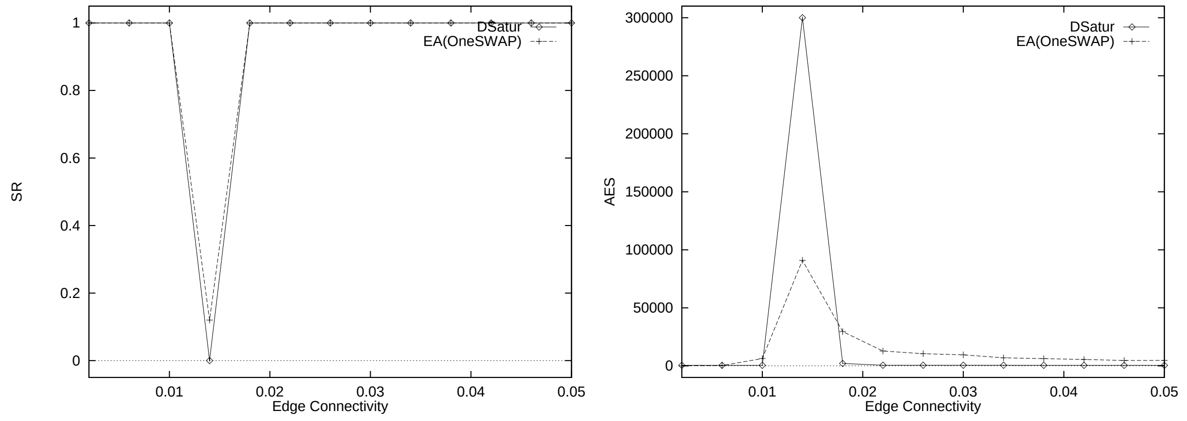


Figure 6.44: Comparison around phase transition for 3-colorable graphs and $n = 500$.

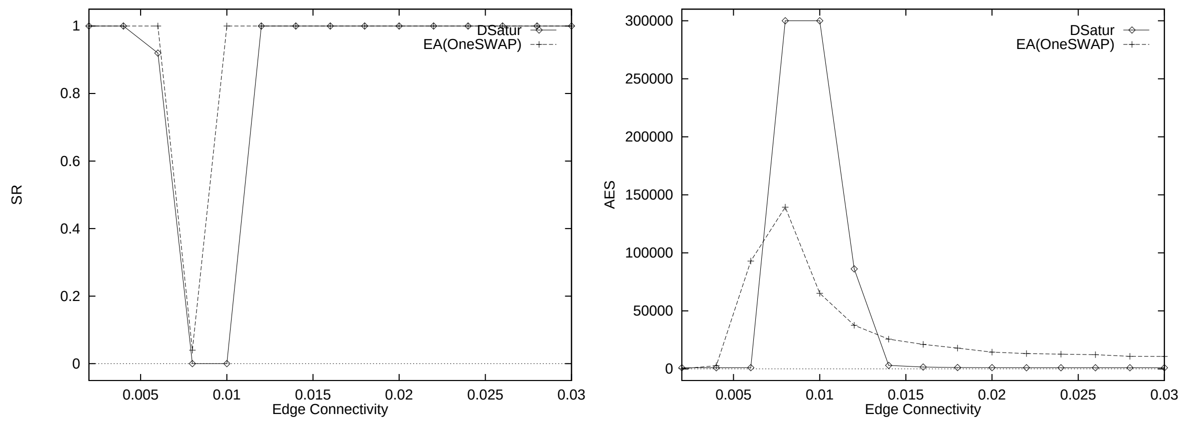


Figure 6.45: Comparison around phase transition for 3-colorable graphs and $n = 1000$.

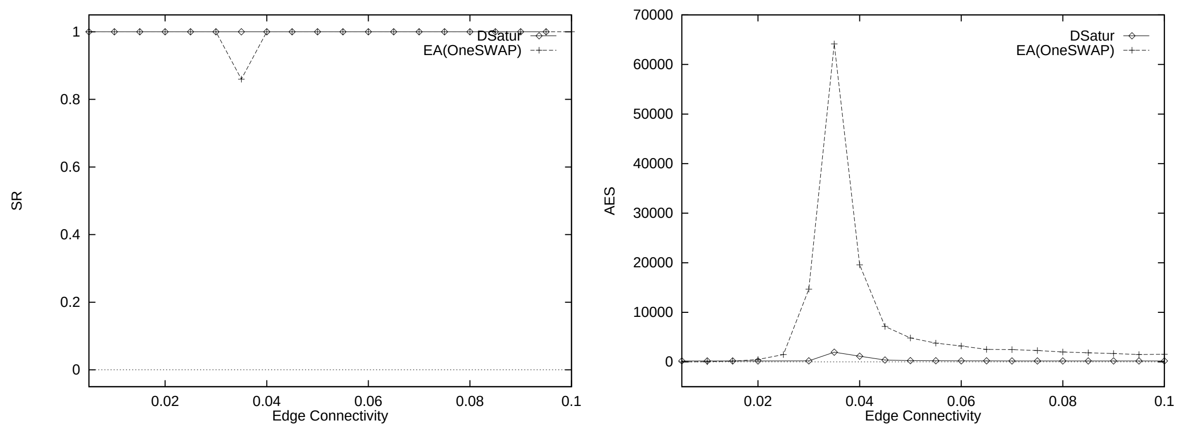


Figure 6.46: Comparison around phase transition for equi-partite, 3-colorable graphs and $n = 200$.

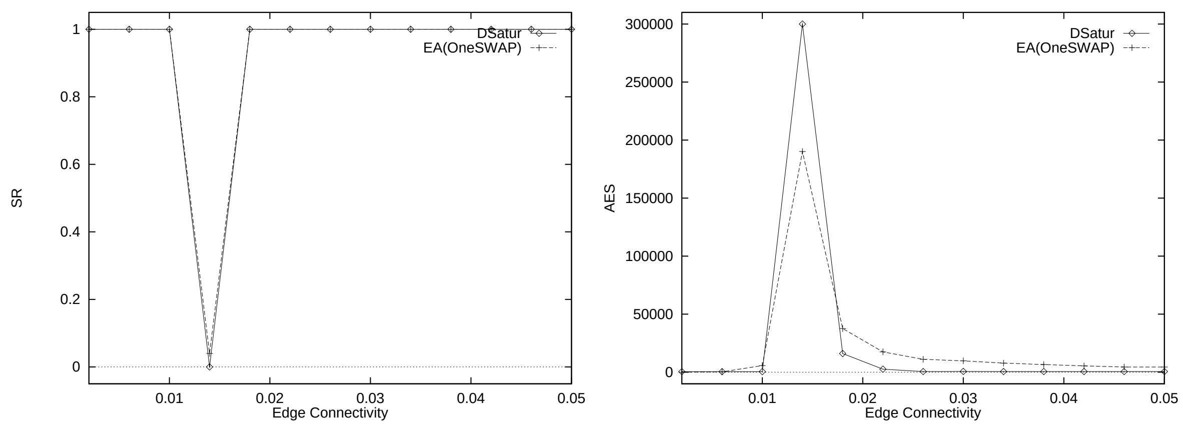


Figure 6.47: Comparison around phase transition for equi-partite, 3-colorable graphs and $n = 500$.

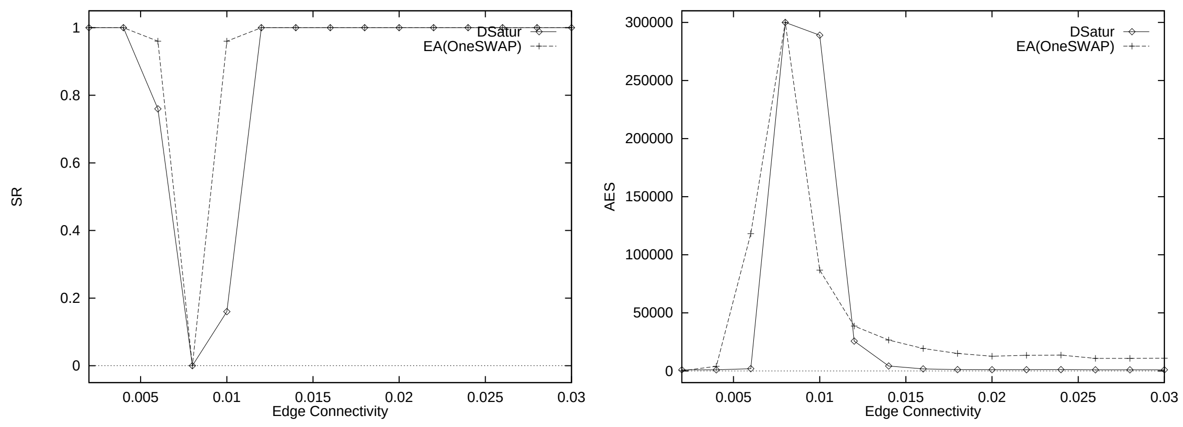


Figure 6.48: Comparison around phase transition for equi-partite, 3-colorable graphs and $n = 1000$.

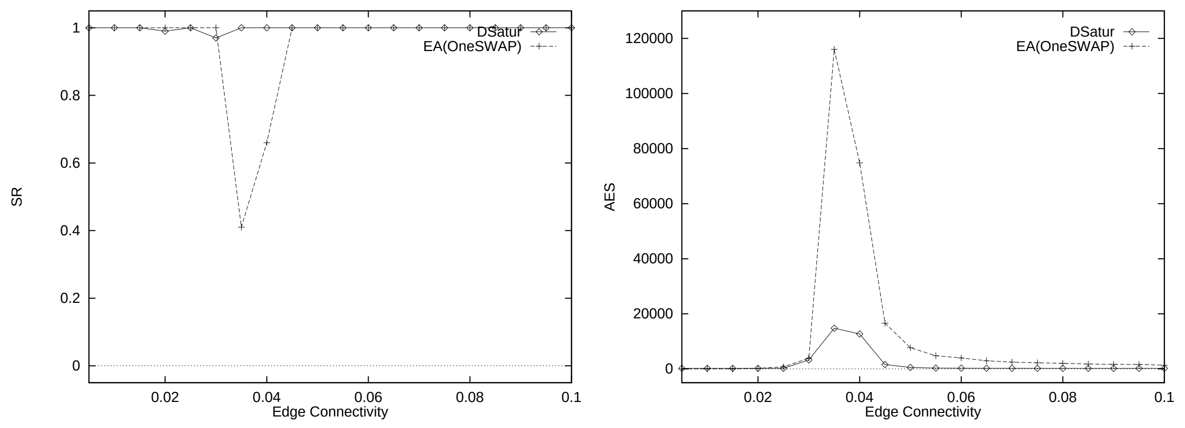


Figure 6.49: Comparison around phase transition for flat, 3-colorable graphs and $n = 200$.

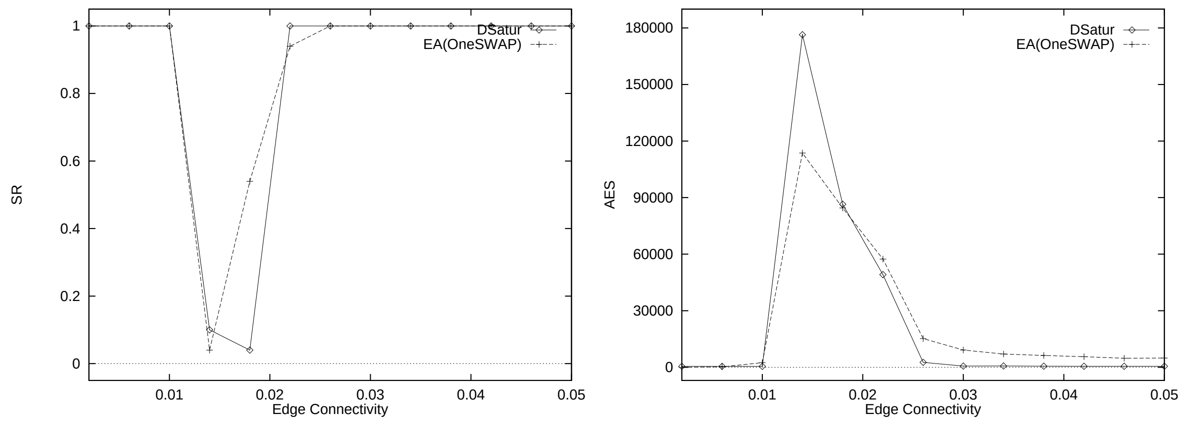


Figure 6.50: Comparison around phase transition for flat, 3-colorable graphs and $n = 500$.

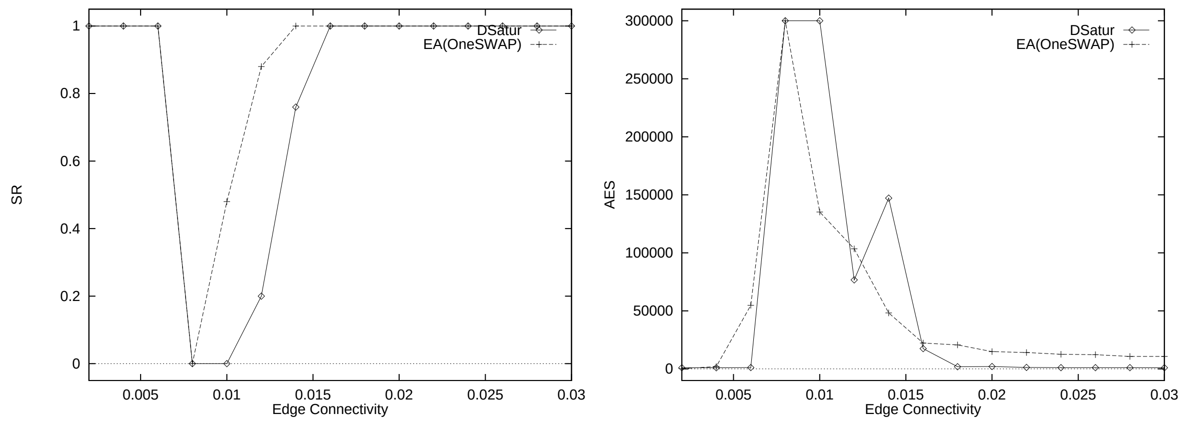


Figure 6.51: Comparison around phase transition for flat, 3-colorable graphs and $n = 1000$.

integer representation that has a very fast evaluation function could perform better far from the phase transition.

One could doubt the usefulness of the EA because it's only better than DSatur at such a small range of instances, but we have to realize that these instances are the really hard instances and that EAs were designed to tackle hard problems with many local optima and this small range is exactly where the instances with most local optima are and it's for these instances that the EA beats DSatur. Also EAs were designed to cope with problems for which no domain knowledge exists so that no special purpose algorithm can be made. Earlier in section 6.3, we saw that the EA with integer representation, for which hardly any domain knowledge was used, still performs quite well at a broad range of instances.

Another useful property of the EA is that its performance can be boosted by giving it extra time, which does not hold for DSatur. If we increase the total number of evaluations for DSatur for $G_{eq,n=1000,p=0.008,s=5}$ to 1.000.000 we still get SR=0.00, but if we do the same for the EA with SAW and SWAP we get SR = 0.44 and AES = 407283 (SES = 175889), showing the EA is able to benefit from more evaluations, where for backtracking these extra evaluations are not enough to get out of the local optima.

6.7.1 Scalability

For any algorithm, it is important how the performance scales up with the problem size n. For complete algorithms, only the time complexity is important, because either a solution is found or a solution does not exist. As we are dealing with NP-complete problems, the interesting question is whether the EA works in polynomial time or exponential time. A complete polynomial algorithm probably does not exist, unless P = NP. Still, as the EA is an incomplete algorithm, it could have polynomial time complexity. The time complexity normally is defined as the number of search steps necessary to find a solution or conclude that no solution exists, expressed in the problem size n. So for our probabilistic algorithms this could be EFE_α and we're interested how EFE_α scales up with n. Also interesting is how ET_α and AES scale up with n. As we use a maximum of T_{max} search steps, also DSatur will not always find the solution if one exists.

To be able to say anything about the scalability we have to test various values for n, but we should use instances with similar hardness and this is done by using $p = c/n$, for a constant c. As DSatur, only found solutions for $n = 1000$ for $p \leq 0.006$ or $p \geq 0.010$, we use $p = 10/n$. Also to get an idea about the scalability closer to the phase transition, $p = 8/n$ is used.

In figure 6.52 we see how the ability of finding existing solutions scales up with n for both DSatur and the best EA version. Clearly this ability scales up far better for the EA than for DSatur. When the instances grow bigger, DSatur does not find any solutions anymore, where the EA still finds solutions even for very big instances at the phase transition. For $p = 10/n$, the time complexity of DSatur and EA is compared.

Figure 6.53, clearly shows that for $p = 10/n$, the EA scales up almost linearly both for AES and EFE! DSatur is faster than the EA up to $n = 750$, but after that the EA's performance is far better as DSatur scales up superlinearly and possibly exponentially.

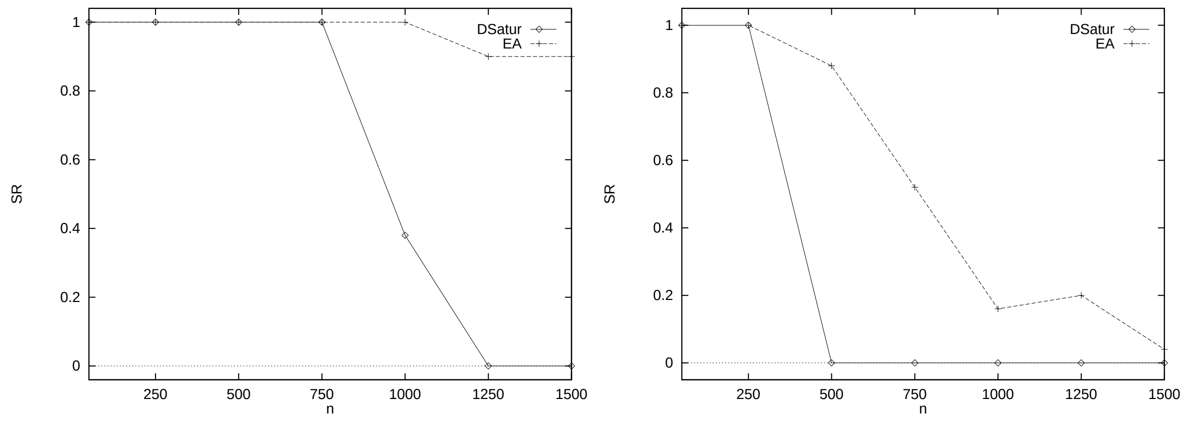


Figure 6.52: SR for different problem sizes n, $p = 10/n$ (left) and $p = 8/n$ (right). For $p = 10/n$, $T_{max} = 500.000$ and results are averaged over 50 runs. For $p = 8/n$, $T_{max} = 1.000.000$ and results are averaged over 25 runs.

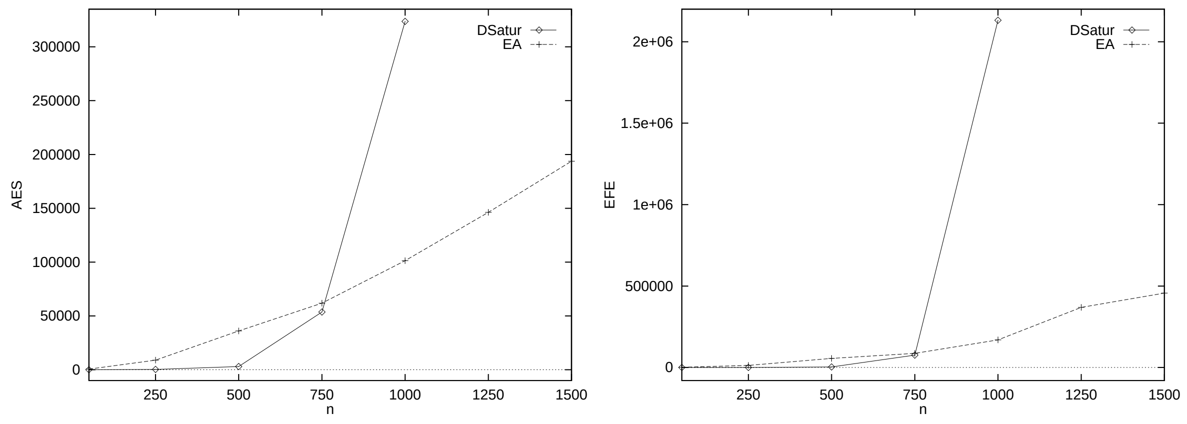


Figure 6.53: AES and EFE_α for different problem sizes n. For all instances $T_{max} = 500.000$, $p = 10/n$ and results are averaged over 50 runs.

The time complexity of the greedy algorithm (and therefore the time complexity of a search step in the EA) is $O(n \cdot n \cdot p)$, but as we fixed p to $p = c/n$, we get $O(n)$, which is confirmed in figure 6.54 with ATE for the EA and DSatur. As ATE scales up linearly, ET_α, which is $\text{EFE}_\alpha \cdot \text{ATE}$, has the same complexity as EFE_α for both algorithms.

Still we also want to know how this behavior scales up when p is chosen closer to the phase transition and therefore for the EA alone, the same experiments have been done for $p = 8/n$, as shown in figures 6.55 and 6.56. AES still seems to scale up linearly for $p = 8/n$, which makes it easier to make a good guess for T_{max}. Though for the hardest graphs, more

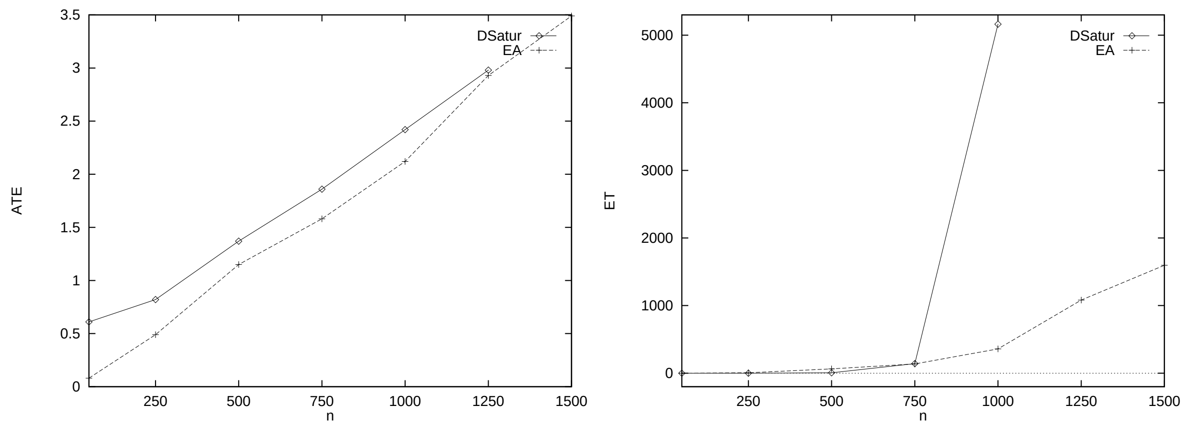


Figure 6.54: ATE and ET_α for different problem sizes n for $p = 10/n$. For all instances $T_{max} = 500.000$, $p = 10/n$ and results are averaged over 50 runs.

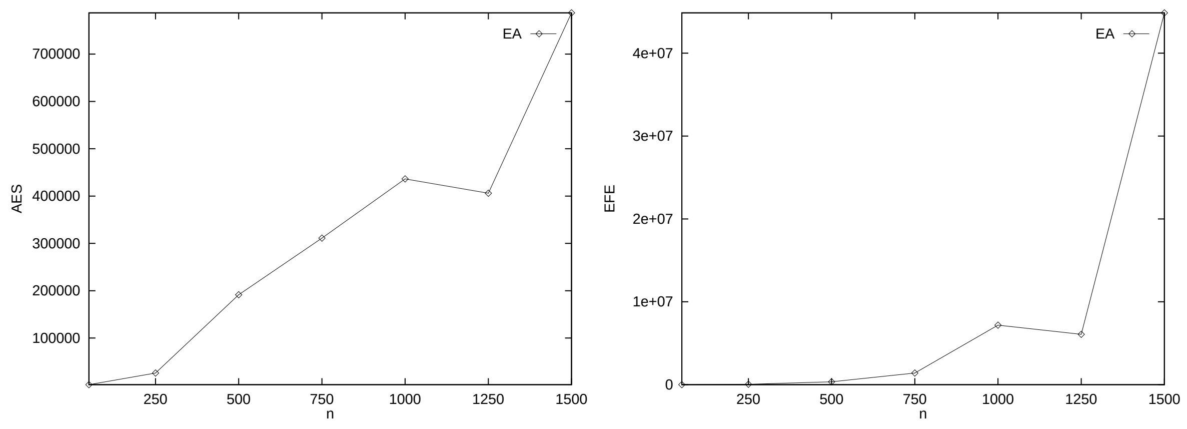


Figure 6.55: AES and EFE_α for different problem sizes n. For all instances $T_{max} = 1.000.000$, $p = 8/n$ and results are averaged over 25 runs.

effort seems necessary as EFE_α and ET_α do not scale up linearly anymore. To get an idea whether ET_α scales up polynomially or exponentially for $p = 8/n$, in figure 6.56 ET_α is approximated with a polynomial function: $\mathrm{ET}(n) = n^5/5.5 \cdot 10^{-6}$ and an exponential function: $\mathrm{ET}(n) = (1.0107)^n$. Clearly, the polynomial function fits better, indicating the EA possibly has polynomial time complexity.

Together with the performance results from this chapter, we see that the EA and DSatur complement each very nicely. For instances up to $n = 500$ (or for bigger instances far from the phase transition), DSatur is fastest and seems the best choice. For bigger and harder problem sizes, the EA performs better both in SR and time.

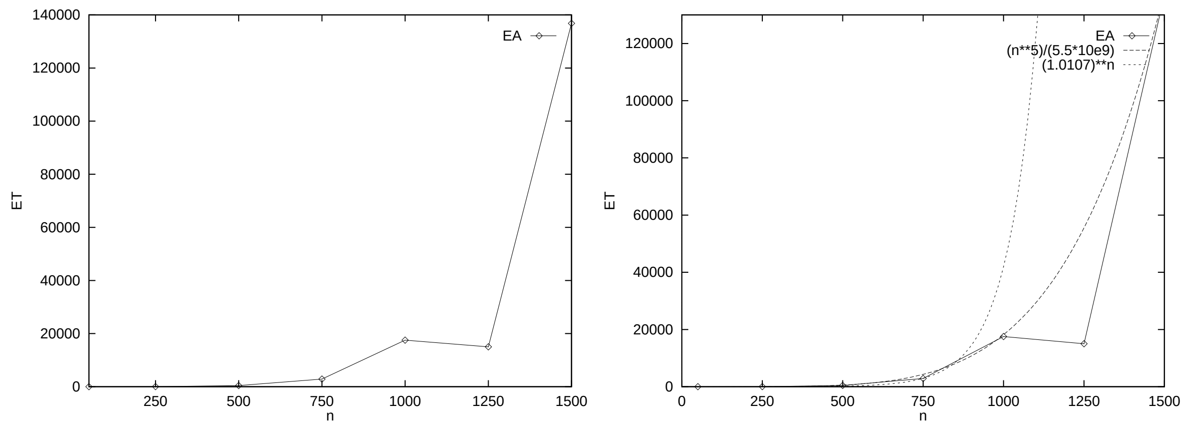


Figure 6.56: ET_α for different problem sizes n for $p = 8/n$. In the right figure a polynomial function ($\mathrm{ET}(n) = n^5/5.5{\cdot}10^{-6}$) and an exponential function ($\mathrm{ET}(n) = (1.0107)^n$ are fitted to the ET_α figure.

6.8 Conclusions

In this section, a summary of the conclusions and results in this chapter is given. It should be noted that these are only verified for the Graph 3-Coloring problem.

EA with Integer Representation

- Using operators with more parents, gives better performance, though is costly.
- Incest prevention gives better performance though is costly.
- The population size is an important parameter and needs to be fine tuned. $\mu = 250$ seems to be optimal for both $n = 200$ and $n = 1000$.
- The difference between recombination operators is small, though with incest prevention, uniform crossover gives best performance.
- An asexual EA using only mutation also performs well.
- For an asexual EA, $\mu = 1$ is optimal!
- An asexual EA with optimal μ gives better performance, and is far faster, than an EA with recombination and optimal μ!

EA with Order-based Representation

- The OX2 recombination operator and the SWAP mutation operator work best for the order-based representation.

- For the EA with OX2, $\mu = 600$ and $\mu = 700$ are optimal for resp. $n = 200$ and $n = 1000$.
- An asexual EA using only SWAP mutation also performs well.
- For an asexual EA, $\mu = 1$ is optimal!
- An asexual EA with optimal μ gives better performance, and is faster, than an EA with recombination and optimal μ!

Comparison of the Algorithms

- DSatur is better than both EA representations.
- Order-based representation is far better than the integer representation.
- The EA with order-based representation comes close to the performance of DSatur for the graphs very close to the phase transition.

Hybrid Evolutionary Algorithms

- The EA is not able to benefit from using the IG heuristics.
- Initializing the order-based EA that uses recombination with permutations found by DSatur, increases the performance.
- Using degree weights for the fitness function, greatly increases the performance for the EA with recombination, but does not work for the asexual EA.
- Hybridizing the EA with recombination with a DSatur-based decoder yields an algorithm that is better than either DSatur or the EA with recombination, but is very time consuming.

Mutation Rate

- For the EA with recombination and a fixed p_m, $p_m = 1/L$, is optimal. For the asexual EA, higher values for p_m (like $p_m = 10/L$) give similar results as for $p_m = 1/L$, showing several changed genes per mutation are not harmful.
- The asexual EA benefits from using a dynamically changing mutation rate, though for the EA with recombination this decreases the performance.

The Stepwise Adaptive Weights Mechanism

- SAW spectacularly increases the performance of the asexual EA and of the EA with recombination (and $\mu \geq 600$), though more for the asexual EA.
- With SAW, the EA with recombination has optimal population size $\mu = 2$, which really makes the recombination operator useless.
- SAW also works for the integer representation.
- With SAW, the asexual EA is better than DSatur for the graphs very close to the phase transition. This shows that the EA with SAW is a worthy algorithm for the Graph 3-Coloring problem.

SAW does not seem to be natural with a dynamically changing mutation rate as in fact the fitness function changes continually.

Main Conclusions

The two main conclusions from this chapter are that:

- The asexual EA, with $\mu = 1$, performs better than the EA with recombination with respect to success rate, average number of evaluations to find a solution and necessary CPU-time.
- The SAW mechanism greatly increases the performance and makes the asexual EA better than the traditional method on the hardest graph instances.

In the next chapter these conclusions will be verified on SAT.

Chapter 7

Verifying The Conclusions On 3-SAT

As the results from chapter 6 are really only valid for Graph 3-Coloring, in this chapter another NP-complete problem, 3-Satisfiablility (3-SAT), is investigated to verify the main results. SAT is chosen as it is an important NP-complete CSP that differs enough in characteristics from Graph Coloring, to be useful for verifying the conclusions.

In the next sections, the problem, known solvers and problem instances will be described. Then the EA will be presented and the most important results from the previous chapter will be verified. The EA will be compared with the best known method and in the last section a transformation from 3-SAT instances to Graph 3-Coloring instances will be discussed.

7.1 Problem Description

Propositional Satisfiability, or SAT, is the problem of finding a truth assignment for the variables in a propositional formula, that makes the formula true. The version examined here, constrains formulas to be in *conjunctive normal form* (CNF), where a formula must be a conjunction of clauses and a clause a disjunction of literals. In 3-SAT we also constrain the clauses to have exactly three literals. In the common notation, a formula has l clauses and n variables.

SAT was the first computational task shown to be NP-hard by Cook [12], but is also of practical interest as it has applications in for example reasoning, diagnosis, planning [49] and image interpretation [64]. Originally SAT was formulated as a decision problem, but here the variant is used where we ask for an actual solution.

The most well known complete algorithm for SAT, DP (Davis-Putnam, which is a backtracking algorithm based on resolution and combined with unit propagation) [18] , can take a time exponential in the length of the formula to run. To overcome this, an incomplete algorithm, GSAT, was proposed by Selman *et al.* [69]. Frank presented a modification of a weighted GSAT version [32], and showed that this modification, WGSAT, has better performance than one of the best known variants of GSAT, HSAT [36], and so the EA will be compared to the WGSAT algorithm.

7.2 Problem Instances

As for the Graph 3-Coloring problem, we need instances that are known to have a solution, because the EA is an incomplete algorithm. To do this, the generator `mkcnf.c` by Allen van Gelder[1] that was loosely based on `mwff.c` by Bart Selman, was used. This generator can force formulas to be satisfiable which should be harder to solve than those generated by `mwff.c`. With it, 3-CNF instances are created that have n variables, l clauses and seed s[2], notated as $S_{n,l,s}$.

Mitchell *et al.* [58] report that the phase transition is found when $l = 4.3 \cdot n$. Because another generator is used, this might not give the exact location of the phase transition for these instances, but as the SAT instances obtained with $l = 4.3$ were hard enough for the algorithms, this does not really matter. Tests are done for $S_{n=50,l=215,s}$ and $S_{n=100,l=430,s}$ and $s = 0 \ldots 10$.

An average over several instances was used because the performance could enormously differ from instance to instance. It is not possible to compare EFE_α when it is averaged over all instances, this because it can be better for an algorithm to have no solutions at all for a specific instance (so the instance is not counted) then to have one solution and a tremendous big EFE_α for that instance. And to a lesser extent we also have this for AES. In fact we are only allowed to compare different settings on the instances where *both* find a solution. So either we remove the instances with no solution or we take a value for l where the algorithms find solutions for all instances. As we want hard problem instances to compare our algorithms on, comparisons are only done with results averaged over instances where all algorithms (in the comparison) find solutions. The specific seeds that were used will be reported.

In the final comparison with WGSAT, the generator that Frank used to compare WGSAT with GSAT and HGSAT was used. This generator creates 3-CNF instances by generating each clause by selecting 3 of n literals without replacement and negating each literal with probability $\frac{1}{2}$. From Frank, we obtained the same 1000 seeds for $n = 100$ as were used in his comparison with HSAT and GSAT [32]. The instances that were generated with these seeds, were proven to have solutions with a variant of the DP procedure. Formulas obtained in this way tend to be somewhat harder than formulas that are forced to be satisfiable.

7.3 The Traditional Algorithms

7.3.1 GSAT

To present GSAT, a more general framework, presented by Gent and Walsh, "GenSAT" [35], that captures all versions of GSAT (and an EA version as we will see in section 7.5), will be used. GenSAT is shown in figure 7.1.

[1]Available by ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances

[2]The shell scripts for mkcnf.c are not used, so the seeds are not redistributed.

```
procedure GenSAT($\Sigma$)
    for i := 1 to Max-tries
        T := initial($\Sigma$)
        for j := 1 to Max-flips
            if T satisfies $\Sigma$ then
                return T
            else
                Poss-flips := hill-climb($\Sigma$,T)
                V := pick(Poss-flips)
                T := T with V's truth assignment flipped
        end
    end
    return "no satisfying truth assignment found"
end
```

Figure 7.1: GenSAT Algorithm.

GSAT performs a greedy local search to find a satisfying truth assignment for a formula Σ. It starts with a randomly generated truth assignment, generated in procedure *initial*. Then *hill-climb* creates a local neighborhood by returning those variables whose truth assignment if flipped gives the *greatest* increase in the number of clauses satisfied and then *pick* chooses one of these variables at random. So GSAT is a local steepest ascent hill-climbing algorithm, but an important feature is the possibility of sideways flips: if no flip exists which improves the solution, then a variable is flipped which does not change the score. Without sideways flips GSAT's performance degrades greatly. Now these flips are repeated until a satisfying assignment is found or a maximum number of flips (Max-flips) is reached. The whole process can be repeated Max-tries times. As a rough guideline, Max-flips is advised to be set to a few times the number of variables. Max-tries is fixed to one for comparing purposes.

By investigating several variants of GSAT, Gent and Walsh [35][37] found that neither greediness nor randomness was essential for the effectiveness of GSAT. As long as the algorithm prefers moves that do not degrade a solution and uses a fair[3] deterministic selection mechanism for choosing from the set of possible flips, similar performance is found. The issue of greediness will be discussed later in the context of the EA. They also introduced a variant, known as HSAT, which uses historical information to choose deterministically which variable to pick. HSAT was found to perform considerably better

[3]A selection is fair if it guarantees to eventually select an element that is continuously in the set of possible selections. Random selection is a fair selection mechanism. It this research it was also observed that random tie breaking has similar performance as circular tie breaking, which is a also fair deterministic selection method, when removing the worst individual from the population. But both are better than a version that broke ties (unfairly) by always selecting the first individual in the population.

than GSAT.

7.3.2 WGSAT

In [68], Selman and Kautz describe a modification in which GSAT associates a weight with each clause. The weights of all clauses that remain unsatisfied at the end of a *try* are incremented. This really is the off-line version of the SAW mechanism! Now Frank adapted this to an on-line version, by updating the weights after each *flip* instead of after each try [32][31] and this is exactly the same as the SAW mechanism, but used in another algorithm. Frank had very good results with this version, which was called WGSAT and proved to be better than HSAT. This shows that the SAW mechanism also works for another algorithm and on SAT. Later it will be shown that WGSAT and GSAT can be seen as a special case of an EA and that ideas from EAs can improve on the performance of WGSAT. Frank also investigated some other interesting ideas that can be used together with the weights idea, but the improvements were not so high and were dependent on the setting of a new parameter.

7.4 Evolutionary Algorithms on 3-SAT

In this section, the EA for SAT will be presented and the conclusions about asexual EA vs. EA with recombination and about the SAW mechanism will be verified. A few other issues will be introduced which were not covered in the previous chapter.

7.4.1 Representation

The most natural representation for SAT, is the bit representation where each variable in the SAT problem is represented by a gene, that can have a value 0 for false and 1 for true. Then an individual can directly (without a decoder) be interpreted as a truth assignment. The fitness function just counts the unsatisfied clauses, so again we obtain a minimization problem. De Jong and Spears [48] also used a bit representation for GAs on SAT, but they used a different evaluation function than the above one, because they experimented with more general CNF (not necessarily 3-CNF) formulas. Hao [44] used a clausal representation, where each clause in the SAT problem is represented by a group of three genes and some adapted operators are used. As this would give individuals with length $L = 1290$ instead of $L = 100$ for an instance with $n = 100$ and because he did not report any results in [44], that representation will not be used. Next will be shown that the order-based representation is not suited to SAT and therefore the bit representation will be used in the following sections. For each instance 50 runs with $T_{max} = 300.000$ are done. The seeds that are used in a comparison will be reported.

Order-based Does Not Represent SAT

It's very hard to represent SAT solutions as permutations, as for the most natural decoders it's always possible to create a problem instance so that no permutation exists that can be decoded to a solution.

The most natural decoder seems to be the following one (which was also used by Gent and Walsh for their OSAT version of GSAT). The idea is to create a function that evaluates partial truth assignments just by ignoring uninstantiated variables and give variable x_i (the i^{th} variable in the permutation x) the value that maximizes the number of satisfied clauses in the CNF formula that is obtained by instantiating all x_j with $0 < j < i$ to their (in this way) found values and leaving out al literals that contain variables x_k with $i < k \leq n$, so for example for x_1, the CNF formula will consist only of literals that contain x_1.

This decoder would only be useful if always a permutation exists for which it will create the satisfying truth assignment (if it exists). Unfortunately this is not true as is proven by the following 3-CNF formula with $3n-5$ clauses:

$$\begin{array}{lllllllll}
(\neg x_1 \vee \neg x_1 \vee \neg x_1) & \wedge (x_2 \vee x_1 \vee x_1) & \wedge \dots & \wedge (x_i \vee x_1 \vee x_1) & \wedge \dots & \wedge (x_{n-1} \vee x_1 \vee x_1) \\
 & \wedge (\neg x_2 \vee x_n \vee \neg x_n) & \wedge \dots & \wedge (\neg x_i \vee x_n \vee \neg x_n) & \wedge \dots & \wedge (\neg x_{n-1} \vee x_n \vee \neg x_n) \\
 & \wedge (\neg x_2 \vee x_n \vee \neg x_n) & \wedge \dots & \wedge (\neg x_i \vee x_n \vee \neg x_n) & \wedge \dots & \wedge (\neg x_{n-1} \vee x_n \vee \neg x_n)
\end{array}$$

Obviously, the only satisfying truth assignment is found when $x_1 = 0$, $x_i = 1$ for $1 < i < n$ and any value for x_n. Now for $n \geq 5$, for *any* ordering, the first variable in the ordering will always be assigned the wrong value (except for x_n) so that a satisfying assignment can never be found with this decoder. The correct truth assignment would be found by *minimizing* the fitness for the partial assignments, but a similar formula can be constructed for a decoder that always minimizes the fitness. Possibly decoders can be created that avoid this problem, but more probably the order-based representation is not suited to the SAT problem as there does not seem to be any advantage for a variable to be assigned a value very early (or lately).

7.4.2 Population Size and Operator Comparison

In this section, an EA with the best recombination operators for Graph 3-Coloring: uniform, DIAG and SCAN and an asexual EA (with similar mutation operator as for the integer representation and $p_m = \frac{1}{L}$[4]) will be fine tuned with respect to their population size and will then be compared. As, like for Graph Coloring, the distance in position between two genes does not seem to say anything about the relationship between the two genes, 1- and 2-point crossover are not considered. For DIAG and SCAN, only 15 parents will be considered because of time constraints.

As figure 7.3 shows, for mutation alone for $n = 50$[5] and $n = 100$[6], obviously again,

[4]Though one gene is always flipped to obtain a new value because possibly an operator that does not change anything is not interesting for the asexual EA and more importantly, to compare better with GSAT.

[5]For mutation alone, seeds 0,2,3,5,7,8 and 9 and for the other operators, seeds 2,3,7,8 and 9 were used for $n = 50$.

[6]For both operators, only seeds 2 and 8 were used for $n = 100$.

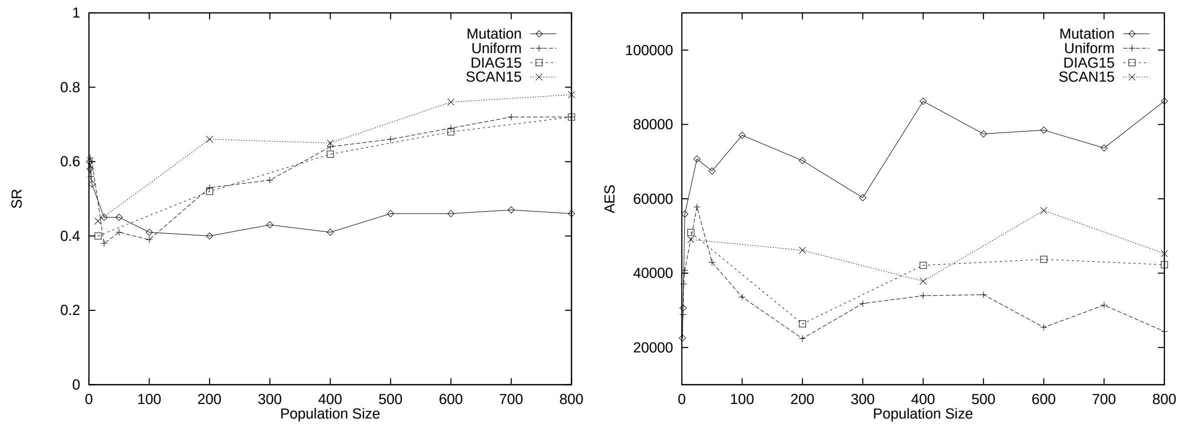


Figure 7.2: Fine tuning of population size for integer representation operators $n = 50$.

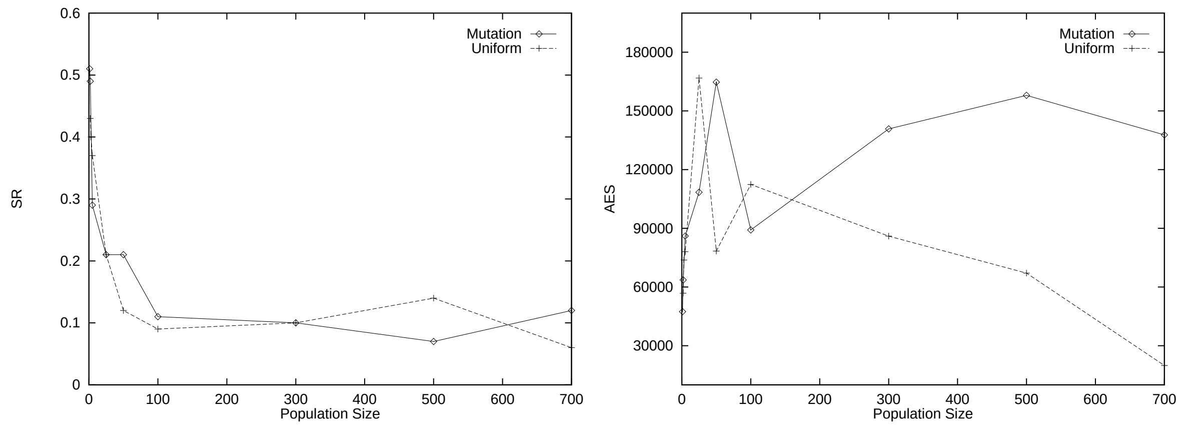


Figure 7.3: Fine tuning of population size for integer representation operators $n = 100$.

$\mu = 1$ gives the best results, though it is interesting that for SAT now the success rate starts increasing a little for $n = 50$ and a big population size, showing that also an asexual EA is able to benefit a little from a population.

For the other operators, for $n = 50$, clearly $\mu = 800$ (and possibly bigger) for DIAG15 and SCAN15 is optimal, though this comparison is not really fair as for $\mu = 15$ both SCAN and DIAG are able to solve 9 of the 10 instances, whereas for bigger μ only 5 of the 10 instances are solved. The strange thing is that some instances greatly benefit from a bigger population size and some instances have SR decrease from about 20% to 0%. This shows that it's hard to do a comparison with averages over different instances. For uniform crossover, the optimal population size is also $\mu = 800$ and again there is a minimal population size under which it is useless to have a population at all. Though for $n = 100$,

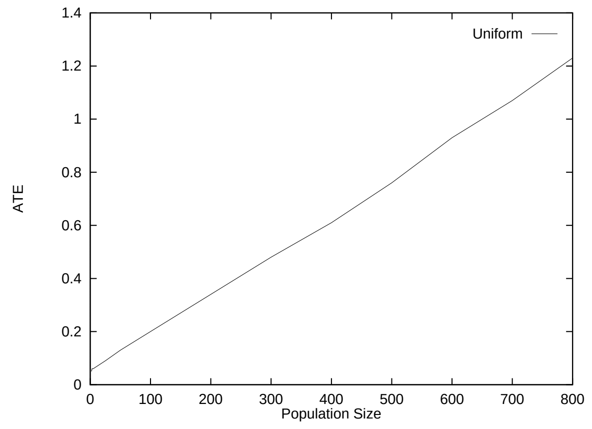


Figure 7.4: Time per evaluation step against population size for uniform and $n = 50$.

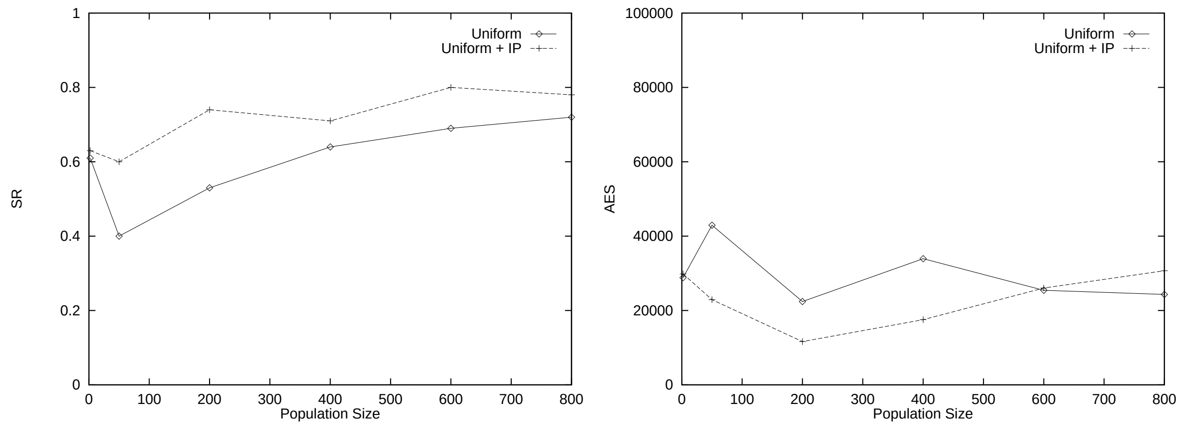


Figure 7.5: Effect on Incest Prevention for varying population sizes for $n = 50$.

for uniform crossover, the optimal population size is 2, possibly because of too small a value for T_{max} (though this does not follow from AES and its standard deviation).

Although SCAN15 has the best success rate for $n = 50$, followed closely by DIAG and uniform, as figure 7.4 shows, the time per evaluation (ATE) almost doubles as μ doubles, so that ATE=0.05 for $\mu = 2$ and ATE=1.23 for $\mu = 800$. Because of this, the asexual EA would have far higher SR when allowed to take the same total runtime (e.g. by restarting). Also because not all instances can be used in the comparison, we do not see that uniform crossover with $\mu = 2$ sometimes finds solutions for more instances, than with higher μ. Further for $n = 100$, the optimal populations size is 2 and therefore it could more efficient to choose $\mu = 2$ for uniform crossover, but then it's even better to omit the recombination and use $\mu = 1$.

	SR	AES	EFE_α	ET_α
mutation($\mu = 1$)	0.66	23568	21133	0.87
uniform+IP($\mu = 600$)	0.80	26041	104455	50.96
DIAG15($\mu = 800$)	0.72	42261	127196	82.79
SCAN15($\mu = 800$)	0.77	45263	128861	81.95

Table 7.1: Comparing fine tuned operators for $n = 50$.

	$n = 50$		$n = 100$	
	SR	AES	SR	AES
Mutation	0.47	25934	0.30	43483
MutOne	0.35	6472	0.30	40147
MutOne_1^+	0.48	17857	0.34	56619
MutOne_2^+	0.53	14832	0.33	52388

Table 7.2: Performance of various mutation operators.

Still it's interesting to see the effect of Incest Prevention on uniform crossover and as shown in figure 7.5[7], like for Graph 3-Coloring, IP again is able to significantly improve the performance (both SR and AES), and now the optimal population size becomes $\mu = 600$, though the difference with $\mu = 800$ is not very big.

The most striking result from table 7.1[8] is that the performance not really differs spectacularly among the different operators. We also saw this for the integer representation with Graph 3-Coloring. Still if we consider time and the fact that mutation alone with $\mu = 1$ finds solutions for more instances than the other operators and that $\mu = 2$ is optimal for uniform crossover and $n = 100$, the asexual EA should get the best credits and therefore it is interesting to try a few variations on the mutation operator.

As was done before, a mutation operator that always changes exactly one gene per mutation (MutOne) was tested, this because it's completely useless to mutate zero genes in an asexual EA with preservative selection. As the results in table 7.2[9] show, AES decreases strongly, which could be the effect of omitting the mutations that do not change any gene (which is about 36% of the mutations). But SR drops, which indicates that it can be useful to mutate more than one gene per mutation. Therefore two other variants are tested that mutate at least one gene. The first variant, MutOne_1^+, uses the old probability distribution to redistribute the probability of zero mutated genes over the other probabilities. So the probability of mutating one gene increases more than the probability of mutating two

[7]Results for IP were based on seeds 2,3,7, 8 and 9.

[8]Seeds 2,3,7,8 and 9 are used.

[9]For $n = 50$, seeds 0,2,3,4,5,6,7,8 and 9 are used. For $n = 100$, seeds 2,3,4,5 and 8 are used.

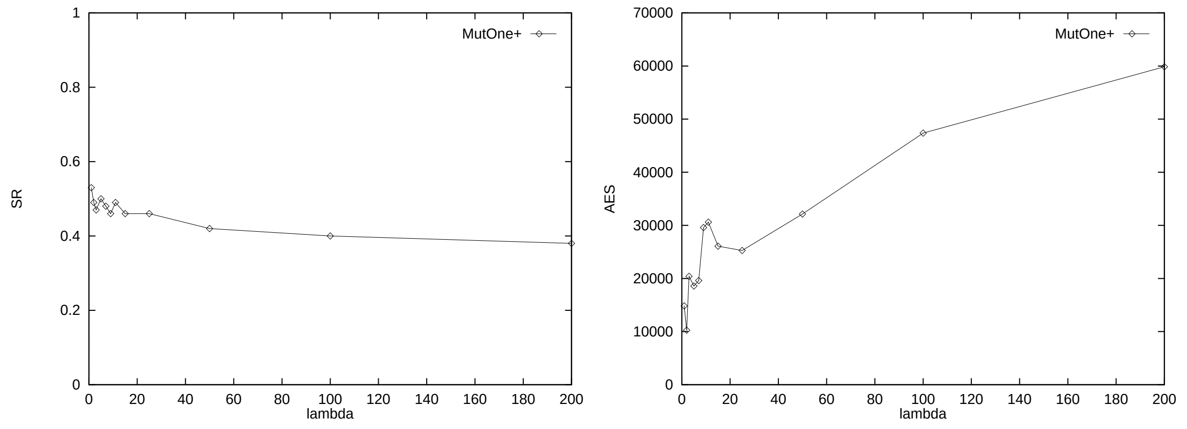


Figure 7.6: Performance for $(1+\lambda)$-selection and varying λ for $n = 50$.

genes, etc. The second variant, MutOne$^+_2$, adds the probability of mutating zero genes to the probability of mutating one gene. So the probability of mutating one gene is now about 70%. Both MutOne$^+_1$ and MutOne$^+_2$ have better results than the old mutation which shows the benefit of leaving out useless mutations, but keeping a probability to changing more than one gene. Though MutOne$^+_2$ has slightly better performance than MutOne$^+_1$, indicating that a change of more than one gene per mutation should not occur too often. As only MutOne$^+_2$ will be used in later sections, it will be denoted as MutOne$^+$.

7.4.3 Extinctive Selection

Because the asexual EA will be used from now on, it is interesting to test the effect of the (1,λ)-strategy with extinctive selection. To be able to to fairly compare it with (1+λ)-strategy, for preservative selection λ is also optimized. As figure 7.6[10] shows, $\lambda = 1$ is the optimal offspring population size for (1+λ)-selection which supports the argument given in section 6.6.2.

Figures 7.7 to 7.9[11] show the performance for varying values of λ for extinctive selection for the three mutation operators. The optimal offspring population sizes (λ^*) are shown in table 7.3

A problem with the extinctive selection schemes is that their performance is highly dependent on the value of λ and the optimal value for λ does not really seem to scale up. The reason for the difference in λ^* between the operators is of course that a more disruptive operator needs more offspring with extinctive selection to ensure that a good child is produced. Because the standard mutation can *copy* a parent to the offspring population in about 36% of the cases, it is the least disruptive. MutOne$^+$ is slightly more disruptive than MutOne, because it can change more than one gene.

[10] For MutOne$^+$, seeds 0,2,3,4,5,6,7,8 and 9 are used.

[11] For al operators, seeds 0-9 are used for $n = 50$ and seeds 2,3,4,5 and 8 for $n = 100$.

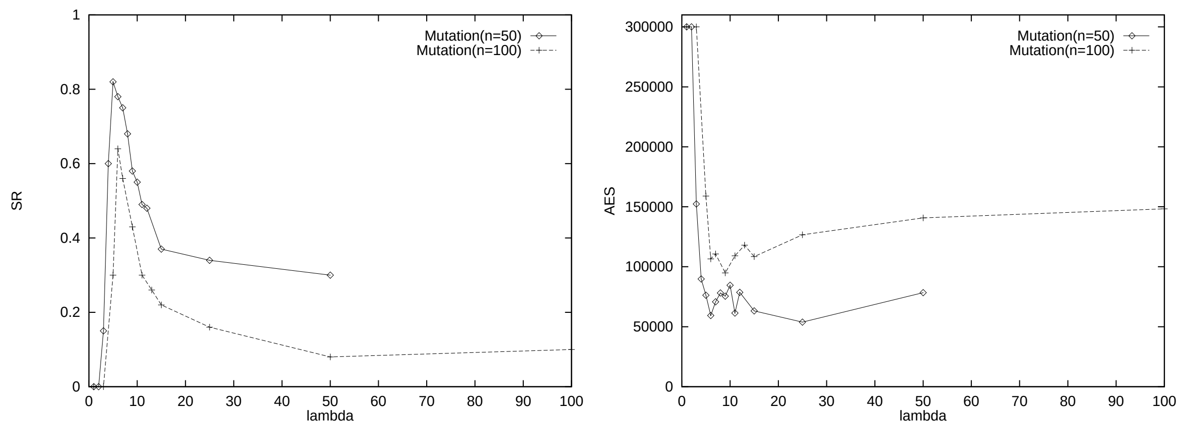


Figure 7.7: Fine tuning of λ with $(1, \lambda)$ selection for Mutation operator for $n = 50$ and $n = 100$.

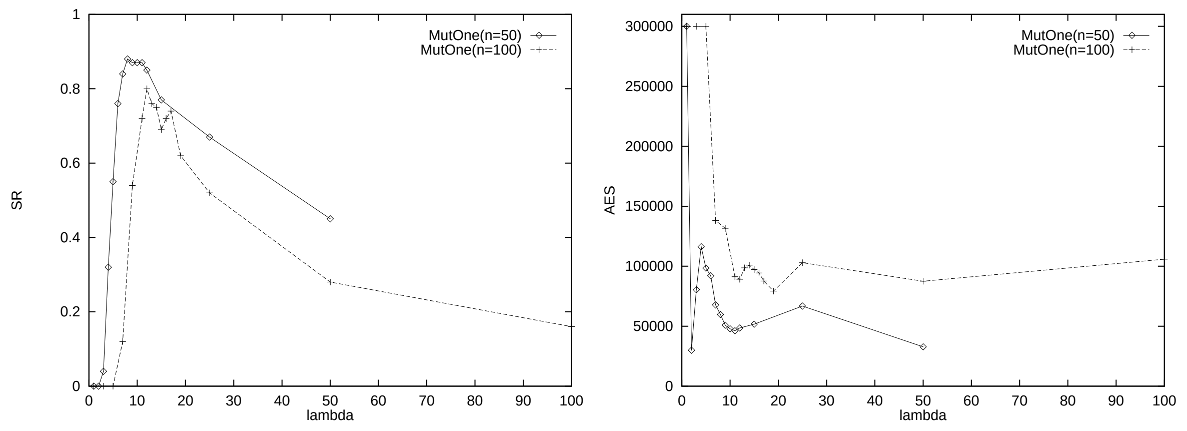


Figure 7.8: Fine tuning of λ with $(1, \lambda)$ selection for MutOne for $n = 50$ and $n = 100$.

Table 7.4[12] compares the performance of extinctive selection against preservative selection and it shows that for SAT extinctive selection with optimal λ is better than preservative selection when $\mu = 1$ (though AES increases). Still the high dependency on λ might make the extinctive scheme less useful in a practical application. Further we see that MutOne$^+$ is not consistently the best operator anymore when extinctive selection is used, as for $n = 100$ MutOne is slightly better. Though both are better than the normal Mutation.

[12]For $n = 50$, seeds 0-9 are used. For $n = 100$, seeds 1,2,3,4,5 and 8 are used.

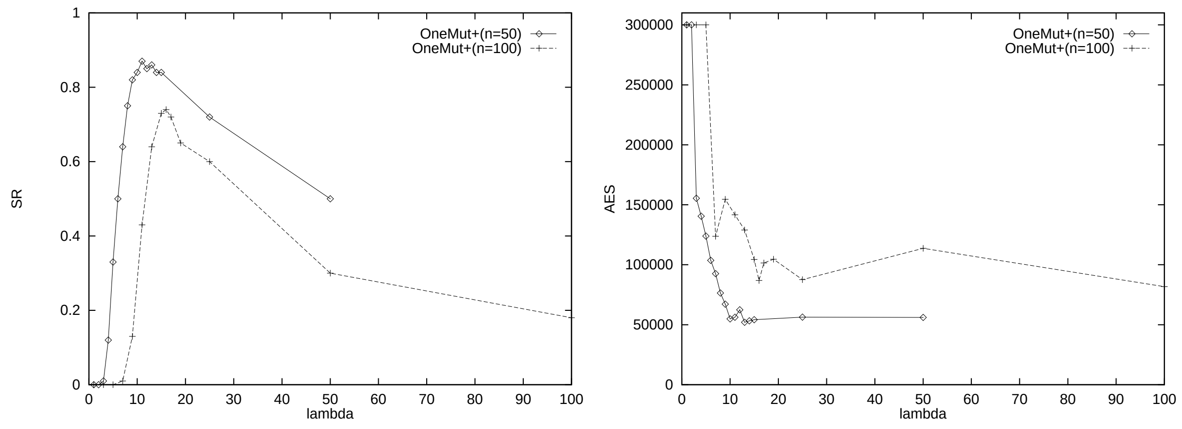


Figure 7.9: Fine tuning of λ with $(1,\lambda)$ selection for MutOne$^+$ operator for $n = 50$ and $n = 100$.

	λ^*	
	$n = 50$	$n = 100$
Mutation	5	6
MutOne	10	12
MutOne$^+$	11	16

Table 7.3: Optimal offspring population sizes for extinctive selection.

	$n = 50$		$n = 100$	
	SR	AES	SR	AES
Mutation (1+1)	0.47	25934	0.25	43483
Mutation (1,λ^*)	0.89	58943	0.54	109571
MutOne (1+1)	0.31	6472	0.25	40147
MutOne (1,λ^*)	0.87	47847	0.67	75263
MutOne$^+$ (1+1)	0.53	14832	0.28	47554
MutOne$^+$ (1,λ^*)	0.94	44223	0.62	75107

Table 7.4: Extinctive versus preservative selection. For Mutation, $\lambda^* = 5$ for $n = 50$ and $\lambda^* = 6$ for $n = 100$. For MutOne, $\lambda^* = 10$ for $n = 50$ and $\lambda = 12$ for $n = 100$. For MutOne$^+$, $\lambda^* = 11$ for $n = 50$ and $\lambda^* = 16$ for $n = 100$.

7.4.4 SAW on SAT

Very interesting is whether the SAW mechanism also works for other problems and if the same conclusions about the parameters T_p and $\triangle w$ hold.

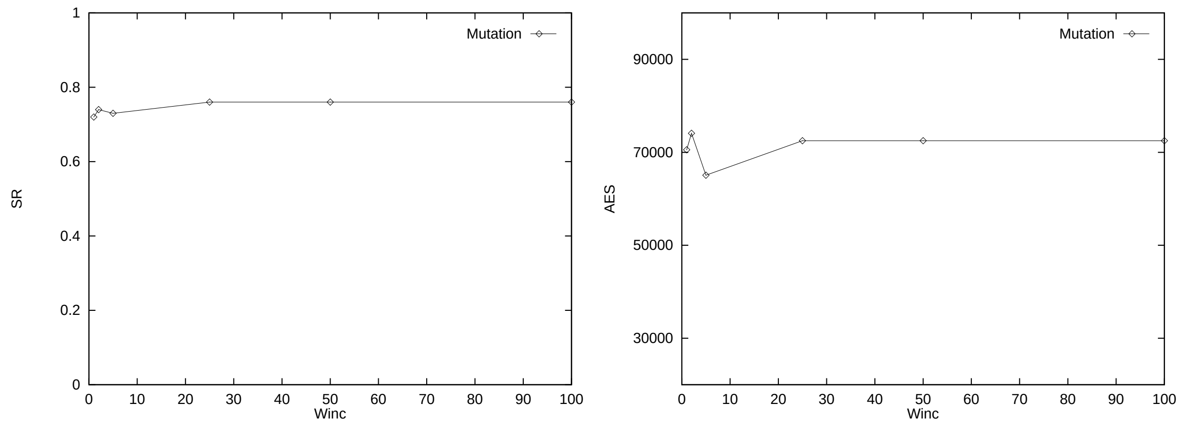


Figure 7.10: Effect of varying $\triangle w$ for SAT for $n = 50$. Normal mutation with $\mu = 1$ and $T_p = 250$ is used.

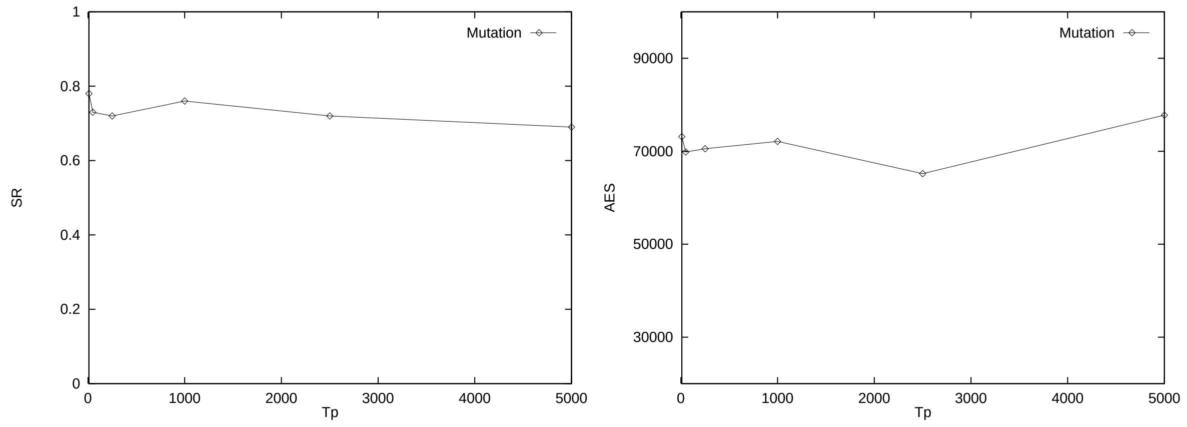
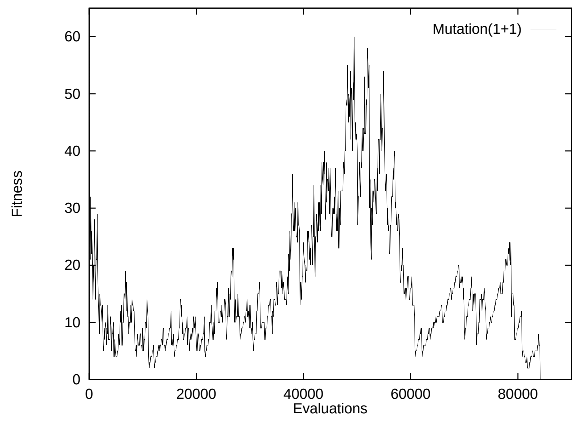


Figure 7.11: Effect of varying T_p for SAT for $n = 50$. Normal mutation with $\mu = 1$ and $\triangle w = 1$ is used.

Figure 7.10 shows that also for SAT, $\triangle w$ does not have a significant effect on the performance and like for Graph 3-Coloring, the results for $\triangle w = 25$ were exactly the same as for $\triangle w = 50$ and $\triangle w = 100$, showing again it's not necessary to keep $\triangle w$ as a parameter. Therefore $\triangle w$ is fixed to one.

Figure 7.11 shows that the period does not affect the performance significantly as long as it is not too big (e.g. $T_p = 5000$), because then hardly any weight updates remain. This confirms again the conclusions for Graph 3-Coloring about the robustness of the SAW mechanism and for no specific reason $T_p = 250$ will be used.

The fitness curves in figure 7.12, like for Graph 3-Coloring, also show an *exploration*



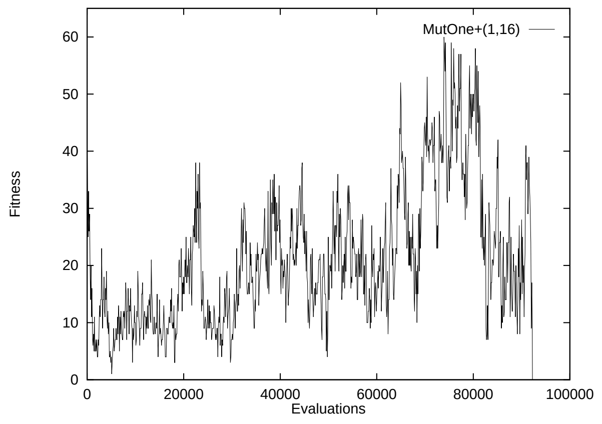


Figure 7.12: Fitness curves for SAW with preservative(left) and extinctive(right) selection.

phase and an *exploitation* phase, where the plot for extinctive selection oscillates stronger, which is natural as the individual (and so the fitness) has bigger probability of changing per generation.

In table 7.5[13] the effect of the SAW mechanism is shown for several EA settings and as can be seen, for all settings SAW is able to improve on the performance.

Firstly, it can be seen that SAW works also for uniform crossover, both without and with a population. Though the results are somewhat better when a small population is used. Interesting could be to try a variant of SAW that bases a weight adaptation on several individuals in the population.

Secondly and surprisingly, for preservative selection the normal mutation operator now seems to be the best operator when SAW is used. This indicates that it can be useful just to copy an individual from time to time which would have the same effect as reducing T_p. Perhaps $T_p = 250$ is too big to gradually guide the EA to a solution and we should choose T_p smaller. Indeed figure 7.11, shows a slight improvement when $T_p = 10$ was used but this did not seem significant as $T_p = 1000$ also gave slightly better results than $T_p = 250$. This should be tested in further research, with possibly $T_p = 1$ being the best period for the asexual EA and thereby loosing a parameter again. For preservative selection, a disadvantage is that the AES strongly increases, which is perhaps the price to be paid.

Thirdly, for extinctive selection and SAW, the MutOne operator has better performance than the MutOne$^+$ operator. We saw already that for extinctive selection it is not clear anymore if the MutOne or MutOne$^+$ mutation is better but when SAW is used, MutOne clearly wins. Perhaps the more disruptive MutOne$^+$ operator disturbs the gradual guiding process of SAW. As T_p might have a different effect for extinctive selection, also a test was run with $T_p = \lambda^*$, the smallest period that makes sense, which gave significant improvement on SR and AES. This confirms the previous observation that a smaller period

[13]For $n = 50$, seeds 0,2,3,5,7,8 and 9 are used. For $n = 100$, seeds 2,4,5 and 8 are used.

	$n = 50$		$n = 100$	
	SR	AES	SR	AES
Uniform+Mutation(2+2)	0.57	30713	0.36	29714
Uniform+Mutation(2+2)+SAW	0.95	43032	0.49	128770
Uniform+Mutation(600+2)+IP	0.62	27503	0.13	56896
Uniform+Mutation(600+2)+IP+SAW	0.75	73731	0.23	97282
Mutation(1+1)	0.60	22548	0.37	51744
Mutation(1+1)+SAW	0.95	41473	0.54	119294
MutOne(1+1)	0.44	8185	0.36	42109
MutOne(1+1)+SAW	0.85	34932	0.46	129709
MutOne$^+$(1+1)	0.67	18294	0.40	48551
MutOne$^+$(1+1)+SAW	0.91	48623	0.51	104520
Mutation(1,λ^*)	0.98	39494	0.74	102193
Mutation(1,λ^*)+SAW	1.00	23202	0.85	103649
Mutation(1,λ^*)+SAW($T_p = \lambda^*$)	1.00	21167	0.86	102170
MutOne(1,λ^*)	1.00	17816	0.88	77964
MutOne(1,λ^*)+SAW	1.00	11642	0.94	72763
MutOne(1,λ^*)+SAW($T_p = \lambda^*$)	1.00	11621	0.96	66940
MutOne$^+$(1,λ^*)	0.99	27836	0.82	87461
MutOne$^+$(1,λ^*)+SAW	1.00	22964	0.86	108627
MutOne$^+$(1,λ^*)+SAW($T_p = \lambda^*$)	1.00	14444	0.92	78019

Table 7.5: Results for SAW. When not stated otherwise, $T_p = 250$. For extinctive selection and normal mutation, $\lambda^* = 5$ for $n = 50$ and $\lambda^* = 6$ for $n = 100$. For MutOne, $\lambda^* = 10$ for $n = 50$ and $\lambda^* = 12$ for $n = 100$. For MutOne$^+$, $\lambda^* = 11$ for $n = 50$ and $\lambda^* = 16$ for $n = 100$.

might be better for SAW to gradually guide the asexual EA through the search space. If it's consistently better to choose $T_p = \lambda$, then a parameter of the SAW mechanism is lost which would be a great advantage. Again, with the right parameters settings, extinctive selection outperforms preservative selection on SAT.

So concluding this section, we saw that again SAW is robust with respect to its parameters, SAW is able to improve the performance for various EA settings and that with SAW, normal Mutation is seems the best operator for preservative selection, MutOne is the best operator for extinctive selection and for extinctive selection we should choose $T_p = \lambda$.

	$n = 50$		$n = 100$	
	SR	AES	SR	AES
DO	0.31	27359	0.18	81894
MutOne	0.50	58919	0.20	119557

Table 7.6: Results for different mutations and (1,n)-strategy.

7.5 The EA versus (W)GSAT: The Last EA Features?

For the SAT problem, two solution approaches were proposed. GSAT came from the AI field and is seen as a local hill-climber. The asexual EA in this report started from the EA field. Now the asexual EA is very similar to GSAT and in fact it's easy to show that GSAT (and so also WGSAT) is a special case of an EA.

If we define a special asexual operator, the Distinct Offspring (DO) operator, that produces n distinct offspring that all differ exactly one gene from their parent, then GSAT is an EA with (1,n)-selection and the DO operator.

Similarly we can see an asexual EA with extinctive selection and $\mu = 1$ as a special case of the GenSAT algorithm (see figure 7.1) if we set Max-flips to T_{max}, Max-tries=1, *initial* gives a random truth assignment, *hill-climb* just randomly selects λ variables and *pick* selects the best individual and breaks ties randomly.

Because the EA framework is more general than the GenSAT algorithm, GSAT will be seen as a special EA.

7.5.1 Comparison with GSAT

As GSAT is in fact an EA with (1,n)-strategy and the DO operator, it is interesting to compare it with an EA with an (1,n)-strategy that uses the MutOne operator. The only difference is that with DO, all children are different, whereas for MutOne redundant children can be produced. As both change only one gene in each child, DO ensures that the flip that gives maximal fitness is performed so the EA with DO essentially is a *steepest ascent* hill-climber, which does not hold for the EA with MutOne.

As table 7.6[14] shows, it is better *not* to use steepest ascent hill-climbing. This was also found by Gent and Walsh [37], who discovered that *greediness* is unimportant and could even degrade the performance. Without steepest ascent hill-climbing, the search will be slower but will not be trapped as easily by local minima.

Now we know from earlier experiments that a small value for λ is optimal so that the performance of the MutOne operator can be spectacularly increased even further. The results with optimal λ^* are presented in table 7.7[15].

[14]For $n = 50$, seeds 0,2,3,4,5,6,7,8 and 9 are used. For $n = 100$, seeds 2,4,5 and 8 are used.

[15]For $n = 50$, seeds 0,2,3,4,5,6,7,8 and 9 are used. For $n = 100$, seeds 2,3,4,5 and 8 are used.

	$n = 50$		$n = 100$	
	SR	AES	SR	AES
GSAT≡DO(1,n)	0.31	27359	0.14	81894
MutOne (1,n)	0.50	58919	0.16	105856
MutOne (1,λ^*)	0.92	40109	0.80	89166
MutOne$^+$ (1, λ^*)	0.94	44223	0.74	86894
MutOne$^+$ (1+1)	0.53	14832	0.33	52388

Table 7.7: Improvements on GSAT. For MutOne, $\lambda^* = 10$ for $n = 50$ and $\lambda^* = 12$ for $n = 100$. For MutOne$^+$, $\lambda^* = 11$ for $n = 50$ and $\lambda^* = 16$ for $n = 100$.

When extinctive selection is used, MutOne$^+$ does not give a consistent improvement over MutOne and so is of no use for improving the performance further. Also preservative selection is able to improve on the performance of GSAT and for preservative selection (without SAW) we saw that MutOne$^+$ was the best operator (see table 7.2). Obviously, the scheme with extinctive selection gives best performance. But as it is highly dependent on the right parameter settings, it might be better in a practical situation to use the preservative (1+1)-scheme which also significantly improves on the performance of GSAT and is more robust.

So we can conclude that the performance of GSAT can be increased by the following changes:

- Avoiding steepest ascent hill-climbing by allowing duplicates in the λ offspring.
- Producing $\lambda < n$ offspring, with best performance for λ^* offspring.

7.5.2 Comparison with WGSAT

As WGSAT is exactly the same as an EA with (1,n)-strategy, DO operator and the SAW mechanism with $T_p = n$, we can follow the same line of reasoning as in section 7.5.1 to see if the conclusions also hold under the SAW mechanism.

Table 7.8[16] confirms that the conclusions from the previous section also hold when SAW is used. So it's better not to use steepest ascent hill-climbing and it's better to use $\lambda < n$. Also, using $T_p = \lambda^*$ seems to give a better overall performance. Again preservative selection, which is more robust, is also able to significantly improve on the performance of WGSAT. In the next section, a bigger test will be done to check the improvement on WGSAT.

[16]For $n = 50$, seeds 0–9 are used. For $n = 100$, seeds 2,3,4,5 and 9 are used.

	$n = 50$		$n = 100$	
	SR	AES	SR	AES
WGSAT≡DO(1,n)+SAW ($T_p = n$)	0.60	84096	0.23	150985
MutOne(1,n)+SAW ($T_p = n$)	0.62	75275	0.25	126418
MutOne(1,λ^*)+SAW ($T_p = n$)	0.97	34498	0.75	116557
MutOne(1,λ^*)+SAW ($T_p = \lambda^*$)	0.95	37027	0.94	73350
MutOne$^+$(1,λ^*)+SAW ($T_p = \lambda^*$)	0.97	35230	0.86	90836
Mutation (1+1) + SAW ($T_p = 250$)	0.72	70557	0.46	115679

Table 7.8: Improvements on WGSAT. For MutOne, $\lambda = 10$ for $n = 50$ and $\lambda = 12$ for $n = 100$. For MutOne$^+$, $\lambda = 11$ for $n = 50$ and $\lambda = 16$ for $n = 100$. For Mutation, $\lambda = 5$ for $n = 50$ and $\lambda = 6$ for $n = 100$.

	SeedSet1		SeedSet2	
	SR	AES	SR	AES
WGSAT≡DO(1,100)+SAW	0.30	110507	0.53	119262
MutOne(1,λ^*)+SAW	0.48	70182	0.88	73166
Mutation(1+1)+SAW	0.37	88359	0.68	97751
MutOne$^+$(1+1)+SAW	0.41	76351	0.74	81112

Table 7.9: Comparison between EA and WGSAT. $T_{max} = 300.000$, $n = 100$, $l = 430$ and results are averaged over 10 runs. For preservative selection, $T_p = 250$, for extinctive selection $T_p = 12$ and $\lambda^* = 12$ and for the DO operator $T_p = 100$.

7.6 The Final Tests

In this section, a big comparison between WGSAT and the EA will be done using the same 1000 satisfiable instances (SeedSet1) as Frank used in his comparison with HSAT [32] (see section 7.2 for a description). SeedSet2 consists of 1000 random instances for the same generator, so not necessarily all instances are satisfiable. The results are given in table 7.9.

Clearly, both EA versions (preservative and extinctive) greatly improve on the performance of WGSAT. Again the extinctive EA outperforms the preservative EA, though the latter is more robust with respect to its parameters. For this bigger test, we see now that for preservative selection with SAW, MutOne$^+$ gives better performance than the normal mutation, contrary to what was found in section 7.4.4, showing that changing at least one gene per mutation gives better performance.

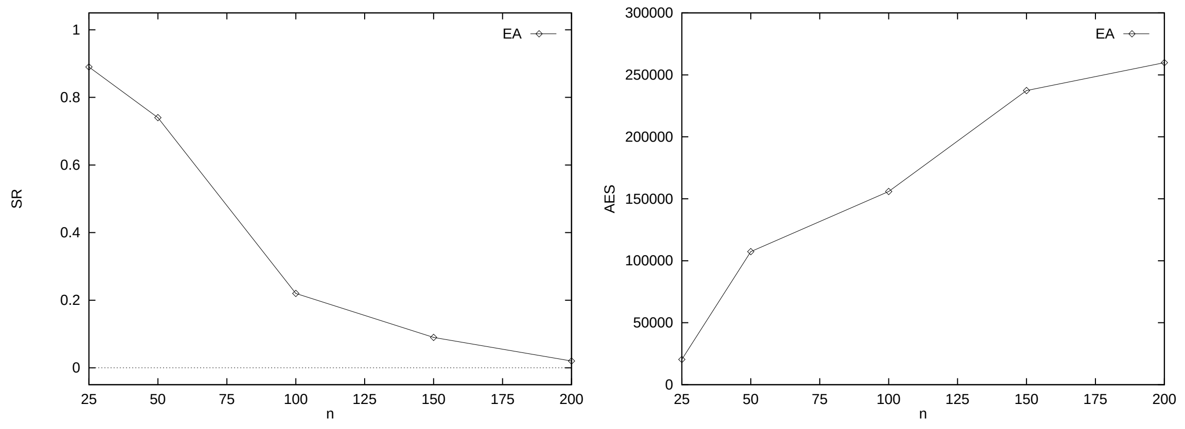


Figure 7.13: Scalability with n for SR and AES for asexual, preservative EA with SAW. $T_{max} = 500.000$ and the results were averaged over 50 runs and over 10 SAT instances.

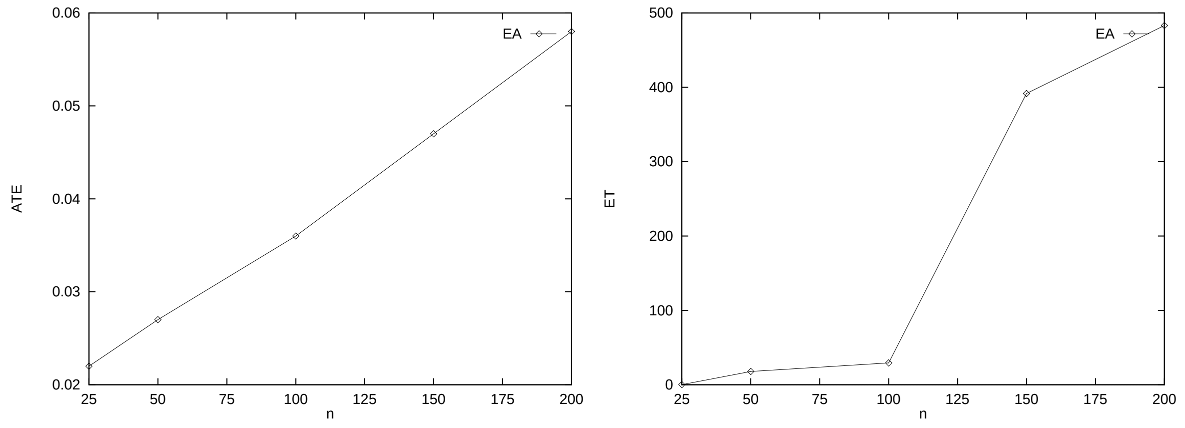


Figure 7.14: Scalability with n for ATE and ET$_\alpha$ for asexual, preservative EA with SAW.

7.6.1 Scalability

Again, interesting is how the performance of the EA scales up for the Satisfiability. As WGSAT and GSAT are in fact inferior EAs, their scalability has not been tested. This because it is not expected that it will give better results than the best EA. Though the extinctive EA has better performance, because of its dependency on a correct value of λ and because λ^* does not scale up, preservative selection will be used in this section.

In figure 7.13 we see, that the success rate decreases with n. Still it does not drop abruptly to 0.0 as for a backtracking approach like, DSatur for Graph 3-Coloring, but decreases almost linearly. We do not see however that for many of the 10 instances that the results were averaged over, no solutions were found at all which casts some shadow

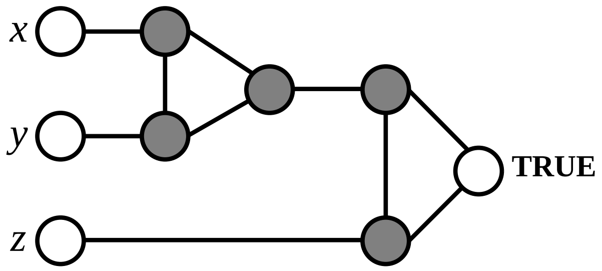


Figure 7.15: Widget for transformation from 3-SAT to Graph 3-Coloring.

over the EA's scalability for SAT.

AES again seems to scale up linearly, which is an advantage as it facilitates the choice of T_{max} for bigger n. As ATE (as explained before) scales up linearly, only the results for ET_α are shown as it will have the same time complexity as EFE_α. It can be seen that ET_α, the expected time to find a solution (if one exists) with certainty α, scales up superlinearly but possibly polynomially. The unexpected results for $n = 200$ possibly come because of the few solutions that were found for $n = 150$ and $n = 200$ and because EFE_α is not defined when SR=0.0.

7.7 Transforming 3-SAT to Graph 3-Coloring

It is known that NP-complete problems can be transformed in polynomial time in other NP-complete problems [33][60]. Theoretically this is important as it means that finding a polynomial algorithm for one NP-complete problem, is finding a polynomial algorithm for all NP-complete problems. In practice however, the transformations often greatly increase the problem size and therefore are not of so much practical use. This can be considered lucky for the EA field, as we still need a specific problem solver for each problem and EA's can be applied to a new problem with a minimal amount of effort while still giving good performance.

To illustrate the effect of transforming problems, we use the following transformation from Cormen *et al.* [13] for a 3-SAT problem to a Graph 3-Coloring problem. Given a formula ϕ of l clauses on n variables, $x_1, x_2, \ldots, x_n$, we construct a graph $G = (V, E)$. The set V consists of a vertex for each variable, a vertex for the negation of each variable, 5 vertices for each clause and 3 special vertices: TRUE, FALSE and RED. The edges of the graph are of two types: "literal" edges that are independent of the clauses and "clause" edges that depend on the clauses. The literal edges form a triangle on the special vertices and also form a triangle on $x_i, \neg x_i$ and RED for $i = 1, 2, \ldots, n$. The widget shown in figure 7.15 is used to enforce the condition corresponding to a clause $(x \vee y \vee z)$. Each clause requires a unique copy of the 5 dark vertices in the figure and they connect as shown to the literals of the clause and the special vertex TRUE.

	SR	AES	ATE
DSatur	0.00	500000	2.41
GC EA	0.16	376870	1.85
SAT EA	1.00	3194	0.06

Table 7.10: Results on transformed and original SAT instance. The SAT instance with $n = 50$, $l = 215$ and $s = 8$ was transformed to a Graph 3-Coloring problem with $n = 1178$ and 2303 edges. All use $T_{max} = 500.000$ and results are averaged over 50 runs. The SAT EA was the asexual extinctive EA with MutOne operator, the GC EA was the asexual preservative EA with SWAP operator.

If the SAT problem has n variables and $4.3 \cdot n$ clauses, then the transformation will produce $23.5 \cdot n + 3$ nodes and $46 \cdot n + 3$ edges. So a relatively easy SAT instance with $n = 50$, would produce a 1178 node Graph Coloring problem, which even for a polynomial algorithm would take a relatively long time. Possibly the transformation moves the problem away from the phase transition or brings structure in the transformed instance, making it easier to solve compared to other instances of that size. Though as shown in table 7.10, transforming a SAT problem instance and solving it with a Graph Coloring algorithm gives far worse performance than solving the original instance directly. So transformations are of little practical use except for creating hard problem instances. Still interesting is that the EA for Graph Coloring is able to solve the transformed instance, whereas DSatur is not.

So in practice we still have to create specific problem solvers for each problem apart. This shows the importance of a general algorithmic framework, like an EA, that allows us with a minimal amount of domain knowledge to obtain a good algorithm!

7.8 Conclusions

In this chapter, the main results from the previous chapter were verified for 3-Satisfiability, which can be summarized as follows.

- Again, the difference in success rate between the recombination operators was not very significant.
- Again an asexual EA with optimal $\mu = 1$ proved to outperform the EA with recombination.
- Again for the EA with recombination, Incest Prevention was able to improve the performance (though being costly).
- A difference with Graph 3-Coloring was that for SAT, extinctive selection with optimal λ performed better than preservative selection at a price of being less robust.

- Again the SAW mechanism proved to be robust with respect to its parameters, $\triangle w$ and T_p, and was able to improve the performance.
- The asexual EA was able to improve the best existing (incomplete) SAT algorithm, WGSAT, on the hardest instances.
- Transforming NP-complete problems to suit a canonical problem solver does not seem practical, unless to create difficult problems.

Chapter 8

Conclusions

saying: *If the only tool you have is a hammer, all problems tend to resemble a nail – anonymous.*
comment: *If all problems tend to resemble a nail, the only tool you need is a hammer (though a saw might be handy).*

In this chapter a summary of the most important results, the main conclusions and ideas for further research will be presented.

8.1 Summary of Results

- A comparison between algorithms is greatly affected by the hardness of the problem instances and hard instances at the phase transition should be used.
- For integer representation, the performance of an EA with recombination can be improved by Incest Prevention.
- A minimal population size exists, above which it is effective to have a big population and under which it is useless to have a population of more than one individual.
- For both considered problems, it was shown that an asexual EA with optimal population size $\mu = 1$ outperforms the more traditional EA which uses recombination.
- For Graph 3-Coloring, order-based representation outperforms integer representation.
- Fine tuned extinctive selection can be better than preservative selection, depending on the problem, but is less robust with respect to its parameters.
- For Graph 3-Coloring, initializing the order-based EA with (partial) colorings found by DSatur, can improve the performance of both the EA and DSatur. As this is similar to backtracking, for harder instances, the EA performs better than backtracking.

- Dynamically changing the mutation rate can improve the performance for the asexual EA with $\mu = 1$.
- A general, robust mechanism was designed that stepwisely adapts the weights (SAW) of the fitness function during the run and that was shown to be very effective for both considered problems.
- For Graph 3-Coloring, the best EA with SAW was shown to perform better on the hardest graph instances than one of the best existing algorithms, DSatur.
- For 3-Satisfiability, the best EA with SAW was shown to outperform the best existing algorithm, WGSAT, on the hardest SAT instances.
- Transforming NP-complete problems to solve them with a canonical problem solver is practically not very interesting and therefore makes EAs more interesting.

8.2 Main Conclusions

The goal of this research was testing EAs on an NP-complete problem, comparing them with good existing algorithms and possibly improving the EA performance when necessary.

It turned out that the more traditional EA with recombination and integer representation, was inferior to the existing method for Graph 3-Coloring and even the asexual GA, which is advocated by more and more researchers, that outperformed the EA with recombination, was inferior to DSatur. A change to order-based representation made the asexual EA come very close to the performance of DSatur for the hardest problem instances and big problem sizes.

As the optimal population size for the asexual EA turned out to be one individual, a population does not really exist anymore. Also, as we already lost the crossing over of individuals, we are only left with mutation and selection. Now the question comes up whether we should still call the algorithm an EA or maybe a local, random hill-climber. Really the name should not matter and realizing this "naming problem" is the most important. This because it makes it possible to draw ideas from both paradigms (EAs and hillclimbers in AI) to try improving the performance. An advantage of the asexual EA is that it performs much faster than EAs with recombination when a fitness function implementation is used that only locally recomputes the fitness. Also the complexity of the EA decreases significantly as no population is used anymore, no recombination is performed and no tournament selection is necessary anymore. Therefore the resulting algorithm is much easier to analyze theoretically which is a great advantage.

So with an asexual EA, we have increased performance, reduced real search time and reduced analytical complexity.

Further, the EA was greatly improved by the SAW mechanism which made the EA outperform DSatur on the hardest graph instances. So for harder and bigger problem instances the DSatur heuristics do not seem powerful enough and the EA should be preferred above a backtracking mechanism. For the easier and smaller graph instances, the DSatur heuristics outperform the EA and so in practice a combination could be advocated that first uses DSatur to try to color the instance very fastly and after failing, the EA can be used for a longer period of time.

An important property of EAs is that they are supposed to be a general robust search method and in this spirit, the SAW mechanism combines very well with the EA as it is very general and very robust, which was also verified on the Satisfiability problem. An important question, when working with NP-complete algorithms, is whether we can find a fast scalable algorithm. As we do not expect an efficient complete algorithm to exist, incomplete algorithms as the EA, become interesting and it was shown that the EA scales up quite well with bigger problem sizes.

Concluding this, we can say that a very effective, general and robust search algorithm has been presented that outperforms the good existing algorithms for two NP-complete problems for the hardest instances. The initial question about the power of EAs needs to be refined, as the best asexual EA only represents a part within the EA framework and further research has to tell us whether the other part within the EA-framework will survive. So in the big space of possible EAs, we have to continue the search for good algorithms. Of course many local optima pass our way, probably more global optima exist, many evaluations will be needed and above all for this problem we definitely need a big population hoping that good problem solvers will evolve in the heads of many researchers.

8.3 Further Research

Still further research will have to show whether the two main conclusions of this research, the superior performance of the asexual EA with population size 1 and the success of the SAW mechanism, can be confirmed for other CSP problems. Also it will be interesting to see if these two conclusions can be claimed for other types of problems, likes Constrained Optimization Problems (COPs). COPs have a function that, under certain constraints, has to be optimized. For the SAW mechanism this can be problematic as it will disturb the outcome of the function. For CSPs this is not a problem as were not interested in partial solutions and the optimal solution will have the same fitness, with or without SAW. As the results of the EA were very good for Graph 3-Coloring, it would be interesting to see how it performs on the Graph k-Coloring problem, where the minimal number of colors, k, to color a graph is searched.

For the asexual EA with population size 1, more research will have to show if it's superior to EAs with recombination. More and more often, researchers report that an asexual EA can perform better [29]. If this happens to be more general, it could be interesting to experiment with many other types of mutation, possibly making use of the fact that the positions in the beginning of a permutation are more important in order-based EAs. For example, we could directly place nodes that cannot be colored in front of the permutation, instead of doing this indirectly by increasing their weights.

For SAW many variations are possible. For example, instead of the best individual in a population, a part of the population could be used to get information about the hardness of the variables. A number of individuals could be selected with tournament selection, or the best individuals from the population could be used. Also instead of only increasing the weights of variables, weights could also be decreased after a while when the variable is correctly instantiated. Among other things, Frank experimented with this in [31], though the results on this were not very significant. Also questions remain about the weights that the SAW mechanism learned. Preliminary tests showed that using the learned weights again in a next run for the same instance, give spectacular degraded performances. Frank also found this and suggested using this negative information in next runs.

As dynamically changing the mutation rate proved effective for the asexual EA, it could be interesting to try a combination with the SAW mechanism, perhaps by increasing the mutation rate directly after each weight update and then slowly decrease it again. Interesting now becomes the self-adaptation by Bäck [7] as this works better for dynamically changing fitness functions which is the case for an EA that uses the SAW mechanism.

Also for Graph 3-Coloring, initializing the EA with recombination and a population with permutations found by DSatur (without backtracking), proved to be very successful. This is especially interesting as we can first use DSatur to filter out easy Graph 3-Coloring instances and then we could use the EA to continue with the unsuccessful DSatur coloring. Possibly this can be done for the asexual EA with population size 1 as well (though first results with SWAP were not very positive). Also, initializing the EA with integer representation with the (partial) colorings found by the Greedy Algorithm for random permutations can be interesting. This as the Greedy Algorithm gives more variation in its output, when given different random colorings, than DSatur. In this way, a wider part of the search space might be covered. For most problems, a kind of greedy algorithm can be defined and for an initializing purpose it does not have to be complete in the sense that a permutation leading to the correct solution always exist. So this method should be generally applicable.

Lastly, as was shown that two paradigms overlapped each other for Satisfiability in the case of GSAT and the asexual EA with population size 1, ideas from both fields can be used for improving the performance possibly even further. For example heuristics like arc or path consistency could prove useful. Very interesting will be a combination with TABU search, by Glover [38][39], which is an improvement on hill-climbing procedures. TABU search maintains a history of moves to prevent it from making the same moves again and is claimed to give very good results. Also Glover's Scatter Search with surrogate constraints [55], which keeps a population of potential solutions and generates trial points (offspring) by weighted linear combinations of them, could be interesting.

Appendix A

Summary of Abbreviations and Notation

Symbol	**Meaning**	**Pagenumber**
$\triangle w$	Weight increment constant for SAW	74
λ	Offspring population size	9
μ	Population size	9
l	Number of clauses for 3-Satisfiability	93
n	Problem instance size	31
p	Edge connectivity for Graph 3-Coloring	31
p_m	Mutation rate	18
p_c	Crossover rate	12
s	Number of seed for random generator	31
L	Number of genes in an individual	11
T_{max}	Maximum number of search steps per run.	22
T_p	Period for updating weights with SAW	74
AES	Average number of Evaluations to find a Solution	27
AFLE	Average Fitness of best individual in Last Evaluation	29
ATE	Average Time per Evaluation in 1000 evaluations/second	27
EA	evolutionary algorithm	3
EFE_α	Expected number of Function Evaluation to find a solution with certainty α	28
ET_α	Expected Time to find a solution with certainty α: $\text{EFE}_\alpha \cdot$ ATE	29
SAW	Stepwise Adaptation of Weights	74
SES	Standard Deviation of AES	29
SFLE	Standard deviation of AFLE	29
SR	Success Rate	26

Bibliography

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings 33rd IEEE Symposium on the Foundations of Computer Science*, pages 14–23, Los Angeles, CA, 1992. IEEE Computer Sociecty.

[2] T. Bäck. The interaction of mutation rate, selection, and self-adaption within a genetic algorithm. In R. Männer and B. Manderick, editors, *Parellel Problem Solving from Nature - 2*, pages 85–94, Amsterdam, 1992. Elsevier.

[3] T. Bäck. Self-adaptation in genetic algorithms. In *Proceedings of the first European Conference on Artificial Life*, pages 263–271, Paris, December 1992. MIT Press.

[4] T. Bäck and S. Khuri. An evolutionary heuristic for the maximum independent set problem. In *Proceedings of the first IEEE Conference on Evolutionary Computation*, pages 531–535. IEEE Press, 1994.

[5] Thomas Bäck. Optimal mutation rates in genetic search. In S. Forrest, editor, *Proceedings of the fifth International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1993.

[6] Thomas Bäck. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Proceedings of the first IEEE Conference on Evolutionary Computation*, pages 57–62. IEEE Press, 1994.

[7] Thomas Bäck and Martin Schütz. Intelligent mutation rate control in canonical genetic algorithms. Technical report, Center for Applied Systems Analysis, Joseph-von-Fraunhofer-Str. 20, D-44227 Dortmund, 1996.

[8] A. Blum. An $O(n^{0.4})$-approximation algorithm for 3-coloring (and improved approximation algorithms for k-coloring). In *Proceedings of the 21st ACM Symposium on Theory of Computing*, pages 535–542, New York, 1989. ACM.

[9] D. Brélaz. New methods to color vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[10] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pages 98–105. ACM Press, 1982.

[11] Scott H. Clearwater and Tad Hogg. Problem structure heuristics and scaling behavior for genetic algorithms. *Artificial Intelligence*, 81:327–347, 1996.

[12] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[13] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1993.

[14] Joseph C. Culberson and Feng Luo. Exploring the k-colorable landscape with iterated greedy. In *Second DIMACS Challenge*, Discrete Mathematics and Theoretical Computer Science. AMS, 1995. Available http://web.cs.ualberta.ca/~joe/.

[15] Charles Darwin. *The Origin of Species.* 1859.

[16] David Bull David Beasley and Ralph Martin. An overview of genetic algorithms: Part 2, research topics. *University Computing*, 15(4):170–181, 1993.

[17] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[18] M. Davis and H. Putnam. A computing procedure for quantification theory. *Jnl. Association for Computing Machinery*, 7:201–215, 1960.

[19] Richard Dawkins. *The Blind Watchmaker.* Penguin Books, 1986.

[20] A.E. Eiben, E.H.L. Aarts, and K.M. van Hee. Global convergence of genetic algorithms: A markov chain analysis. In R. Männer and B. Manderick, editors, *Proceedings of the 2nd Parallel Problem Solving from Nature*, pages 4–12. North-Holland, 1992.

[21] A.E. Eiben, C.H.M. van Kemenade, and J.N. Kok. Orgy in the computer: Multi-parent reproduction in genetic algorithms,. In F. Moran, A. Moreno, J.J. Merelo, and P. Chacon, editors, *Proceedings of the 3rd European Conference on Artificial Life*, number 929 in LNAI, pages 934–945. Springer-Verlag, 1995.

[22] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Genetic algorithms with multi-parent recombination. In Y. Davidor, editor, *Parallel Problem Solving from Nature - 3, LNCS 866*, pages 78–87, 1994.

[23] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Repairing, adding constraints and learning as a means of improving GA performance on CSPs. In M. Meyer, editor, *Benelearn '94 Proceedings of the 4th Belgian-Dutch Conference on Machine Learning*, pages 112–123, 1994.

[24] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In *First IEEE conference on Evolutionary Computation*, pages 542–547, 1994.

[25] A.E. Eiben and Zs. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions,. In *Proceedings of the 3rd IEEE World Conference on Evolutionary Computation*, pages 258–261. IEEE Service Center, 1996.

[26] Larry J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In *Foundations of Genetic Algorithms - 1*, pages 265–283, 1991.

[27] Charles Fleurent and Jacques A. Ferland. Genetic and hybrid algorithms for graph coloring. In I. H. Osman G. Laporte and P. L. Hammer, editors, *Annals of Operations Research*, Metaheuristics in Combinatorial Optimization, Université de Montréal, Département d'Informatique et de Recherche Opérationalle, Canada, 1994. Available via ftp://ftp.iro.umontreal.ca/pub/optim/fleurent/papers/coloring/coloring.ps.Z.

[28] T.C. Fogarty. Varying the probability of mutation in the genetic algorithm. In *Proceedings of the third International Conference on Genetic Algorithms*, pages 104–109, 1989.

[29] D.B. Fogel. *Evolutionary Programming.* IEEE Press, 1995.

[30] B.R. Fox and M.B. McMahon. Genetic operators for sequencing problems. In *Foundations of Genetic Algorithms - 1*, pages 284–300, 1991.

[31] Jeremy Frank. Learning short-term weights for GSAT. In *Proceedings of the AAAI*, 1996. To appear, available by http://rainier.cs.ucdavis.edu/~frank/decay.ml96.ps.

[32] Jeremy Frank. Weighting for godot: Learning heuristics for GSAT. In *Proceedings of the AAAI*, 1996. To appear, available by http://rainier.cs.ucdavis.edu/~frank/weighting.aaai96.ps.

[33] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freedman and Co., 1979.

[34] M.R. Garey, D.S. Johnson, and H.C. So. An application of graph coloring to printed circuit testing. *IEEE Trans. on Circuits and Systems*, CAS-23:591–599, 1976.

[35] I. Gent and T. Walsh. The enigma of SAT hill-climbing procedures. Technical Report 605, Department of AI, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland, 1992.

[36] I. Gent and T. Walsh. Unsatisfied variables in local search. In J. Hallam, editor, *Hybrid Problems, Hybrid Solutions.* IOS Press, 1995.

[37] Ian P. Gent and Toby Walsh. Towards an understanding of hill-climbing prodedures for SAT. *AAAI*, pages 28–33, 1993.

[38] F. Glover. Tabu search - part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[39] F. Glover. Tabu search - part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[40] David E. Goldberg. Sizing populations for serial and parallel genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the third International Conference on Genetic Algorithms*, pages 70–79. Morgan Kaufmann, 1989.

[41] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms - 1*, pages 69–93, 1991.

[42] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[43] G.R. Grimmet and C.J.H. McDiarmid. On colouring random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 77:313–324, 1975.

[44] Jin-Kao Hao. A clausal genetic representation and its evolutionary procedures for-satisfiability problems. In *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, France, April 1995.

[45] J. Hesser and R. Männer. Towards an optimal mutation probability in genetic algorithms. In Hans-Paul Schwefel, editor, *Parallel Problem Solving from Nature - 1, LNCS 496*, pages 23–32, 1991.

[46] J.H. Holland. *Adaption in natural and artificial systems.* MIT Press, 1975.

[47] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[48] K.A. De Jong and W.M. Spears. Using genetic algorithms to solve NP-complete problems. In *Proceedings of the third International Conference on Genetic Algorithms*, pages 124–132, 1989.

[49] H.A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th ECAI*, pages 359–363, 1992.

[50] C.H.M. van Kemenade, J.N. Kok, and A.E. Eiben. Raising GA performance by simultaneous tuning of selective pressure and recombination disruptiveness. Technical Report CS-R9558, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, August 1995.

[51] S. Khuri and T. Bäck. An evolutionary heuristic for the minimum vertex cover problem. In J. Kunze and H. Stoyan, editors, *KI-94 Workshops (Extended Abstracts)*, pages 83–84, Bonn, 1994.

[52] S. Khuri, T. Bäck, and J. Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In J. Urban E. Deaton, D. Oppenheim and H. Berghel, editors, *Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 188–193, New York, 1994. ACM Press.

[53] J.R. Koza. *Genetic Programming.* MIT Press, 1992.

[54] L. Kučera. The greedy coloring is a bad probabilistic algorithm. *Journal of Algorithms*, 12:674–684, 1991.

[55] A. Lokketangen and F. Glover. Surrogate constraint methods with simple learning for satisfiability problems. In D.-Z. Du, J. Gu, and P. Pardolos, editors, *Proceedings of the DIMACS workshop on Satisfiability Problems: Theory and Applications*, 1996.

[56] Bernard Manderick and Piet Spiessens. How to select genetic operators for combinatorial optimization problems by analyzing their fitness landscape. In J.C. Bioche and X.-H. Tan, editors, *Proceedings of the 7th Dutch Conference on Artificial Intelligence*, pages 127–136, 1995.

[57] Z. Michalewicz. *Genetic Algorithms + Data structures = Evolution programs.* Springer-Verlag, second edition, 1994.

[58] D. Mitchell, B. Selman, and H.J. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the AAAI*, pages 459–465, San Jose, CA, 1992.

[59] H. Mühlenbein. How genetic algorithms really work: I. mutation and hillclimbing. In R. Männer and B. Manderick, editors, *Parellel Problem Solving from Nature - 2*, pages 15–25, Amsterdam, 1992. Elsevier.

[60] B. Nudel. Consistent-labeling problems and their algorithms: Expected complexities and theory based heuristics. *Journal of Artificial Intelligence*, 21:135–178, 1983.

[61] I.M. Oliver, D.J. Smith, and J.C.R. Holland. A study of permutation crossover operators on the travelling salesman problem. In *Proceedings of the second International Conference on Genetic Algorithms*, pages 224–230, 1987.

[62] Anne L. Olsen. Penalty functions and the knapsack problem. In *Proceedings of the first IEEE Conference on Evolutionary Computation*, pages 554–558, 1994.

[63] B. Kenefsky P. Cheeseman and W. M. Taylor. Where the really hard problems are. In *Proceedings of the International Joint Conference on Artifical Intelligence*, pages 331–337, 1991.

[64] R. Reiter and A. Mackworth. A logical framework for depiction and image interpretation. *Artificial Intelligence*, 41(3):123–155, 1989.

[65] Jon T. Richardson, Mark R. Palmer, Gunar Liepins, and Mike Hilliard. Some guidelines for genetic algorithms with penalty functions. In J.D. Schaffer, editor, *Proceedings of the third International Conference on Genetic Algorithms*, pages 191–197. Morgan Kaufmann, 1989.

[66] J.D. Schaffer, R.A. Caruna, L.J. Eshelman, and R. Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In J.D. Schaffer, editor, *Proceedings of the third International Conference on Genetic Algorithms*, pages 51–60. Morgan Kaufmann, 1989.

[67] H.-P. Schwefel. *Evolution and Optimum Seeking.* Sixth-Generation Computer Technology Series. Wiley, New York, 1995.

[68] Bart Selman and Henry Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of IJCAI*, 1993.

[69] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

[70] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whiley. A comparison of genetic sequencing operators. In R.K. Belew and L.B. Booker, editors, *Proceedings of the fourth International Conference on Genetic Algorithms*, pages 69–76. Morgan Kaufmann, 1991.

[71] G. Syswerda. Uniform crossover in genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the third Internation Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1989.

[72] G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 21, pages 332–349. Van Nostrand Reinhold, New York, 1990.

[73] Gilbert Syswerda. A study of reproduction in generational and steady-state genetic algorithms. In *Foundations of Genetic Algorithms - 1*, pages 94–101, 1992.

[74] E. Tsang and T. Warwick. Applying genetic algorithms to constraint satisfaction optimization problems. In L.C. Aiello, editor, *Proceedings of ECAI-90*, pages 649–654. Pitman Publishing, 1990.

[75] Jonathan S. Turner. Almost all k-colorable graphs are easy to color. *Journal of Algorithms*, 9:63–82, 1988.

[76] D. De Werra. An introduction to timetabling. *European Journal of Operations Research*, 19:151–162, 1985.

[77] Darrell Whitley. The GENITOR algorithm and selective pressure: Why rank-based allocation of reproductive trials is best. In J.D. Schaffer, editor, *Proceedings of the third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann, 1989.