Resume of Genetic Algorithms and Evolutionary Computing [H02D1A]

Wouter Schaekers

—

Master Ingenieurswetenschappen: Wiskundige Ingenieurstechnieken
Master Ingenieurswetenschappen: Biomedische Technologie
Master Artificial Intelligence
Master Statistics
Master Informatica
Master Ingenieurswetenschappen: Computerwetenschappen
Master Biomedical Engineering
Master Engineering: Computer Science

March 15, 2014

Table of Contents

0 Introduction

This resume has been composed on the basis of Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications, by Michael Affenzeller, Stephan Winkler, Stefan Wagner, and Andreas Beham and also a bunch of papers (see the table of contents for a full list).
Because books and courses only contain 5% of content these days, I was able to reduce the total amount of pages significantly. This resume has been prepared with the idea that this course can be completed successfully, solely on the basis of this summary.
If you encounter empty sections, this is not because I have forgotten it, but because these sections contain a lot of grumbling without content. This is only a summary of the course text, not the associated slides.
This summary is probably not flawless.
Any adjustments can be made on https://github.com/WouterSchaekers/GeneticAlgorithms-Resume.

The author is not willing to sign summaries.
Sending spam is forbidden. Stalking of the author is, only after permission, granted under exceptional conditions.
The author is not responsible for any consequences of using this summary. If you are offended by the content of this summary, please stop reading and retract yourself in your bunker.
It is forbidden to burn or eat the printed version of this resume.

No long-term use without mathematical advice.

All line segments reserved. Do not throw on the public road.

This resume is released under the beerware license. Donations on the following bitcoin address are really appreciated. Thanks.

"Everything should be made as simple as possible, but no simpler."
-
Albert Einstein

19Shd7w2UEUns1zJDDqWgxPRf9ARzo1iD9

1 Chapter 1 - Simulating Evolution: Basics about Genetic Algorithms

1.1 The Evolution of Evolutionary Computation

There are two approaches in computer science that copy evolutionary mechanisms: evolution strategies (ES) and genetic algorithms (GA). In contrast to GAs, where the main role of the mutation operator is simply to avoid stagnation, mutation is the primary operator of evolution strategies. Genetic programming (GP) is an extension of the genetic algoritm. While GAs are intended to find arrays of characters or numbers, the goal of a GP process is to search for computer programs.

1.2 The Basics of Genetic Algorithms

A genetic algorithm works the following way:

- An initial population is generated (randomly or heuristically).
- During each iteration, individuals are selected (according to fitness) to produce offspring.

For the production of offspring, two methods are being used:

- Crossover: Combine substrings of the parents.
- Mutation: A random modification.

Convergence is not guaranteed. The algorithm terminates if the maximum number of iterations is reached or by finding an acceptable solution.
See figure 1.1, page 4 for a short overview. (This is the canonical genetic algorithm (CGA), because it is the basis for theoretical GA research.)

1.3 Biological Terminology

It is obvious that there is a close connection between genetic algorithms and biological evolution.

- A gene can be understood as an ”encoder” of a characteristic, such as eye color. The different possibilities for a characteristic is called an ’allele’.
- The sum of all chromosomes is called the genome. A genotype is the particular set of genes contained in a genome.
- The fitness of an organism is typically defined as its probability to reproduce, or as a function of the number of offspring the organism has produced.

The term chromosome refers to a solution candidate.

1.4 Genetic Operators

1.4.1 Models for Parent Selection

According to the fitness, individuals are chosen for the production of offspring. There are many ways of accomplishing this:

- Proportional selection (the roulette wheel): Each individual receives a proportional chance of being chosen (fitness/total fitness). Method can easily be programmed by calculating the cumulative chance (which is 1 for the last individual) and then generate a random number between 0 and 1.
- Linear-rank selection: The best individual receives x times more chance of being chosen than the worst individual. x is pre-determined. This 'flattens out' the differences between the fitness values. Each individual has at least one representation (even if the fitness is $0.00\ldots1$).
- Tournament selection: The most common variant is k-tournament, where k individuals are picked. The fittest individual is then chosen.

1.4.2 Recombination (Crossover)

- Single point crossover: Crossover over 1 point. See figure 1.2, page 8 for a graphical illustration.
- Multiple point crossover: Crossover over n points. These points are divided uniformly among the chromosome. The higher the n, the more disruptive this process is (obviously).
- Uniform crossover: For each gene, randomly choose one of the parents.

1.4.3 Mutation

Mutation takes the form of replacing an allele with a randomly chosen value in the appropriate range with probability p_m (usually smaller than 0.1). Mutation must be used with caution. In some problems like routing, this can lead to illegal chromosomes. The mutation can also adapt during the process (eg, more local to the end).

1.4.4 Replacement Schemes

In this subsubsection, we will discuss the way a new offspring should become members of the next generation.

- Generational Replacement: The entire population is replaced.
- Elitism: The bst individual of the previous generation are retained for the next generation. This strategy is commonly applied and is called the 'golden cage model'. If mutation is applied to the elite, it is called 'weak elitism'. All methods to lower the amount of elitism, will usually cause less convergence.

- Delete-n-last: Replace the n weakest individuals.
- Delete-n: Replace n random individuals.
- Tournament Replacement: Competitions are run between sets of individuals from the last and the actual generation, with the winners becoming part of the new population.

1.5 Problem Representation

1.5.1 Binary Representation

Chromosomes can be in binary representation. An example of a binary representation is the Traveling Salesman Problem, where each city is represented as $log_2(n)$ bits. A chromosome has a length of $nlog_2(n)$. There is also an alternative representation where a matrix is used (1 if city i is visited after city j, 0 otherwise). Because the offspring may be illegal, repair strategies must be used. These can lead to offspring that has not much in comon with the parents, something that counteracts the general functioning of GAs.

1.5.2 Adjecency Representation

The adjecency representation of a city is the following. If a tour in the TSP leads from city i to city j, j is listed on position i. To produce offspring, one can use alternating edges crossover. This means that an edge from one parent is chosen from the first location. Then this new city is the new location. The city in this location from the other parent is chosen. This goes on until all cities are chosen. If a city is chosen twice, this means that there is a cycle. A random other city must then be chosen.
The advantage is the fact that the child is build up with edges from its parents. The disadvantage is the fact that the child does not inherit longer tour segments.

1.5.3 Path Representation

The n cities of a tour are put in order according to a list of length n. Despite of being quite popular, the obvious disadvantage of this representation is the fact that each configuration can be described is $2n$ different ways.

1.5.4 Other Representations for Combinatorial Optimization Problems

Other combinatorial optimization problems are logistics and production planning, the job scheduling problem, ...

1.5.5 Problem Representations for Real-Valued Encoding

When using real-valued encoding, the dimension of the chromosomes is equal to the dimension of the solution vectors.
Crossover concepts are divided into discrete and continuous recombination. The discrete crossover copies the exact allele value of the parent. The continuous crossover performs some

kind of averaging.
In real-valued encoding, long building blocks are less or not important, in comparison with typical discrete representations.

1.6 GA Theory: Schemata and Building Blocks

Holland has introduced a construct called schema. A schema is a string with fixed and variable symbols (wild cards/don't cares). For example, [0#11#01] can represent the following strings: [0011001], [0011101], [0111001] and [0111101].
The number of individuals of the population belonging to a particular schema H at time $t+1$ is related to the same number at the time t as:
$m(H, t+1) = m(H, t)\frac{f_H(t)}{\overline{f}(t)}$
$f_H(t)$ is the average fitness value of the string representing schema H at time t. $\overline{f}(t)$ is the average fitness value of the whole population at time t.
When the fitness of a particular schema remains above the average by a fixed amount $c\overline{f}(t)$ and we fill it in in the original formula, this gives us:
$m(H, t+1) = m(H, t)\frac{\overline{f}(t)+c\overline{f}(t)}{\overline{f}(t)} = m(H, t)(1+c) = m(H, 0)(1+c)^t$
This exponential increase is reduced by the effect of crossover. The reduction can be calculated by $p_c\frac{\delta(H)}{l-1}$, where p_c is the crossover rate, $\delta(H)$ is de length of schema H, and l the length of the string. The length of schema [###0#0101] is 5.
Schemata with above average fitness and short defining length are the so-called building blocks.
Mutations also have an influence. The total survival probability is $(1-p_m)^{o(H)} \simeq 1-o(H)p_m$ for $p_m << 1$, where p_m is the probability that a mutation happens in a particular bit. $o(H) = \delta(H)$, they just wanted to troll and confuse everyone.
To sum up:
$m(H, t+1) \geq m(H, t)\frac{f_H(t)}{\overline{f}(t)}[1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m]$
This means that a schemata will grow or decrease exponentially.
The drawback of the building block theory is given by the fact that the underlying GA (binary encoding, proportional selection, single-point crossover, strong mutation) is applicable only to very few problems as it requires more sophisticated problem representations and corresponding operators to tackle challenging real-world problems.

1.7 Parallel Genetic Algorithms

Parallel genetic algorithms can be classified as global parallelization or coarse-grained parallel GAs. The coarse-grained variant (island model) is the most popular.

1.7.1 Global Parallelization

In global parallelization, there is a master and many slaves. The master gives a part of the population to the slaves, while the slaves return the fitness value of the subset they have received.
This only works for populations that are panmictic, where all individuals are possible mating partners.

1.7.2 Coarse-Grained Parallel GAs

Here, the population is divided into multiple subpopulations (so called islands or demes) that evolve isolated from each other, exept for an occasionally 'migration'. See figure 1.4, page 19 for the ones who like pictures. The qualitative performance is influenced by the number and size of the demes and also by the amount of migrations.
Relatively isolated demes will converge to different regions of the solution-space, and migration and recombination will combine these.
The only example is the SASEGASA algorithm (see chapter 5).

1.7.3 Fine-Grained Parallel GAs

The basic idea behind this model is that the individuals are spread throughout the global population like molecules in a diffusion process. Diffusion models are also called cellular models. In the diffusion model, a processor is assigned to each individual and recombination is restricted to the local neighborhood of each individual. This diffusion model is a sort of fine-grained parallel GA.
This kind of approach can be compared with the coarse-grained method, but the demes are very small.
Coarse-grained and fine-grained models are often mixed.

1.7.4 Migration

Important parameters for migration are:

- The communication topology of the interconnections between the demes.
- Which individuals that will migrate and which ones that will be replaced.
- The migration rate.
- The migration frequency.

Synchronous migration (constant intervals) is usefull for parallel GAs, but has shown to be slow and inefficient in some cases. Asynchronous migration only happend after specific events.
Self-adaptive selection pressure steering can detect local premature convergence (in a certain deme). This can be very efficient for parallel implementations. The fact that selection pressure is adjusted self-adaptively, makes (parallel) GAs more independent in terms of migration parameters.

1.8 The Interplay of Genetic Operators

Because of crossover, a breadth search will happen.
Mutation will avoid stagnation, so it is a depth search. However, crossover can bring this new information to other places of the search space. Then it becomes breadth search again.
Migration in coarse-grained parallel GAs functions somehow like a meta-model of mutation and is therefore a depth operator. Migration causes an increase of genetic diversity in the

specific demes, but decreases the diversity over all islands. This is called *exploitation of the search space*.

2 Chapter 2 - Evolving Programs: Genetic Programming

2.1 Introduction: Main Ideas and Historical Background

Genetic Programming is a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform the given task; it is a domain-independent, biologically inspired method that is able to create computer programs from a high-level problem statement.

Similar to the GA, GP is an evolutionary algorithm inspired by biological evolution to find computer programs that perform a user-defined computational task. It is therefore a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform the given task; it is a domain-independent, biologically inspired method that is able to create computer programs from a high-level problem statement.

2.2 Chromosome Representation

2.2.1 Hierarchical Labeled Structure Trees

2.2.1.1 Basics

- All tree nodes are either functions or terminals.
- Terminals are evaluated directly, i.e., their return values can be calculated and returned immediately.
- All functions have child nodes which are evaluated before using the children's calculated return values as inputs for the parents' evaluation.
- The probably most convenient string representation is the prefix notation, also called Polish or Lukasiewicz notation: Function nodes are given before the child nodes' representations (optionally using parentheses). Evaluation is executed recursively, depth-first way, starting from the left; operators are thus placed to the left of their operands. In case of fixed aritites of the functions (i.e., if the numbers of function's inputs is fixed and known), no parantheses or brackets are needed.

This can be summarized as follows:

- Symbolic expressions can be defined using
  - a terminal set T
  - a function set F
- The following general recursive definition is applied:

  - Every $t \in T$ is a correct expression.
  - $f(e_1, \ldots, e_n)$ is a correct expression if $f \in F$, $arity(f) = n$ and $e_1, \ldots, e_n$ are correct expressions.
  - There are no other forms of correct expressions.

- In general, expressions in GP are not typed (closure property: any $f \in F$ can take any $g \in F$ as argument). Still, as we see in the discussion of genetic operators, this might be not true in certain cases depending on the function and terminal sets chosen.

In the following we give exemplary simple programs. We thereby give conventional as well as prefix (not exactly following LISP notation) textual notations:

- (a) $IF(Y > X\ OR\ Y < 4)\ THEN\ i := (i+1),\ ELSE\ i := 0$.
  Prefix notation: $IF(OR(>(Y,X), <(Y,4)), := (i, +(i,1)), := (i,0))$.

- (b) $\frac{X+5}{2Y}$. Prefix notation: $DIV(ADD(X,5), MULT(2,Y))$.

Graphical representations of the programs are given as FIGURE 2.1, page 30.

2.2.1.2 Evaluation

As already mentioned previously, the execution (evaluation) of GP chromosomes representing hierarchical computer programs as structure trees is done recursively, dept-first way, and starting from the left.
Internal states before execution: $X = 7, Y = 3, i = 2$.
Execution:
$IF(OR(>(Y,X), <(Y,4)), := (i, +(i,1)), := (i,0))$
$\Rightarrow IF(OR(>(3,7), <(Y,4)), := (i, +(i,1)), := (i,0))$
$\Rightarrow IF(OR(FALSE, <(Y,4)), := (i, +(i,1)), := (i,0))$
$\Rightarrow IF(OR(FALSE, <(3,4)), := (i, +(i,1)), := (i,0))$
$\Rightarrow IF(OR(FALSE, TRUE), := (i, +(i,1)), := (i,0))$
$\Rightarrow IF(TRUE, := (i, +(i,1)), := (i,0))$
$\Rightarrow IF(:= (i, +(i,1)))$
$\Rightarrow IF(:= (i, +(2,1)))$
$\Rightarrow IF(:= (i,3))$
Internal states after execution: $X = 7, Y = 3, i = 3$

2.2.1.3 Genetic Operations: Crossover and Mutation

As genetic programming is an extension to the genetic algorithm, GP also uses two main operatorsfor producing new solution candidates in the search space, namely crossover and mutation.

Single-point crossover can be simply performed by replacing a subtree of (a copy of) one of the parents by a subtree of the other parent; these subtrees are chosen at random. There can be a chance of creating invalid chromosomes. Mutation can be seen as an arbitrary modification introduced to prevent premature convergence by randomly sampling new points in the search space. In the case of genetic programming, mutation is applied by modifying a randomly chosen node of the respective structure tree:

- A subtree could be deleted or replaced by a randomly re-initialized subtree.
- A function node could for example change its function type or turn into a terminal node.

2.2.1.4 Advantages

- Most programming language compilers internally convert given programs into parse trees representing the underlying programs.
- As evaluation is executed recursively starting from the root node, a newly generated or manipulated program can be (re-)evaluated immediately without any intermediate transformation step.
- Structure trees allow the representation of programs whose size and shape change dynamically.

2.2.2 Automatically Defined Functions (ADF) and Modular Genetic Programming

The main idea of ADFs is that program code (which has been evolved during the GP process) is organized into useful groups (subroutines); this enables the parameterized reuse and hierarchical invocation of evolved code as functions that have not been taken from the original functions set F but are rather defined automatically. The (re-)use of subroutines (subprograms, procedures) is enabled in this way.

With ADFs, a GP chromosome program is split into a main program tree (which is called and executed from outside) and arbitrarily many separate trees representing ADFs. These separate functions can take arguments as well as be called by the main program or another ADF.

Different approaches realizing modular genetic programming which have gained popularity and are well known in the GP community are the genetic library and the adaptive representation through learning (ARL) algorithm. In both approaches, some parts of the evolved code are automatically extracted from programs (usually of those that show rather good fitness values). These extracted code fragments are then fixed and kept in the GP library, thus they are available for the evolving programs in the GP population.

2.2.3 Other Representations

There are GP systems that are not based on trees.

2.2.3.1 Linear Genetic Programming

Individuals are represented by linear chromosomes. These linear solutions represent lists of computer instructions which are executed linearly.

Linear GP chromosomes are more similar to those of conventional GAs; however, their size is usually not fixed.
Advantages:

- A chromosome in a *stack-based GP* represents exactly a stack-based program by storing the program instructions in a list and using a stack for executing the program.
- *Register-based* and *machine code GP* are essentially similar. In both cases data are stored in registers, and instructions read data from and write results back to these registers. Initially, a program's inputs are written to registers, and after executing the program the results are given in one or more registers. The main difference is:
  - Register-based GPs have to be compiled before execution.
  - Machine code GPs don't have to be compiled. This is therefore a lot faster.

2.2.3.2 Graphical Genetic Programming

Parallel Distributed Graphical Programming (PDGP) is a form of GP in which programs are represented as graphs representing functions and terminals as nodes. Links between those nodes define the flow of control and results. PDGP defines a fixed layout for the nodes whereas the connections between them and the referenced functions are evolved by the GP process. PDGP enables high degree of parallelism as well as an efficient and effective reuse of partial results.

2.3 Basic Steps of the GP-Based Problem Solving Process

2.3.1 Preparatory Steps

There are some things to be done regarding the execution of the GP algorithm:

- Parameters that control the GP run have to be set.
- A termination criterion has to be defined.
- A result designation method has to be defined. (Explained later in section 2.3.4.)

2.3.2 Initialization

There are two possibilities for creating random initial programs:

- Full method: Nodes at depth $d < D_{max}$ point to randomly chosen functions from function set F, and nodes at depth $d = D_{max}$ are randomly chosen terminals (from terminal set T).

- Grow method: Nodes at depth $d < D_{max}$ become either a function or a terminal (randomly chosen from $F \cup T$), and nodes at depth $d = D_{max}$ are again randomly chosen terminals (from T).

2.3.3 Breeding Populations of Programs

Populations cannot grow infinitely in most applications. New programs have to replace old ones.

- Generational replacement: The entire population is replaced.
- Steady state replacement: New individuals are produced continuously and old individuals are removed continuously.
- Selection of replaced programs: The individuals that have to be removed can either be unfit ones, old ones or random ones.

The next creation scheme applies: In GAs, crossover and mutation are used sequentially. In GP, crossover and mutation (or a simple copy action) are executed independently; each time a new offspring is to be created, one of these variants is chosen probabilistically.

2.3.4 Process Termination and Results Designation

The termination criteria of genetic algorithms are also applicable for geneteic programming. For example the number of generations, when a problem-specific success predicate is fulfilled or when the values of fitness for numerous successive best-of-generation individuals appear to have reached a plateau. See figure 2.10, page 42 for a full summary.

After terminating the algorithm, the best individual has to be chosen as the result. In some cases it is a good idea to let the program run against a validation set V, which was not used during the GP training phase.

2.4 Typical Applications of Genetic Programming

2.4.1 Automated Learning of Multiplexer Functions

The automated learning of functions requires the development of compositions of functions that can return correct values of functions after seeing only a relatively small number of specific examples; these training samples are combinations of values of the function associated with particular combinations of arguments.

An example is the multiplexer. k address bits and 2^k data bits. This makes 2^{k+2^k} combinations. The evaluation of a solution candidate is done by applying the formula to all possible input bit combinations and counting the number of correct output values.

Korza was able to show that GP is able to solve the 3-address multiplexer problem 100% correctly.

2.4.2 The Artificial Ant

The problem is to navigate an artificial ant on a grid consisting of 32x32 cells. The grid is toroidal so that if the ant moves off the edge of the grid, it reappears and continues on the opposite edge. On this grid, 'food' units are distributed; each time the ant enters a square containing food, the ant eats it. At the beginning of the ant's wanderings it starts at cell (0,0) facing in a particular direction; at each time step, the ant is able to move forward in the direction it is facing, to turn right, or to turn left. The goal is to find a program that is able to navigate the ant so that as many food items as possible are eaten in a certain number of time units.

- The operations Move, Left and Right are used as terminals in the GP process.
- IfFoodAhead is a function. If food is ahead, it executes the first child. Otherwise the second child.
- Prog2 and Prog3, are also functions. They take 2 or 3 arguments and execute them consecutively.

An example of the ant problem is the so-called "Santa Fe trail".

2.4.3 Symbolic Regression

Symbolic regression is the induction of mathematical expressions on data. The key feature of this technique is that the object of search is a symbolic description of a model, not just a set of coefficients in a pre-specified model.

The main goal of regression in general is to determine the relationship of a depdendent (target) variable t to a set of specified independent (input) variables x. What we want to get is a function f that uses x and a set of coefficients w such that

$$t = f(x, w) + \epsilon$$

The form of f is usually pre-defined in standard regression and ANNs. This is not true in symbolic regression. Instead low-level functions are used and combined to more complex formulas during the GP process.

The taks of GP in symbolic regression is to find a composition of the functions, input variables, and coefficients that minimizes the error of the function with repsect to the desired target values. One of the ways to measure the error is taking the mean squared value.

2.4.4 Other GP Applications

3 Chapter 3 - Problems and Success Factors

3.1 What Makes GAs and GP Unique among Intelligent Optimization Methods?

What makes GAs and GP unique compared to neighborhood-based search techniques is the crossover procedure which is able to assemble properties of solution candidates which may be located in very different regions of the search space. In this sense, the ultimate goal of any GA or GP is to assemble and combine the essential genetic information step by step. This information is initially scattered over many individuals and must be merged to single chromosomes by the final stage of the evolutionary search process. This is exactly the essential property that has the potential to make GAs and GP much more robust against premature stagnation in local optimal solutions than search algorithms working without crossover.

3.2 Stagnation and Premature Convergence

Unfortunately, also users of evolutionary algorithms using crossover frequently encounter a problem which is quite similar to the problem of stagnating in a local optimum. This is called premature convergence and occurs if the population of a GA reaches such a suboptimal state that the genetic solution manipulation operators (crossover and mutation) are no longer able to produce offspring that outperform their parents. In general, this happens mainly when the genetic information stored in the individuals of a population does not contain that genetic information which would be necessary to further improve solution quality.

Classical Measures for Diversity Maintenance
The most common techniques for this purpose are based upon pre-selection, crowding, or fitness-sharing. The main idea of these techniques is to maintain genetic diversity by the preferred replacement of similar individuals or by the fitness-sharing of individuals which are located in densely populated regions.

Limitations of Diversity Maintenance
In basic GA literature the topic of premature convergence is considered to be closely related to the loss of genetic variation in the entire population. In natural evolution the maintenance of genetic diversity is of major importance as a rich gene pool enables a certain species to adapt to changing environmental conditions. In the case of artificial evolution, the environmental conditions, for which the chromosomes are to be optimized, are represented in the fitness function which usually remains unchanged during the run of an algorithm. Therefore, we do not identify the reasons for premature convergence in the loss of genetic variation in general but more specifically in the loss of what we call essential genetic information, i.e., in the loss of alleles which are part of a global optimal solution. Even more specifically, whereas the alleles of high quality solutions are desired to remain in the gene pool of the evolutionary process, alleles of poor solutions are desired to disappear from the active gene pool in order to strengthen the goal-directedness of evolutionary search.

Therefore, in the following we denote the genetic information of the global optimal solution (which is unknown to the algorithm) as essential genetic information. If parts of this essential genetic information are missing or get lost, premature convergence is already predetermined in a certain way as only mutation (or migration) is able to regain this genetic information.

Reflecting the basic concepts of GAs, the following questions and associated problems arise:

- Is crossover always able to fulfill the implicit assumption that two above-average parents can produce even better children?
- Which of the available crossover operators is best suited for a certain problem in a certain representation?
- Which of the resulting children are "good" recombinations of their parents chromosomes?
- What makes a child a "good" recombination?
- Which parts of the chromosomes of above-average parents are really worth being preserved?

4 Chapter 4 - Preservation of Relevant Building Blocks

4.1 What Can Extended Selection Concepts Do to Avoid Premature Convergence?

The ultimate goal of the extended algorithmic concepts described in this chapter is to support crossover-based evolutionary algorithms, i.e., evolutionary algorithms that are ideally designed to function as building-block assembling machines, in their intention to combine those parts of the chromosomes that define high quality solutions. In this context we concentrate on selection and replacement which are the parts of the algorithm that are independent of the problem representation and the according operators.

The unifying purpose of the enhanced selection and replacement strategies is to introduce selection after reproduction in a way that checks whether or not crossover and mutation were able to produce a new solution candidate that outperforms its own parents. Offspring selection realizes this by claiming that a certain ratio of the next generation has to consist of child solutions that were able to outperform their own parents. The RAPGA, the second newly introduced selection and replacement strategy, ideally works in such a way that new child solutions are added to the new population as long as it is possible to generate unique and successful offspring stemming from the gene pool of the last generation. Both strategies imply a self-adaptive regulation of the actual selection pressure that depends on how easy or difficult it is at present to achieve evolutionary progress. An upper limit for the selection pressure provides a good termination criterion for single population GAs as well as a trigger for migration in parallel GAs.

4.2 Offspring Selection (OS)

After selection of the parents, crossover and mutation, we introduce a further selection mechanism. A new parameter called success ratio ($SuccRatio \in [0, 1]$) is introduced. This is the quotient of the next population members that have to be generated by successful mating in relation to the total population size. A child is successful if its fitness is better than the fitness of its parents (the weak one, the strong one or weighted average?).

Accoring to simulated annealing, it is a weighted average with a comparison factor ($CompFactor$). 0 means the weak parent, 1 means the strong parent.

In the original formulation of the SASEGASA we have defined that in the beginning of the evolutionary process an offspring only has to surpass the fitness value of the worse parent in order to be considered as "successful"; as evolution proceeds, the fitness of an offspring has to be better than a fitness value continuously increasing between the fitness values of the weaker and the better parent. As in the case of simulated annealing, this strategy gives a broader search at the beginning, whereas at the end of the search process this operator acts in a more and more directed way. Once we filled up the claimed $SuccRatio$ of the next generation with successful individuals using the success criterion defned above, the rest of the next generation $(1 - SuccRatio)|POP|$ is simply filled up with individuals randomly chosen from the pool of individuals that were also created by crossover, but did not reach success criterion. The actual selection pressure $ActSelPress$ at the end of generation i is defined by the quotient of individuals that had to be considered until the success ratio was reached and the number of individuals in the population in the following way:

$$ActSelPress = \frac{|POP|Succ + \#children\ in\ POOL}{|POP|}$$

The upper limit of selection pressure ($MaxSelPress$) defines the maximum number of offspring considered for the next generation (as a multiple of the actual population size) that may be produced in order to fulfill the success ratio. This can also be used to detect premature convergence:

If it is no longer possible to find a sufficient number (SuccRatio $*$ $|POP|$) of offspring outperforming their own parents even if ($MaxSelPress * |POP|$) candidates have been generated, premature convergence has occured.

Higher success ratios cause higher selection pressures. Nevertheless, higher settings of success ratio, and therefore also higher selection pressures, do not necessarily cause premature convergence. The reason for this is mainly that the new selection step does not accept clones that are derived from two identical parents per definition. In conventionals GAs such clones represent a major reason for premature convergence of the whole population around a suboptimal value, whereas the new offspring selection works against this phenomenon.

4.3 The Relevant Alleles Preserving Genetic Algorithm (RAPGA)

Assuming generational replacement as the underlying replacement strategy, the most essential question at generation i is which parts of genetic information from generation i should be maintained in generation $i + 1$.

Offspring are accepted as members of the next generation if and only if they are able to outperform the fitness of their own parents and if they are new in the sense that their chromosome consists of a concrete allele alignment that is not represented yet in an individual of the next generation.

Aspects to consider:

- The algorithm should offer the possibility to use different settings also for conventional parent selection, so that the selection mechanisms for the two parents do not necessarily have to be the same. In many examples a combination of proportional selection and random selection has already shown a lot of potential. The two different selection operators are called male and female selection. It is also possible to disable parent selection totally (in the case of scalable selection pressure, after reproduction).
- It is reasonable to use more than one crossover and mutation operator. Increase in average selection pressure, the average running time and a broader search space, so retard premature convergence.
- An upper and lower limit of population size are necessary.
- Include a check for genotypical identiy. This means that only new idividuals are accepted in the sense that there may not be an identical copy of this individual in the population yet. The fitness value can also be used as a (possible dangerous) identity approximation check if a normal check is too time-consuming.

- If the maximally allowed population size cannot be filled up with new successful individuals, an upper limit of effort in terms of generated individuals is necessary. The maximum effort per generation is the maximum number of newly generated chromosomes per generation.

4.4 Consequences Arising out of Offspring Selection and RAPGA

Is crossover always able to fulfill the implicit assumption that two above-average parents can produce even better children?
Unfortunately this is not accomplished for a lot of operators in many theoretical as well as practical applications. A lot of offspring solution candidates do not meet certain constraints have to be 'repaired'. This leads to the fact that certain alleles are not present in the parents. Also, some operators do not support the evolvement of longer building blocks.

Which of the available crossover operators is best suited for a certain problem in a certain representation?
Maybe more disruptive operators perform quite well at the beginning of evolution whereas other crossover strategies succeed rather in the final phase of the algorithm. Also, running multiple crossover operators in parallel is a good idea.

Which of the resulting children are 'good' recombinations of their parents' chromosomes?
Outperform their own parents.

What makes a child a 'good' recombination?
This question is not really answered.
Why does the answer to question 3 make sense? Even parts of chromosomes with below average fitness may play an important role for the ongoing evolutionary process, if they can be combined beneficially with another parent chromosome which motivates gender specific parent selection.

Which parts of the chromosomes of parents of above-average fitness are really worth being preserved?
Ideally speaking, exaclty those parts of the chromosomes of above-average parents should be transferred to the next generation that make these individuals above average. What may sound like a tautology at the first view cannot be guaranteed for a lot of problem representations and corresponding operators.

4.5 Genetic algorithms, constraints, and the knapsack problem [Genetic Algorithms + Data Structures = Evolution Programs, by Z. Michalewicz]

A constrained problem is transformed to an unconstrained one by associating a penalty with all constraint violations; these penalties are included in the function of evaluation. A variety of possible penalty functions can be applied.

A version of penalty approach is elimination of non-feasible solutions from the population (death penalty). Such approach has its drawbacks, because for some problems the probability of generating a feasible solution is relatively small. In this approach non-feasible solutions do not contribute to the gene-pool of any population.

Special repair algorithms may be used, but might be computational intensive and difficult to run.

A third approach is the use of special representation mappings which guarantee the generation of a feasible solution or the use of problem-specific operators which preserve the feasibility of the solutions.

4.5.1 The 0/1 knapsack problem and the test data

A set of entities, together with their values and sizes, is given, and it is desired to select one or more disjoint subsets so that the of the sizes in each subset does not exceed given bounds and the total of the selected values is maximized. The task is, for a given set of weights $W[i]$, profits $P[i]$ and capacity C to find a binary vector $x = \langle x[1], \ldots, x[n] \rangle$ such that

$$\sum_{i=1}^{n} x[i] \cdot W[i] \leq C,$$

and for which

$$P(x) = \sum_{i=1}^{n} x[i] \cdot P[i]$$

is maximum.

Three random sets of data are considered:

- uncorrelated:
  $W[i]$:= (uniformly) random ([1 ... v]), and
  $P[i]$:= (uniformly) random ([1 ... v]).

- weakly correlated:
  $W[i]$:= (uniformly) random ([1 ... v]), and
  $P[i]$:= $W[i]$ + (uniformly) random ([-r ... r]).

- strongly correlated:
  $W[i]$:= (uniformly) random ([1 ... v]), and
  $P[i]$:= $W[i]$ + r.

Higher correlation problems have higher expected difficulty.

Data have been generated with the following parameter settings: $v = 10$ and $r = 5$ and $n = 100$, 250 and 500.

4.5.2 Description of the algorithms

Three types of algorithms were implemented and tested: penalty functions ($A_p[i]$), repair methods ($A_r[i]$) and decoders ($A_d[i]$).

4.5.2.1 $A_p[i]$

$$eval(x) = \sum_{i=1}^{n} x[i] \cdot P[i] - Pen(x)$$
$$Pen(x) = \rho \cdot (\sum_{i=1}^{n} x[i] \cdot W[i] - C)$$

is an example of the penalty function. In all three cases:

$$\rho = max_{i=1...n}\{P[i]/W[i]\}$$

4.5.2.2 $A_r[i]$

$$eval(x) = \sum_{i=1}^{n} x'[i] \cdot P[i]$$

where vector x' is a repaired version of the original vector x.

Some percentage of repaired chromosomes may replace the original chromosomes.

Two different repair algorithms are tested. A random selection and a greedy selection, which selects the item with the lowest profit to weight ratio.

4.5.2.3 $A_d[i]$

The vector $L = \langle 4, 3, 4, 1, 1, 1\rangle$ is decoded as the sequence $4, 3, 6, 1, 2, 5$. One-point crossover will produce a feasible offspring. The same can be said about the mutation operator.

Two different decoding algorithms are considered. Random decoding and greedy decoding, which orders the items on the basis of their profit to weight ratios.

4.5.3 Experiments and results

In all experiments the population size was constant and equal to 100. Also, probabilities of mutation and crossover were fixed at 0.05 and 0.65. The best solution found within 500 generations is the result.

The main conclusions:

- Penalty functions $A_p[i]$ do not produce feasible results on problems with restrictive knapsack capacity (C_1).
- $A_p[1]$ with C_2 and logarithmic penalty function is a clear winner.
- $A_r[2]$ is a winner for C_1.

Because in the case of restrictive knapsack capacity, only a very small fraction of possible subsets of items constitute feasible solutions, so most penalty functions will fail. On sparse problems, penalty functions seldom find solutions. This is because penalty functions do not make any distinction between infeasible solutions.

In the future, it might be interesting to experiment with additional formulae for the penalty function.

8 Chapter 8 - Combinatorial Optimization: Route Planning

8.1 The Traveling Salesman Problem

Given a finite number of cities along with the cost of travel between each pair of them, the goal is to find the cheapest way of visiting all the cities exactly once and returning to your starting point. Usually the travel costs are symmetric. A tour that passes through all the vertices (cities) is called a Hamiltonian cycle. The goal is to find the shortest Hameltonian cylce.

8.1.1 Problem Statement and Solution Methodology

8.1.1.1 Definition of the TSP

The aim is to find the optimal tour $s^* \in S$ such that $f(s^*) \leq f(s_k) \forall s_k \in S$.
$[d_{ij}]$ is the distance matrix. If there is no edge between i and j, then the distance is set to ∞. If $\pi_k(i)$ represents the city visited next after city i, the total distance function becomes $f(s_k) = \sum_{i=1}^{n} d_{i\pi_k(i)}$
TSP can, but don't have to satisfy the triangle inquality.
In an Euclidian TSP, it is mandatory to specify the coordinates of each node in order to calculate the distance:
$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

8.1.1.2 Versions of the TSP

Traveling Salesman Subtour Problems (TSSP)
Not all cities have to be visited. There is some kind of profit function in order to decide if the profit is higher than the travel expenses.
Postman Problems
Not a set of nodes, but a set of edges have to be visited (can be interpreted as streets). Also here, the goal is to minimize the total route length.
Time Dependent TSP
The cost is a combination of d_{ij} and the piont of time a certain city is visited.
Traveling Salesman Problem with Time Windows (TSPTW)
This is a normal TSP, but the route must start and end in a specific depot within a certain time window. So the cost is the total travel distance and/or the travel time, waiting time and service time.

8.1.1.3 Review of Optimal Algorithms

Total Enumeration
This has a complexity of $\mathcal{O}(n!)$. This is only applicable for very small problems.

Integer Programming
Introduction of an extra $n \times n$ matrix: $X = [x_{ij}]$, where x_{ij} indicates whether or not there is a connection from city i to city j.
Find $min(\sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij})$ such that:
$\sum_{j=1}^{n} x_{ij} = 1; \forall i \in \{1, \ldots, n\}$
$\sum_{i=1}^{n} x_{ij} = 1; \forall j \in \{1, \ldots, n\}$
$x_{ij} \geq 0; \forall i, j \in \{1, \ldots, n\}$
These constraints simply mean that each city has exaclty one successor and one predecessor. It is possible that two or more subcycles exist, which does not specify a valid TSP.
The solution space can be restricted by giving more constraints.

8.1.2 Review of Approximation Algorithms and Heuristics

8.1.2.1 Nearest Neighbor Heuristics

It considers a city as its starting point and takes the nearest city in order to build up the Hamiltonian cycle. See for figure 8.1, page 126 for the drawback of this heuristic.

8.1.2.2 Partitioning Heuristics

Partition the TSP, solve them and then recombine them. Particularly suitable for partitioning heuristics are only higher dimensional Euclidean TSPs with rather uniformly distributed cities.

8.1.2.3 Local Search

These are k-change methods that examine a k-tuple of edges of a given tour and test whether or not a replacement of the tour segments effects in an improvement of the actual solution quality.

8.1.2.4 The 2-Opt Method

2 edges are replaced (a 2-change operation). A tour s_i is adjacent to a tour s_j if and only if s_j can be derived from s_i by replacing two of s_i's edges. See figure 8.2, page 128 for an example.
Any route can be derived from any other route by at most $n - 2$ 2-change operations. Any solution has exactly $\frac{n(n-1)}{2}$ neighbors. For symmetrical TSPs this is $\frac{n(n-2)}{2}$. Do a 2-change until there is no improvement possible.
It is very unlikey that a 2-optimal tour is globally optimal.

8.1.2.5 The 3-Opt Method

Cfr.
$\frac{n(n-1)(n-2)}{2}$ operations are possible.
The probability to obtain a global optimal solution, is about $2^{-\frac{n}{10}}$.

8.1.2.6 The k-opt Method

Cfr.
A k-optimal solution is a global solution if $k = n$.

8.1.3 Multiple Traveling Salesman Problems

A set of salesman will try to do the job, each serving a subset of the cities involved. One city has to be the depot, representing the start and end point of all routes. There are two definitions for the MTSP:

- Bellmore's definition: Find exactly m tours.
- Second definition: Find at most m tours.

The depot has to be visited in all tours.
Notice that in the second definition, the solution will be a single tour (because of the triangle inequality). More constraints are needed.

8.1.4 Genetic Algorithm Approaches

8.1.4.1 Problem Representations and Operators

Adjacency Representation
A tour is represented as a list of n cities where j is listed in position i if and only if the tour leads from city i to city j. For example:
(7 6 8 5 3 4 2 1) means that city 7 comes right after city 1.
This representation is unique, but can represent an illegal tour. (2 1 4 3) contains two separate cylces.
The normal crossover can lead to illegal representations. Alternatives:

- Alternating Edge Crossover: Take the city from position one from the first parent. Then take the city from this new location from the second parent. Keep switching until all cities have been chosen. If a cycle appears, take a random unused city for this location.
  Example: (2 3 8 7 9 1 4 5 6) and (7 5 1 6 9 2 8 4 3) can give us (2 5 8 7 9 1 6 4 3).
  Good subtours are often disrupted by the crossover operator. Therefore, this operator doesn't perform good in practice.
- Subtour Chunks Crossover: Alternate between the parents, while selecting random subtours. Illegal tours are avoided by selecting a randomly chosen edge that does not lead to an illegal tour when this is necessary.
- Heuristic Crossover: The same as alternating edge crossover, but here the shortest next city is chosen.

The performance of the crossover operators are $3rd > 2nd > 1st$, but the performance of these crossover operators is overall quite bad.

Ordinal Representation

The best way to construct an ordinal representation is to start from the normal representation of a tour: $1-2-7-5-6-3-4$

and its indices: (1 2 3 4 5 6 7)

Then construct the ordinal representation by saying that the first city is also the first index, then remove 1 from the indices. 2 is again the first index (because we removed 1), ...

In this example we get: (1 1 5 3 3 1 1)

The main advantage of this method is the fact that classical crossover can be used because only cities can be chosen that have not been chosen before. Despite this, results using ordinal representaion have been generally poor (the partial tour to the right of the crossover point are split in a random way).

Path Representation

The path representation is exactly the same as a normal tour representation. Crossover operators:

- Partially Matched Crossover (PMX): The best way to explain this crossover operator is with an example. See figure 8.4, page 134. Two cutting points are selected. Then that substring from the first parent is replaced with the corresponding substring from the second parent. The elements from parent 1 that replace the elements of parent 2 in the child are being replaced by the elements that have been replaced.
  The PMX operator tries to keep the positions of the cities in the path representation which is useless, because we want to keep the sequences, not the positions. This crossover operator performs rather poor.

- Order Crossover (OX): The same applies here. See the example in figure 8.5, page 135 and you'll understand immediately. The elements of parent 1 that replace some elements in parent 2 are being eliminated in parent 2. All the elements move to the left or right until all empty spots are filled. The order of the cites is preserved, and not their position.

- Cyclic Crossover (CX): See the example in figure 8.6, page 136. Take the first element from the first parent which will replace the first element from the second parent. Now the first element from the second parent is removed, it has to be added again. Take this element on the position of the first parent and replace another element in the second parent in this position. This goes on until the first element, which was added to the second parent, is removed again.

- Edge Recombination Crossover (ERX): In the OX operator, there are still quite a lot of new edges in the offspring. Instead of recombining cities, this operator tries to recombine edges. The algorithm is rather easy. See Table 8.1, page 138 for an example. The algorithm goes as follows:
  - Take an initial city. This is also the first city in the new child.
  - Remove this city entirely from the edge map.

- Choose a random city that is connected to the current city that has the lowest amount of other connections. Add this city to the child and remove all of its occurences from the edge map.

This goes on until all cities are removed. If there are no more connections left, then a random city is chosen.
No mutation happens. All edges of the offspring occur in at least one of the parents.
An extension to this operator is the *enhanced edge recombination crossover (EERX)*, which gives priority to edges that appear in both parents.
The 2-change and 3-change mutation technique have been shown to be very effective in combination with this operator. Some important mutation operators:

- Exchange Mutation: Swap two cities.
- Insertion Mutation: Move a city to a random spot.
- Simple Inversion Mutation: Select two random cut points and reverse the string.
- Inversion Mutation: The same as simple inversion mutation, but then the string is moved to a random spot.

A Evolving Inventions

In the field of electronics, genetic programming has duplicated 15 previously patented inventions.

The first practical commercial area for genetic programming will probably be design. Design is especially well suited to genetic programming because it presents tough problems for which people seek solutions that are very good but not mathematically perfect. We frequently see creative things come out of the evolutionary process that would never occur to human designers.

A.1 Out of the Primordial Ooze

Example of the low-pass filter. Start embryonic. Apply progressive application of circuit-constructing functions. Other examples are antennas and (cruise/PID) controllers.

A.2 Evolvable Hardware

Rapidly reconfigurable field-programmable gate arrays. Each cell is programmable and also the interconnections are programmable by changing the configuration bits.

A.3 Run Times

Use a mass amount of low-grade computers to simulate semi-isolated subpopulations.

A.4 Passing an Intelligence Test

The patent office receives written descriptions of inventions and then judges whether they are unovbvious to a person having ordinary skill in the relevant field. Whenever an automated method duplicates a previously patented human-designed invention, the automated method has passed the patent office's intelligence test. The fact that the original, human-designed version satisfied the patent office's criteria means that the computer-created duplicate would also have satisfied the patent office.

B Using Genetic Algorithms For Supervised Concept Learning

A GA concept learner (GABL) is implemented that learns a concept from a set of positive and negative examples.

B.1 Introduction

The focus of this paper is to illustrate that GAs are more general and can be effectively applied to more traditional symbolic learning tasks as well.

B.2 Supervised Concept Learning Problems

A concept learning program is presented with both a description of the feature space and a set of correctly classified examples of the concepts, and is expected to generate a reasonably accurate description of the (unknown) concepts. An important issue is the choice of the concept description language. The language must have sufficient expressive power to describe large subsets succinctly and yet be able to capture irregularities. The two language forms generally used are decision trees and rules.

Another important issue arises from the problem that there is a large set of concept descriptions which are consistent with any particular finite set of examples. This is generally resolved by introducing either explicitly or implicitly a bias for certain kinds of descriptions (e.g. shorter or less complex descriptions may be preferred).

Finally, there is the difficult issue of evaluating and comparing the performance of concept learning algorithms. The most widely used approach is a batch mode in which the set of examples is divided into a training set and a test set. The validity of the description produced is then measured by the percentage of correct classifications made by the system on the second (test) set of examples with no further learning.

The alternative evaluation approach is an incremental mode in which the concept learner is required to produce a concept description from the examples seen so far and to use that description to classify the next incoming example. In this mode learning never stops.

B.3 Genetic Algorithms and Concept Learning

B.3.1 Representing the Search Space

The traditional internal representation used by GAs involves using fixed-length strings to represent points in the space to be searched. This representation works well for parameter optimization, but not for concept descriptions which are generally symbolic in nature.

One can either change the fundamental GA operators to work effectively with complex non-string objects or one can attempt to construct a string representation which minimizes any changes to the GAs. Both approaches are interesting. The second approach will be discussed now, the first approach at the end of this paper.

B.3.2 Defining Fixed-length Classifier Rules

A natural way to express complex concepts is as a disjunctive set of classification rules. The left-hand side of each rule consists of a conjunction of one or more tests involving feature values. The right-hand side of a rule indicates the classification to be assigned.

By restricting the complexity of the elements of the conjunctions, we are able to use a string representation and standard GAs, with the only negative side effect that more rules may be required to express the concept. This is achieved by restricting each element of a conjunction to be a test of the form:

Return true if the value of feature i of the example
is in the given value set, else return false.

For example:

if F1 = blue then it's a block
or
if (F2 = large) and (F5 = tall or thin) then it's a widget

With these restrictions we can now construct a fixed-length internal representation for classifier rules. Each fixed-length rule will have N feature tests, one for each feature. Each feature test will be represented by a fixed length binary string, the length of which will depend of the type of feature. E.g. F1 = 0111110 for the days of the week.

B.3.3 Evolving Sets of Classifier Rules

Two strategies: Michigan and Pittsburgh.
Michigan: A population of individual rules which compete with each other for space and priority in the population. Pittsburgh: Maintain a population of variable-length rule sets which compete with each other with respect to performance on the domain task. In this paper, the Pittsburgh approach will be evaluated. The number of rules in one individual is unrestricted.

Crossover can occur anywhere. The only requirement is that the corresponding crossover points on the two parents match up semantically.

B.3.4 Choosing a Payoff Function

$$payoff\ (individual\ i) = (percent\ correct)^2$$

B.3.5 The GA Concept Learner

Given the representation and payoff function described above, a standard GA can be used to evolve concept descriptions in several ways. The simplest approach involves using a batch mode in which a fixed set of examples is presented, and the GA must search the space of variable-length strings described above for a set of rules which achieves a score of 100%. We will call this approach GABL (GA Batch concept Learner).

The search terminates as soon as a 100% correct rule set is found within a user-specified upper bound on the number of generations. If a correct rule set is not found within the specified bounds or if the population loses diversity, the GA simply returns the best rule set found.

The simplest way to produce an incremental GA concept learner is to use GABL incrementally in the following way. The concept learner initially accepts a single example from a pool of examples. GABL is used to create a 100% correct rule set for this example. This rule set is used to predict the classification of the next example. If the prediction is incorrect, GABL is invoked to evolve a new rule set using the two examples. If the prediction is correct, the example is simply stored with the previous example and the rule set remains unchanged. As each new additional instance is accepted, a prediction is made, and the GA is re-run in batch if the prediction is incorrect. We refer to this mode of operation as batch-incremental and we refer to the GA batch-incremental concept learner as GABIL.

B.4 Empirical Studies

B.4.1 Evaluating Concept Learning Programs

We prefer the use of learning curves which measure the change in a system's performance over time in a possibly changing environment.

We are interested in examining how an incremental system changes its predicticve performance over time. Instead of computing the overall performance (batch mode), we examine a small window of recent outcomes.

B.4.2 Implementation Details

B.4.3 Initial Experiments

In addition to studying the behavior of our GA-based concept learner (GABIL) as a function of increasing complexity, we were also interested in comparing its performance with an existing algorithm. ID5R uses decision trees as the description language and always produces a decision tree consistent with the instances seen.

As the number of disjuncts and/or conjuncts increases, individual features become less informative, resulting in larger decision trees and poorer predictive performance. ID5R's information theoretic biases will therefore perform better on simpler target concepts.

GABIL, however, should perform uniformly well on target concepts of varying complexity. GABIL should not be affected by the number of conjucts, since with our fixed-length rule representation, large conjunctions are no more difficult to find than small ones. There is also no bias towards a small number of disjuncts. Given these biases (and the lack of them), then it is natural to expect that while ID5R will outperform GABIL on the simpler concepts, there will exist a frontier at which the situation will reverse.

Recall that each point on a curve represents the percent correct achieved over the previous 10 instances (and averages over 10 runs).

In ID5R, each negative example is represented by a unique leaf node in the decision tree. For this reason, ID5R cannot generalize over the negative examples, and has a good chance of predicting any negave example incorrectly. Even positive examples are not generalized well. It is clear that the decision tree representation is poor for representing this particular concept since the syntactic complexity of a terget concept corresponds roughly with the size of its decision tree representation.

B.5 Further Analysis and Comparisons

B.6 Conclusions

C A Knowledge-Intensive Genetic Algorithm for Supervised Learning

C.1 Introduction

Statistical models: The only representation is by means of all stored examples or some statistics on them. Connectionist models: The knowledge is distributed among network connections and an activation method.

C.2 Genetic algorithms

C.3 Inductive learning from examples

When the objects are described by features (attribute-value pairs) based on a number of multi-values attributes, the learning is said to be in an attribute-based space. A priori knowledge consists of a set of events that are examples of the space. When each such event is preclassified as belonging to a category, the learning is said to be supervised. The task is to generalize the a priori knowledge in order to produce descriptions of the concepts. When a rule-based framework is used to express the descriptions, the acquired knowledge is often called decision rules.

A decision tree can naturally produce complete and consistent partitions of the search space. On the other hand, it is more difficult for a set of rules to cover the search space in the same manner. Therefore, an extra mechanism is needed to account for possible cases of no-match and multiple-match when recognizing new events. Such problems can be avoided while learning single concepts if the system learns only the concept description and assumes that the subspace not covered by this description represents the complement of the concept.

One widely used language, which is closely associated with rules, is VL_1. Variables (attributes) are the basic units having multi-valued domains.

Selectors [*variable relation value*] can be used to express conditions on single attributes. Complexes can be used to express rules of the form *complex ::>decision.*

$$[\mathit{BloodPressure} = \mathit{High}][\mathit{Age} \neq \mathit{Young}] ::> \mathit{HeartRiskGroup}$$

There are two different approaches to learning from examples: full memory learning (as in our system) and only memory for the generated knowledge.

The system should possess incremental learning capabilities.

C.4 Previous approaches

C.4.1 Traditional approaches

Statistical approaches account for the vast majority of non-symbolic, or numerical, approaches. They usually operate in batch mode on the data set in order to obtain some statistical measures, which are later used as probabilistic approximations of appearances of different features.

Another numerical approach comes from the neural network community.

C.4.2 Genetec algorithm approaches

- **Michigan**
- **Pittsburgh**

C.5 The modified genetic algorithm

The actual start of the paper.

C.5.1 Ideas used

There is a whole spectrum of possible GA designs along the dimension of task-specific knowledge utilization. On one side of the spectrum lies a method that only uses the classical operators of mutation and crossover. On the other side of the spectrum lies a knowledge-intensive method that completely abandons the traditional domain-independent operators, and instead, fully implements the specific problem-solving methodology.

Even though the last approach does not have the same theoretical support, it is backed by the task-specific knowledge used to guide and conduct the search. This property should provide for a faster convergence to a desired solution. Moreover, it may be easily shown that all the operators we subsequently define and use are actually special cases of the traditional mutation and crossover. This provides and intuitive support for the same theoretical foundations.

While applications of the traditional domain-independent operators provide for a domain-independent search conducted in the artificial representation space, we set the genetic algorithm to operate directly in the problem space by organizing the work there.

C.5.2 Representation and search space

We adopt the multiple-valued logic language VL_1 as the choise for the chromosome's representation. Then the search space is the space of sets of rules, spanned by given features. This is the space of VL_1 concept descriptions. Because we do not employ any extra axioms, it is quite feasible and possible to have redundant descriptions, such as

$$[Age > Young] ::> HeartRiskGroup, \; [Age = Old] ::> HeartRiskGroup$$

For simplicity of presentation, from now on we only consider VL_1 formulas built using the sufficient '=' relation with internal disjunctions.

C.5.3 Initial population

We allow for three different types of chromosomes to fill the population initially:

- Random initialization. Each individual is a set of a random number of complexes, randomly generated on the search space.
- Initialization with data. Each individual is a random positive training event.
- Initialization with prior hypotheses, provided such are available.

The best avarage behavior is obtained while using a combination of these three.

C.5.4 Evaluation mechanism

In supervised learning from examples, the criteria include completeness, consistency and possibly complexity. One may wish to accommodate some additional criteria, such as cost of attributes, length of descriptions, their generality, ..., but we did not consider them in the current implementation.

$e^{+/-}$ is the number of positive/negative training events covered by a rule, $\epsilon^{+/-}$ is the number of such events covered by a rule set and $E^{+/-}$ is the total number of such events. See table 2, page 202 for the completeness and consistency formula for a rule set and a rule. Correctness can be calculated this way:

$$correctness = \frac{w_1 * completeness + w_2 * consistency}{w_1 + w_2}$$

and

$$evaluation = correctness * (1 + w_3 * (1 - cost))^f$$

where w_3 determines the influence of *cost* and f grows very slowly on $[0, 1]$ as the population ages. The cost is measured by the complexity: $2 * \#rules + \#conditions$

Initial experiments suggest that the system performs better when the f exponent somehow fluctuates, and that the final increase should start based upon an anticipated exhaustion of resources or when the currently learned description is already complete and consistent.

C.5.5 Operators

C.5.5.1 Rule set level

This is the level of sets of VL_1 complexes.

- Independent: It exchanges random rules between two parent rule sets.
- Generalization:
  - Rules copy. It copies a random rule from each of the sets to the other.
  - New event. If there is a positive event not covered yet by the current rule set, this event's description is added to the set as a new rule.
  - Rules generalization. It selects two random rules and replaces them by their most specific generalization.
- Specialization:
  - Rules drop. It drops a random rule from a rule set.
  - Rules specialization. It replaces two random rules by their most general specialization.

C.5.5.2 Rule level

This is the level of VL_1 complexes.

- Independent:
  - Rule split. Splits a rule into a number of rules.
- Generalization:
  - Condition drop. Removes a present condition from a rule.
  - Turning conjunction into disjunction. Splits the complex into a disjunction at a certain point.
- Specialization:
  - Condition introduce. Introduces a random condition.
  - Rule directed split. If a rule covers a negative event, it is split into a set of maximally general rules that are yet consistent with that event.

C.5.5.3 Condition level

This is the level of VL_1 selectors.

- Independent:
  - Reference change. It randomly adds or removes a single domain value to this condition.
- Generalization:
  - Reference extension. It extends the domain of a single condition by allowing a number of additional values.
- Specialization:
  - Reference restriction. It removes some domain values of a single condition.

C.5.5.4 Operator selection probability

Chinese.

C.5.6 Algorithm

The algorithm uses the above components and the control of genetic algorithms. At each iteration, all rule sets of the population are evaluated and a new population is formed by drawing members from the original one in such a way that more fit individuals have a higher chance of being selected. Following that, the operators are applied to population members in order to move these partial solutions, hopefully, closer to the desired state.

C.6 Some implementation issues

Genetic-Based Inductive Learning (GIL).

C.6.1 Sampling mechanism

We use the stochastic universal sampling mechanism.

C.6.2 Internal representation

For example, assuming that an attribute has five domain values, the binary vector 11001 represents the condition saying that the attribute must have the first, second or the last value.

A complex is represented by a vector of conditions. See the example on page 210.

C.6.3 Data compilation

Prior to learning all data are precompiled into vectors, like in table 3, page 211.

To compute the result of a 'rules copy' operator, only a bitwise OR has to by applied.

C.7 Experimental studies

C.7.1 Experimental methodology

Split the available events into training and testing groups (70/30).

The formula in section C.5.4 can be used for qualitative properties (comprehensibility).

C.7.2 Emerald's robot world

See a summary of the world on page 213.

C.7.2.1 The trace

...

C.7.2.2 Comparative experiments

GIL produced the highest recognition rate, especially when seeing only a small percentage of the events. This result can be attributed to the simplicity-biased evaluation formula that was used.

C.7.3 DNF concepts

GIL achieves very high performance for all kinds of problems. It outperforms GABIL in terms of learning variability.

C.7.4 Multiplexers

...

C.7.5 Breast cancer

...

C.7.6 Incremental learning

It's interesting to note that there was no significant difference in the quantitative performance, suggesting GIL can be used in incremental environments. Moreover, observed faster convergence in the incremental mode suggests that the reused knowledge in forms of the initial population was processed very usefully, indicating the system's ability to process initial hypotheses and maintain knowledge in a dynamic environment.

C.8 Conclusions and further research

C.8.1 Multiple domains

If we assume independence of different class descriptions, we may apply subsequent learning settings of the same algorithm, one session per class. Then, during a learning session for class n, the examples of category n are considered as positive events, while the examples of all other categories are considered as the current negative events. The same idea may be used differently: we may conduct the learning sessions simultaneously in different populations, with one population assigned to one class being learned.

C.8.2 Other issues

One of the major disadvantages of the current implementation is its high parameterization: there are about 40 input parameters that must be specified, with most of them being continuous probability values. This is the reason for the poorer performance noticed on more complex problems.

All these parameters could be replaced by few conceptual ones. Such an abstraction would provide for both easier use and more efficient performance. Some recent results show that adjusting the bias (non-probabilistic parameters) appropriately can improve both the speed and the quality of learning.

Our simplified approach assumes a crisp concept representation with rule-based conceptualization and is not well suited for dealing with noise.

D Evolving 3D Morphology and Behavior by Competition

D.1 Introduction

Gezever.

D.2 The Contest

See figure 1, page 29 for the set up. Creatures are wedged into these 'starting zones' until they contact both the ground plane and the diagonal plane, so taller creatures must start further back.

A creature gets a higher score by being closer to the cube but also gets a higher score when its opponent is further away. The fitness for each creature is given by

$$f_i = 1 + \frac{d_j - d_i}{d_i + d_j}$$

D.3 Approximating Competitive Environments

In the most extreme case, each individual competes with all the others in the population and the average score determines the fitness. This requires $(n-1)n/2$ competitions.

A compromise is for each individual to compete against several opponents chosen at random for each generation.

An other compromise is a tournament pattern (See figure 2c, page 30).

A third compromise is for each individual to compete once per generation, but all against the same opponent. This gives a fair relative fitness value since all are playing against the same opponent which has proven to be competent.

D.4 Creature Morphology

A phenotype hierarchy of parts is made from a graph by starting at a defined root-node and synthesizing parts from the node information while tracing through the connections of the graph. Nodes can connect to themselves or in cycles to form recursive or fractal like structures. See figure 3, page 31.

D.5 Creature behavior

A virtual 'brain' determines the behavior of a creature. See figure 4, page 32.

D.5.1 Sensors

Each sensor is contained within a specific part of the body. Three types of sensors were used:

- *Joint angle sensors* give the current value for each degreee of freedom of each joint.

- *Contact sensors* activate if a contact is made.
- *Photosensors* react to a global light source position. The source of one color is located in the desirable cube, and the other is located at the center of mass of the opponent.

D.5.2 Neurons

In this work, different neural nodes can perform diverse functions on their inputs to generate their output signals.

Some functions compute an output directly from their inputs, while others such as the oscillators retain some state and give time varying outputs even when their inputs are constant.

In this work, two brain time steps are performed for each dynamic simulation time step so signals can propagate through multiple neurons with less delay.

D.5.3 Effectors

Each effector simply contains a connection from a neuron or a sensor from which to receive a value. This input value is scaled by a constant weight, and then exerted as a joint force which affects the dynamic simulation and the resulting behavior of the creature. Only effectors that exert simulated muscle forces are used here.

D.5.4 Combining Morphology and Control

See figure 5, page 33 for an example of how morphology (the physical form of the creature) can be combined with the nueral circuit.

D.6 Physical Simulation

...

D.7 Creature Evolution

Seed genotypes are synthesized 'from scratch' by random generation of sets of nodes and connections.

Before creatures are paired off for competitions and fitness evaluation, some simple viability checks are performed, and inappropriate ceratures are removed from the population by giving them zero fitness values. Those that have more than a specified number of parts are removed.

The number of offspring that each surviving individual generates is proportional to its fitness.

D.7.1 Mutating Directed Graphs

A directed graph is mutated by the following sequence of steps:

1. Boolean values are flipped, scalar values are increased/decreased Gaussian-like or negated. Sometimes a random value is picked.

2. A new random node is added to the graph.

3. The parameters of each connection are subjected to possible mutations.

4. New random connections may be added and existing ones removed.

5. Unconnected elements are garbage collected.

D.7.2 Mating Directed Graphs

The first is a crossover operation. See figure 7a, page 35.

A second mating method grafts two genotypes together by connecting a node of one parent to a node of another parent. See figure 7b, page 35.

D.7.3 Parallel Implementation

This process has been implemented to run in parallel on a Connection Machine CM-5 in a master/slave message passing model.

D.8 Results and Discussion

See figure 8, page 36. Some species took many generations before they could even reach the cube at all, while others discovered a fairly successful strategy in the first 10 or 20 generations.

After the results from many simulations were observed, the best were collected and then played against each other in additional competitions. The different strategies were compared, and the behavior and adaptability of creatures were observed as they faced new types of opponents that were not encountered during their evolutions.

It is possible that adaptation of an evolutionary scale occurred more easily than the evolution of individuals that were themselves adaptive. Perhaps individuals with adaptive behavior would be significantly more rewarded if evolutions were performed with many species instead of just one or two. To be successful, a single individual would then need to defeat a larger number of different opposing strategies.

D.9 Future Work

...

D.10 Conclusion

...

E Optimization of Control Parameters for Genetic Algorithms

E.1 Introduction

The problem of tuning the primary algorithm represents a secondary, or metalevel, optimization problem. See figure 2, page 123 for a representation.

E.2 Overview of Genetic Algorithms

We desire to optimize a process having a response surface u, which depends on some input vector x.

E.3 Experimental Design

These experiments were designed to search the space of GA's defined by six control parameters, and to identify the optimal parameter settings with respect to two different performance measures. The search for the optimal GA's were performed by GA's, which demonstrates the efficiency and power of GA's as metalevel optimization techniques.

E.3.1 The Space of Genetic Algorithms

This study is limited to a particular subclass of GA's characterized by the following six parameters:

- *Population Size (N)*: Large populations are more likely to contain representatives from a large number of hyperplanes. As a result, a large population discourages premature convergence.
- *Crossover Rate (C)*
- *Mutation Rate (M)*
- *Generation Gap (G)*: The percentage of the population to be replaced durin each generation. $G = 1 - elites$
- *Scaling Window (W)*: The performance value $u(x)$ of a structure x is $u(x) = f(x) - f_{min}$ with f_{min} the minimum value that $f(x)$ can assume in the given search space. If $W = 0$, then f'_{min} is set to the minimum $f(x)$ in the first generation. Each structure that has an evaluation less than f'_{min} is ignored. f'_{min} is rescaled once all the structures in one generation are larger than this minimum. If $0 < W < 7$, then we set f'_{min} to the least value of $f(x)$ which occured in the last W generations. A value of $W = 7$ indicated an infinite window (no scaling).
- *Selection Strategy (S)*: First, pure selection (proportional) is used, then elitist selection (best performing structure always survives).

These six parameters defines a space of 2^{18} GAs, so each GA is defined by a 18-bit vector.

E.3.2 Task Environment

Each GA was evaluated by using it to perform five optimizaiton tasks, one for each of five carefully selected numerical test functions.

E.3.3 Performance Measures

Online performance of a search strategy s and a response surface e: $U_e(s,T) = ave_t(u_e(t))$, where $u_e(t)$ is the performance of the structure evaluated at time t. Online performance is the average performance of all tested structures over the course of the search.
Offline performance: $U_e^*(s,T) = ave_t(u_e^*(t))$ where $u_e^*(t)$ is the best performance achieved in the time interval $[0,t]$. Offline performance is preffered if it is possible (e.g. simulation), while the best structure so far is used to control the online system.

E.3.4 Experimental Procedures

Two experiments were performed, one to optimize online performance and one to optimize offline performance.

1. The metalevel GA started with a population of 50 randomly chosen GAs and used a standard parameter setting.
2. The GAs showing the best performances during step 1 are subjected to more extensive testing. Each of the 20 best GAs in step 1 was again run against the task environment, this time for five trails for each test function, using different random number seeds for each trail. The GA which exhibited the best performance in this step was declared the winner of the experiment.

E.4 Results

E.4.1 Experiment 1 - Online Performance

The first experiment was designed to search for the optimal GA with respect to online performance on the task environment. Figure 4, page 126 shows the average online performance for the 50 GAs in each of the 20 generations of experiment 1.

E.4.2 Experiment 2 - Offline Performance

The second experiment was designesd to search for the optimal GA with respect to offline performance on the task environment. Figure 5, page 126 shows the average offline performance of random search (100). This finding verifies the experience of many practitioners that GAs can prematurely converge to suboptimal solutions when given the wrong control parameters.

E.4.3 General Observations

Mutation rates above 0.05 are generally harmful with respect to online performance. The absence of mutation is also associated with poorer performance. Population size: 30-100, 60-110 for offline performance.

In smaller populations crossover plays an important role in preventing premature convergence.

E.5 Validation

To validate the experimental results, the three algorithms GA_S, GA_1 and GA_2 were applied to an optimization problem which was not included in the experimental task evironment.

E.6 Conclusion

. . .

F Successful Lecture Timetabling with Evolutionary Algorithms

F.1 Introduction

The most prominent overall constraint is that there should be no clashes.

Typically this is addressed by drawing up an initial draft timetable, followed by perhaps weeks of redrafting as complaints about the most recent draft flow in from various sources.

F.2 Lecture Timetabling Problems

A set of events $E = \{e_1, e_2, \ldots, e_v\}$, agents (lecturers) $A = \{a_1, a_2, \ldots, a_t\}$, places $P = \{p_1, p_2, \ldots, p_q\}$ and a set of times $T = \{t_1, t_2, \ldots, t_s\}$. Each event e_i has an associated length l_i and an associated size s_i, which is either known or an estimate of the number of students expected to attend that event. Tuples of (e, t, p, a) have to be found.

F.3 Evolutionary Timetabling

The timetable chromosome is a vector of symbols of total length $3v$. The three alleles in the ith chunk represent time (t), place (p) and agent (a).

F.3.1 The Fitness Function

$$f(g) = 1/(1 + \textstyle\sum_{i \in C} P_i v_i(g))$$

C is the set of constraints in the problem.
P_i is a penalty associated with constraint i.
$v_i(g) = 1$ if timetable g violates constraint i and 0 otherwise.

F.4 A Specific Lecture Timetabling Problem

T comprises 80 start times, 16 per day. Each day's slots are at half-hourly intervals from 9h to 16h30. E comprises a large collection of lectures, tutorials, and lab sessions, mostly an hour long, but sometimes two hours long. ...
Constraints:

- **Options**: No overlapping.
- **Event Spread**: Spreading during the week, also same lecture on different days.
- **Travel Time**: A student should have at least 30 minutes free for travel between events on different departments.
- **Slot Exclusions**: Lectures at lunch time should be avoided if possible.
- **Slot Specifications**: Specific constraint.

- **Capacity**: The capacity of the room may not be exceeded.
- **Room Exclusions**: Specific exclusions about the room.
- **Room Specifications**: Specific constraints about the room.
- **Juxtaposition**: All tutorials or laboratory sessions for any course should occur later in the week than the first lecture of that course.

In this case, we don't need to consider the set A.

F.4.1 Keeping Options Open

Make sure that there is no overlapping between courses for the same student. All other overlapping is allowed.

This amounts to a collection of binary constraints of the form "e_1 must not overlap in time with e_2.".

F.4.2 Event Spread contraints

Calculate the spread of every student or the whole timetable.

The number of instances of the following two offences:

- Four events are scheduled in one day.
- Five or more events are scheduled in one day.

A different penalty term is associated with each and the penalty weighted sum of instances of these offenses, summed over virtual students, makes up the contribution to fitness of the overall event spread constraint.

Also, a penalty is added when a pair of lectures of the same module is assigned on the same day.

F.4.3 Exclusions and Specifications

Pre-arrangement can be used.

Sometimes, a timetable may violate a single constraint, but make it up for this in being excellent overall.

F.4.4 Capacity Constraints

...

F.4.5 Juxtaposition Constraints

'lisp_1 must be before lisp_t3.'

F.5 Experiments

The EA used a population size of 50, uniform crossover, gene-by-gene mutation, and elitist generational reproduction. Each generation consisted of 50 evaluations. Each trail was run for 200 generations.

Ten trails were run for each experiment and results for each problem record for the best V (= vector of violations) found overall and the mean of V over the ten trails for each of the three selection schemes.

F.5.1 Results

The course organisers did a horrible job. TOUR did an awesome job with just a few or even no violations. > FIT > RANK.

F.6 Discussion

The results clearly indicate the benefits of using a penalty-function based on EA approach on this problem.

F.6.1 Scaling up

Hard to say. Solutions near optimal regions are rapidly found on large complex problems, but further evolution towards optima becomes considerably slow and may stop altogether. Fortunately, this difficulty is readily aided by the use of smart hillclimbing mutation operators.

F.6.2 Generalising Across

As long as the number of constraints and the computational ease of checking them make for a relatively speedy evaluation function, which is usually the case when using the direct chromosome representation.

F.6.3 Different Approaches

Different representations, use of domain specific recombination operators and hybridisation of the EA (evolutionary algorithms) with other techniques are all candidates for refinement.

F.6.4 Practice

...