Poznan University of Technology
Faculty of Computer Science and Management
Institute of Computer Science

Master's thesis

POTENTIAL FITNESS FOR GENETIC PROGRAMMING

Przemyslaw Polewski

Supervisor
dr hab. inż. Krzysztof Krawiec

Poznań, 2008

karta pracy dyplomowej;

.

Contents

Abstract

The goal of this thesis was to propose a novel approach to the estimation of an individual's fitness during the evolutionary process in tree-based Genetic Programming, a variant of evolutionary computation where solutions encode programs to be executed, for the case when desired output values of the individual are defined for all fitness cases. The area of application of this new approach is restricted to boolean functions. The standard fitness function used in GP usually calculates the individual's *current* fitness, that is, how well the output values produced by the individual conform to the provided sample outputs for every fitness case. As opposed to this, the method presented in this thesis, further referred to as *potential fitness*, attempts to estimate the fitness of the individual's direct offspring.

To estimate potential fitness, the effect of tree-replacing recombination operations on the individual's semantics (output values) is analyzed. The fitness is calculated based on solely the subset of fitness cases where the individual's output is invariant to recombination operations (such as crossover, mutation). The individual is rewarded for each such fitness case for which its output is consistent with training data, and penalized for fitness cases where its output value and the provided output value differ. Fitness cases for which the individual's output is not fixed (i.e. is affected by recombination operations) are not taken into account, as we cannot know in what way the recombination operators will alter the individual, and thus what values will be produced.

The potential fitness approach was implemented using the Java programming language, as an extension to the existing *Evolutionary Computations in Java (ECJ)* package. It was then tested on the following boolean benchmark problems:

- Odd Parity: 4-bit,5-bit,6-bit
- Multiplexer: 6-bit,11-bit
- Comparator: 4-bit,6-bit

The results were than compared to those obtained using standard Genetic Programming. All calculations were performed using the standard Koza-I settings.

Analysis of the results indicates that there is a statistically significant difference between the two sets of results, clearly showing that the Potential Fitness approach

is in most cases superior to, and in all cases not worse than standard Genetic Programming, in terms of both success rate and mean quality of best obtained individual per evolutionary run. However, this improvement in quality comes at the expense of longer running times. On average (across all the considered problems), the increase in running times is about 200% as compared to standard GP.

Streszczenie

Celem pracy było zaproponowanie nowego podejścia do obliczania wartości funkcji przystosowania osobników podczas ewolucji w opartym na drzewach programowaniu genetycznym (GP), odmianie obliczeń ewolucyjnych, w której osobniki kodują nie dane, lecz programy. Zakres zastosowania omawianego podejścia jest ograniczony do problemów boolowskich, w których znane są pożądane wartości wyjściowe dla wszystkich kombinacji wartości zmiennych wejściowych (przypadków użyteczności, ang. *fitness case*). Standardowa funkcja przystosowania używana w programowaniu genetycznym (GP) zwykle oblicza *obecną* użyteczność osobnika, tj. stopień, w jakim wartości wyjściowe rozważanego osobnika (drzewa) odpowiadają wartościom pożądanym (dla wszystkich przypadków użyteczności). Metoda zaprezentowana w tej pracy, nazywana dalej *potencjalną użytecznością*, próbuje ocenić użyteczność bezpośrednich potomków ocenianego osobnika.

Aby ocenić potencjalną użyteczność, analizowany jest wpływ operacji rekombinacji (zamieniających poddrzewa) na wartości wyjściowe osobnika. Potencjalna użyteczność jest obliczania względem podzbioru tylko tych przypadków użyteczności, dla których odpowiedź osobnika jest niezmiennicza względem operacji rekombinacji (krzyżowanie, mutacja). Osobnik jest nagradzany za każdy przypadek użyteczności, dla którego jego wartość wyjściowa jest zgodna z wartością pożądaną, oraz karany dla wszystkich tych przypadków użyteczności, dla których wartość pożądana i wartość wyjściowa osobnika różnią się. Przypadki użycia, dla któych wartość wyjściowa osobnika może zmienić się pod wpływem operacji rekombinacji, nie są brane pod uwagę, ponieważ niemożliwe jest przewidzenie, w jaki sposób operacje rekombinacji zmienią drzewo, a więc także jakie wartości wyjściowe będą ostatecznie zwrócone przez osobnika.

Podejście z potencjalną użytecznością zostało zaimplementowane w języku programowania *Java*, jako rozszerzenie do pakietu *Evolutionary Computations in Java (ECJ)*. Następnie zostało przetestowane na następujących problemach boolowskich:

- Odd Parity: 4-bit,5-bit,6-bit
- Multiplexer: 6-bit,11-bit
- Comparator: 4-bit,6-bit

Wyniki zostały porównane do wyników uzyskanych przy użyciu standardowego programowania genetycznego. Wszystkie obliczenia było prowadzone ze standardowymi ustawieniami *Koza-I*.

Na podstawie analizy wyników stwierdzono, że istnieje istotna statystycznie różnica pomiędzy wynikami obu metod. Podejście z potencjalną użytecznością okazało się być w większości przypadków lepsze, a we wszystkich nie gorsze niż standardowe programowanie genetyczne, pod względem zarówno częstotliwości znajdowania optymalnych rozwiązań, jak i średniej jakości najlepszego osobnika otrzymanego w pojedynczym przebiegu ewolucji. Ceną, jaką trzeba zapłacić za wzrost jakości wyników, są dłuższe czasy obliczeń. Uśredniając po wszystkich rozważanych problemach, stwierdzono, że zastosowanie omawianego podejścia powoduje średnio zwiększenie czasu obliczeń o 200% w stosunku do standardowego programowania genetycznego.

Acknowledgements

The author would like to extend his gratitude to dr hab. inż. Krzysztof Krawiec, the supervisor of this thesis and the originator of the idea to use context semantics as a basis for evaluating individuals in Genetic Programing. His remarks and advice have provided valuable insight and contributed to the improvement of the overall quality of this work.

Chapter 1

Scope of the thesis

The objective of this thesis is to investigate the properties of a novel method of calculating an individual's fitness in Genetic Programming (GP), a variant of evolutionary computation where solutions encode programs to be executed, applied to boolean function learning. This method will be further referred to as Potential Fitness (PF). In particular, the impact of the use of Potential Fitness as the evaluation function on the quality of results obtained using Genetic Programming-driven evolution will be examined. Also, a possibly efficient method of calculating Potential Fitness will be proposed. In order for this objective to be satisfied, the following subgoals will be achieved:

- comparison of the Potential Fitness-based evaluation method to the standard evaluation method in terms of success rate and average solution quality
- design of an efficient exact method for calculating Potential Fitness
- analysis of the performance of the exact method in terms of execution speed
- design of a heuristic method for obtaining a good estimate of the actual Potential Fitness value while substantially reducing the computational effort compared to the exact method
- analysis of the performance of the heuristic method in terms of execution speed and solution quality.

Chapter 2

Evolutionary algorithms and genetic programming

2.1 Introduction

Since the introduction of the concept of NP-complete problems after a series of articles in the early 1970s [7], the hypothesis that certain combinatorial problems cannot be solved on a Deterministic Turing Machine (DTM) in polynomial time has been gaining support. Today, the number of problems shown to be NP-complete has grown immensely, from the original 21 presented in Karp's 1972 article [21], to many thousands. Many of these problems are of key importance in real-world industrial, financial, medical and other applications. Therefore a lot of effort and resources have been allocated in order to find algorithms which provide optimal solutions while not exceeding reasonable running time bounds. However, an efficient (polynomial) algorithm for solving any NP-complete problem has yet to be discovered, making it impossible to find exact solutions to NP-complete problem instances even of moderate size.

Such a situation gave rise to alternate approaches to problem solving. The key difference was relaxing the constraint on solution optimality, instead seeking a tradeoff between solution quality and running time of the algorithm. This time marks the rapid development of heuristic and approximation algorithms.

One of the first of the new class of approximation algorithms was Christofides' algorithm for the Traveling Salesman Problem [5]. Christofides did not only provide the algorithm, but also proved that in a worst-case scenario the tour calculated by it would be no more than 1.5 times longer than the optimal tour. In contrast, there exist problems for which it has been proved impossible to find an algorithm with a constant performance guarantee, such as the Maximum Clique Problem. Such theoretical results increased the demand for heuristics which would still be able to provide good results, although without any theoretical guarantee.

An excellent example of such a heuristic is the Lin-Kernighan algorithm for the Traveling Salesman Problem. This local-search-based routine, and its refinements, has been known to produce outstanding results. Helsgaun reports solving every TSP instance available as part of the TSPLIB set to optimality [17].

The main disadvantage of standard heuristics is the fact that they largely remain problem-specific. Moreover, it is often difficult to design adequate heuristics for a given combinatorial problem. Therefore the task of finding more universal methods of problem solving has received much attention.

2.2 Metaheuristics

Problem-solving methods collectively known as *metaheuristics* aim to alleviate domain specificity and provide a framework which can be used across many unrelated, structurally different problems. Although metaheuristics provide a general outline of an algorithm, they usually do not constitute an actual, ready-to-use algorithm *per se*. In order to construct a routine which can be applied to a concrete problem, it is necessary to provide some domain-specific elements. However, in the case of metaheuristics, it is usually easy to provide these elements, due to the fact that they are natural components of the problem's definition, e.g. a function calculating the quality of a given solution, or a neighbourhood operator transforming the current solution into a neighbouring one. Therefore it is usually much simpler to obtain an algorithm from a metaheuristic template by filling in the missing components, than to develop a dedicated heuristic for the problem from scratch.

2.2.1 Intensification and diversification

Two important concepts in the design of metaheuristics are solution *diversification* and *intensification*. The former is associated with the metaheuristic's ability to examine a broad spectrum of candidate solutions, originating from distant regions of the solution space. The latter concerns the way the algorithm iteratively converges to better solutions. Both mechanisms combine to form a set of heuristic rules for creating new, hopefully better solutions on the basis of the ones examined in previous iterations. These two mechanisms are in a way contrary, because their goals are different. Usually, the most successful metaheuristics find a good balance between solution intensification and diversification to ensure best results.

2.2.2 No Free Lunch Theorem

Since metaheuristics are well suited to solving many different classes of problems, it may seem tempting to look for a single method of solving *every* class of problem.

However, Wolpert and Macready have shown in their paper [10] that if certain general conditions are met, for any algorithm, any elevated performance over one class of problems is offset by performance over another class. The collection of theorems, known collectively as No Free Lunch theorems, effectively implies that a perfect global optimizer cannot exist. Therefore it is likely there will always be demand for developing new metaheuristics, better suited to a specific subset of problems.

2.2.3 Performance

Metaheuristics are a versatile tool for solving a broad class of problems. They are extensible in that it is usually possible to enrich them with domain-specific operators which are tailored to a given problem. Studies exist which investigate the relationship between how much domain knowledge is incorporated into the algorithm, and the quality of the obtained result. Not surprisingly, the general purpose operators usually fall behind hybrid solutions in terms of performance.

As an example, consider the Traveling Salesman Problem. Johnson and McGeoch [12] performed a comprehensive case study spanning over a broad spectrum of both dedicated heuristic algorithms and general metaheuristics. Their results indicate that, given a moderate execution time limit, virtually all standard metaheuristic methods are outperformed by the superb dedicated heuristic for the TSP, the Lin-Kernighan algorithm. As the time limit increases significantly, some metaheuristics - Simulated Annealing and Genetic Algorithms - are able to produce better results than many independent iterations of LK. If time is not a limiting factor, once again both SA and GA are beaten by a special version of the Iterated LK heuristic. However, the best results were obtained by combining a genetic algorithm with the iterated LK algorithm. This means that a background knowledge-enriched metaheuristic was able to outperform all other approaches given enough time.

As Johnson and McGeoch point out, the Traveling Salesman Problem is somewhat an exception among NP-Hard optimization problems, due to the fact that it has been studied for a long time, and many dedicated methods have been developed specifically to deal with this one problem. In contrast, for most problems which arise in real-world applications there are no known dedicated heuristics with the excellent execution time and solution quality properties of the Lin-Kernighan algorithm. Therefore, in many cases designing background knowledge-enriched operators for existing metaheuristics is the most popular course of action for solving these problems.

An ample example is the Quadratic Assignment Problem. Despite having been studied for many years since its formulation in 1957, no dedicated heuristic has been designed as of yet which would produce good results regardless of the structure of the

concrete QAP instance being solved. Moreover, Queyrane [31] showed that unless P equals NP, it is impossible to find an approximation algorithm with a constant performance guarantee even if the instance's coefficient matrices are guaranteed to fulfill the triangle inequity. Currently, the best known approaches to solving QAP are hybrids of tabu search and genetic algorithms enriched with a problem-specific crossover operator [28].

2.2.4 Popular metaheuristics

In this section, some classic metaheuristic methods will be presented. Although many years have passed since their introduction into computer science literature, they still remain widely used as both stand-alone algorithms, and parts of complex multi-tiered cooperative metaheuristics.

Local search

One of the simplest and most commonly used metaheuristics, Local Search is often the first step in finding a good algorithm for an NP-hard problem. Also, it has been the basis for developing more advanced algorithms, like Tabu Search and Simulated Annealing.

In order to be able to apply Local Search to a problem, a solution neighbourhood operator needs to be defined. Such an operator transforms a given solution into a structurally adjacent, or neighbouring, solution. The significance of structural adjacence is bound to the specific problem. It is up to the algorithm designer to define a neighbourhood relation on the solution space, taking into account the extent of structural changes between neighbouring solution, as well as the size of the neighbourhood. To illustrate this with an example, a sample solution neighbourhood for the symmetric Traveling Salesman Problem will be defined.

Consider an instance of the Symmetric TSP problem of size N. We are given a weighted undirected graph G=(V,E), where $|V| = N$. The goal is to find a simple cycle within G which minimizes the sum of weights over all edges forming the cycle. Therefore, any feasible (structurally correct) solution is a sequence of N distinct cities, corresponding to exactly 1 cycle within G. Now consider a sample solution with 2 edges removed. What remains are 2 disjoint sequences of vertices, and there is only one way to assemble these 2 sequences to obtain a tour distinct from the starting tour (see Fig. 2.1).

This is how the 2-opt neighbourhood operator transforms one solution into another. Since there are $\binom{N}{2}$ possible ways 2 distinct edges can be chosen from a set of N edges, the size of the 2-opt neighbourhood is a square function of the number of vertices in G. This can be generalized to any neighbourhood of dimension k - the size of a k-opt STSP neighbourhood is $O(N^k)$.

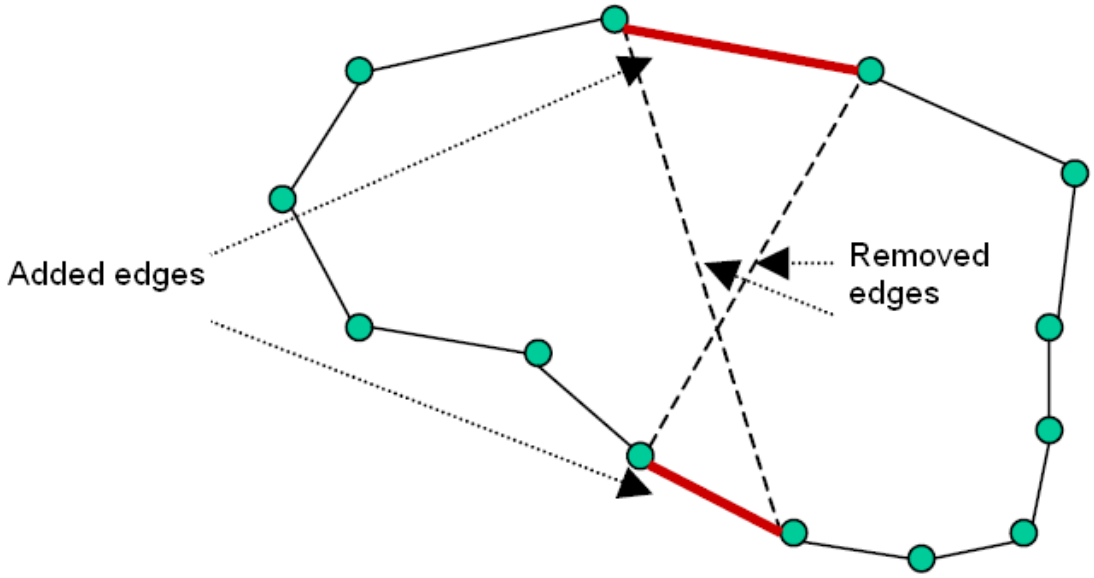

FIGURE 2.1: A 2-opt move for the Symmetric TSP

Once a neighbourhood operator is defined, the Local Search iteratively searches the current solution's neighbourhood (or its subset) and chooses either the first solution with a better goal function value ('Greedy' version) or the best-valued solution within the examined portion of the neighbourhood ('Steepest' version) as the basis for the next iteration. The algorithm stops once there are no more improving moves, that is no neighbouring solution has a better goal function value than the current one.

This is a significant weakness of the Local Search algorithm - what follows is that it can never reach a region in the solution space to which there exists no path comprised of strictly improving moves. In other words, LS can only find the local extremum of the goal function in the starting solution's vicinity - hence its name.

Tabu search

In order to alleviate this weakness of LS, Tabu Search was proposed as a natural extension. Tabu Search assumes that a special type of short-term memory is maintained, the tabu list. Every time a move is performed, the current solution is added to the tabu list. Following that, the use of this solution is disallowed for k consecutive iterations, where k is called the tabu list length. Moreover, the algorithm can now perform deteriorating moves. These two properties enable it to traverse the solution space unhindered by irregularities in the fitness landscape. However, although the tabu list makes it less likely for the algorithm to fall into a solution loop, it is still possible if the tabu list's length is too small.

Through the introduction of the tabu list, the Tabu Search metaheuristic pro-

vides a basic diversification method. This is an improvement over Local Search, which did not make use of any explicit mechanism for solution diversification. However, marking every visited solution as tabu usually results in a search which is too restrictive, and interesting regions of solution space are overlooked. Therefore, the tabu list is usually used in conjunction with so-called *aspiration criteria*, which override a solution's tabu status if certain conditions are met. These conditions are usually connected with solution quality, although other criteria may be used as well.

Tabu search does not by any means guarantee obtaining the holistic optimal solution, however its performance is usually much superior to that of a standard LS algorithm. This metaheuristic is extremely popular and has found numerous fields of application, therefore many different variations have been developed, whose common denominator is the presence of dynamic, adaptive memory [14].

Simulated annealing

Another metaheuristic building on the foundation provided by Local Search, Simulated Annealing (SA)[22] owes its name to the industrial process of recrystallization of metals and liquids. In this process, the metal is initially heated to a very high temperature at which its atoms can move relatively freely. The system is then slowly cooled down, and the atoms' ability to move is gradually constrained, which causes them to find the most stable spatial conformation, i.e. the one that minimizes the system's total energy. If the cooling is done too quickly, or the starting temperature of the melt is too low, some portions of the system may become trapped in local energy minima.

Simulated Annealing forms analogies between the cooling of a physical system and solving a combinatorial problem. The current state is the current solution, the system's energy is the goal function value, and the frozen state corresponds to the global minimum. However, the temperature of the system has no straightforward counterpart in most optimization problems.

The cooling is simulated by dividing the execution of the algorithm into epochs which correspond to decreasing values of temperature of the system. The metaheuristic then applies the Metropolis Monte Carlo simulation method [27] to traverse the solution space. All transitions to improving moves are accepted unconditionally, whereas the probability of transition from solution S to a worse solution S' is an exponential function of the difference in goal function values (Δf), and current temperature T of the system : $e^{-\frac{\Delta f}{kT}}$, where k is a constant.

The difficulty in using Simulated Annealing properly concerns finding the right values for parameters such as:

- initial system temperature

- number of different temperature levels
- number of steps at each temperature level

These parameters are often domain-specific and they result from the problem's fitness landscape. Failure to provide good parameter values may lead to poor performance of the algorithm, however if adequate parameters are used, Simulated Annealing is capable of producing excellent results.

2.3 Biologically inspired computation

A broad group of metaheuristics is based on the observation of natural phenomena. These phenomena, whether on cellular, specimen or even social level, are linked by the an interesting common property. Usually it is the case that such biological systems are comprised of relatively simple entities, or agents, and their interactions, but the overall behaviour of the system appears to be complex and it is difficult to explain when considering only a single agent. This behaviour is often the result of cumulated non-linearities in the interactions between individual components. Moreover, such systems are able to adapt to changing external conditions, and they demonstrate a high tolerance for partial loss of inter-component connectivity.

Examples of such systems which are interesting in the context of designing metaheuristics include [6]:

- *neural networks* - an individual neuron can be viewed as a simple element which integrates electric signals and outputs another electric signal if certain conditions are met. However, if enough neurons are grouped together, and they are exposed to the right external conditions for a long enough period, they form the human brain, which is capable of thinking, learning etc.
- *ecosystems* - an individual animal may be regarded as a self-oriented agent whose only goal is to ensure the survival of its species. However, if a sufficiently long period of time is analyzed, it will become apparent that due to certain 'simple' mechanisms such as random genetic variation and Darwinian survival-of-the-fittest selection, the animals are in fact changing in order to better adapt to the environment they live in.
- *marketplace* - a trader in a market can be viewed as an agent interested only in maximizing his profit from the transactions he participates in. Sellers want to achieve the highest price for their wares, while buyers want to obtain them at the lowest possible price. This competition between two groups of interest may, in the right circumstances, lead to the spontaneous regulation of the traded resource's price to a level which corresponds to its optimal allocation.

Moreover, this adjustment occurs without any external supervisor, and if the conditions (i.e. amount of resource on the market) change, the price readjusts and moves to a new equilibrium.

The design of biologically-inspired metaheuristics remains largely influenced by the field known as Artificial Life. The goals of both fields are not the same, as Artificial Life seeks to answer questions about life itself, rather than try to apply biological mechanisms to combinatorial optimization. However, advancements in both fields are connected and any significant result obtained in one is usually beneficial in the other.

2.3.1 Ant Colony Optimization

Originally proposed by Marco Dorigo [11], this family of metaheuristics is inspired by the behaviour of ants which search the surroundings of their hive in order to acquire food. Initially, the ants move around the hive in a random fashion. When an ant finds some food, it takes the food back to the hive, marking its path with *pheromone*, a chemical substance which attracts other ants. After the pheromone has been laid down, it is less likely for other ants to wander randomly; instead, they will probably follow the pheromone trail. Moreover, if another ant finds food at the end of the marked path, it will reinforce the pheromone mark, making it even more attractive to other individuals (ants). However, the pheromone is subject to evaporation, therefore its effect on the ants is weakened with time (if not reinforced). This is an important mechanism, which allows the ants to constantly find shorter paths to the food. If the pheromone didn't evaporate, the ants' paths would have been greatly determined by the first, random phase. If we regard the process of gathering food by ants as an optimization problem of finding the shortest path from the hive to the food, this would mean that the process would be stuck at a local optimum. The pheromone evaporation allows the process to move to another part of the solution space.

The Ant Colony Optimization-class (ACO) metaheuristics attempt to mimic this social behaviour of ants. More formally, an ACO algorithm operates on a directed graph G=(V,A). It is iterative in nature. At every step, it generates K solutions, corresponding to K independent 'ants', in a random fashion. However, the generation is done with a certain probability distribution. The probability of an agent moving from vertex i to vertex j is given by:

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})} \tag{2.1}$$

where:

- $\tau_{i,j}$ is the amount of pheromone present on the arc (i,j)

- $\eta_{i,j}$ is a heuristic, *a priori*-knowledge-based quality of arc (i,j). This is usually inversely proportional to the distance between nodes i and j
- α and β are parameters which determine how much the amount of pheromone, and the a priori arc quality, respectively, influence the final probability.

After the K solutions have been generated, the pheromone update phase follows. The new amount of pheromone accumulated on arc (i,j) is calculated, taking into account the previous value, the evaporation rate (ρ) and the contributions made by ants in the current iteration $\Delta\tau_{i,j}$:

$$\tau_{i,j}^{l+1} = \rho\tau_{i,j}^{l} + \Delta\tau_{i,j} \tag{2.2}$$

where $\Delta\tau_{i,j}$ is given by:

$$\Delta\tau_{i,j} = \sum_{k \in A} \frac{1}{L_k} \tag{2.3}$$

where A is the set of indices of ants which visited arc (i,j) in the current iteration, and L_k is the length of the path of ant k.

After the redistribution of pheromone, an additional phase called *daemon actions* may occur. Daemon actions can be used to implement centralized actions which cannot be performed by single ants, such as the invocation of a local optimization procedure.

The entire cycle (solution generation, update of pheromone quantities, daemon actions) is repeated until some stop condition is satisfied. An important advantage of Ant Colony Optimisation over more classic techniques such as Simulated Annealing or Tabu Search is that ACO can dynamically respond to changing the structure of input data, and reconfigure the solution to match the changes. This is particularly important in applications such as network routing, where hardware failures may spontaneously occur and a quick reconfiguration of network traffic routes is of key importance.

2.3.2 Particle Swarm Optimization

This stochastic optimization technique, introduced by Kennedy and Eberhart [20], is based on a simulation of a simplified social model of bird flocks and fish schools. It was in part inspired by previous research of Heppner and Grenander concerning the simulations of synchronous bird flock maneuvers during flight. Both scientists had the insight that processes which occur at the level of an individual bird influence the dynamics of the entire flock. Therefore their models were based on the manipulation of distances between individuals, and the behaviour of the flock was regarded as the superposition of every individual's attempt to maintain an optimal distance between

itself and its immediate neighbours. This is an example of an *emergent* phenomenon, where the local interaction of each part with its immediate surroundings causes a complex chain of processes leading to some order on the global level.

Like the previously discussed Ant Colony Optimization technique, Particle Swarm Optimization is an iterative algorithm. Also similarly, at each step a population of independent *particles* is maintained. The particles represent potential solutions, i.e. they are points in solution space. The population is initialized in a random fashion. Each particle has a vector of velocities associated with it. The dimension of the vector is equal to the dimension of the problem being solved. Also, for every particle, the following information is maintained: the best solution visited so far by the particle, denoted *pbest*, and the best solution found by k immediate neighbours (in terms of solution space) of the particle under consideration, referred to as *lbest*.

At every step of the simulation, the velocity vector of every particle is adjusted. For particle i, dimension j, and iteration number k, the new partial velocity is calculated as:

$$v_{i,j}^{k} = v_{i,j}^{k-1} + 2rand() * (pbest_{i,j} - current_{i,j}) + 2rand() * (lbest_{i,j} - current_{i,j}) \quad (2.4)$$

where:

- rand() is a random real number,
- $current_{i,j}$ is the value of particle i's coordinate j in the solution space

Particle Swarm Optimization has the advantage of being very simple to implement and not requiring any complex computations. Yet despite its simplicity it has proven quite effective in solving several classes of problems.

2.3.3 Artificial Neural Networks

History

The origins of this metaheuristic can be traced back to the 1940s, when McCulloch and Pitts attempted to create computer models of biological neural networks based on contemporary neurological knowledge. The newly-emerged field gained much interest, which continued into the mid 1960s. During this period, many important contributions were made, such as Rosenblatt's *Perceptron* model, or the *ADAptive LINear Element (ADALINE)* by Widrow and Hoff (1960). Beginning in late 1960s, a period of disappointment began with little interest offered by the scientific community to the topic of artificial neural networks. It was not until the 1980s that the area experienced a revival, with the significant article collection *Parallel Distributed Processing*, edited by Rumelhart and McClelland, being a mark of its reestablishment as an important method in artificial intelligence.

Biological and artificial neurons

Artificial Neural Networks (ANNs) can be thought of as a data processing paradigm where many simple, interconnected processing entities (neurons) collaborate to solve a problem. The approach is based on a simplified model of the learning processes which occur within the human brain.

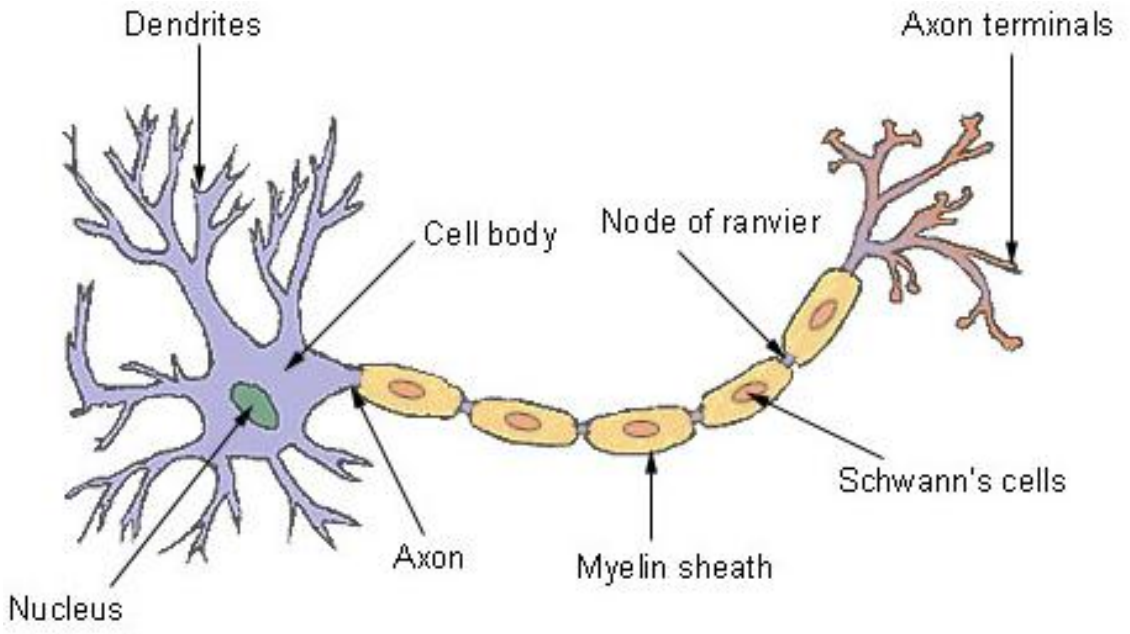

FIGURE 2.2: Structure of a neuron. Source: http://training.seer.cancer.gov/module$_a$*natomy/images/illu*$_n$*euron.jpg*

A neuron is a specialized biological cell capable of transmitting electrochemical impulses. It obtains input signals from surrounding neurons through structures called *dendrites*, which are cytoplasmatic extensions projecting from the cell body (Fig. 2.3). Inside the cell body, the signals received through the dendrites are aggregated. When a neuron becomes active, it sends out its own electrochemical impulse through the *axon*, a long cytoplasmatic extension which splits into thousands of branches. At the end of each branch, structures called *synapses* are formed. Synapses connect the axon to dendrites of adjacent neurons, thereby creating a neural network.

Upon receiving an input impulse, a neuron does not always become active. The sum of all incoming impulses is compared to the neuron's individual threshold voltage value. If the sum is greater, the neuron fires its own impulse, which is then propagated to neighboring neurons through the synapses.

It should be noted that the strength of a signal which a neuron receives through any of its dendrites is not constant, but rather it depends on the effectiveness of the source synapse. Moreover, a synapse's effectiveness can also vary with time, as it

is associated with certain biochemical factors (e.g. the amount of *neurotransmitter* present within the synapse).

This means that from a mathematical standpoint, the aggregation which occurs within the neuron's body can be viewed as a weighted sum of signals, where weights correspond to the effectiveness of the synapses. This is how the artificial neuron models its biological counterpart. Figure 2.3 depicts the complete model.

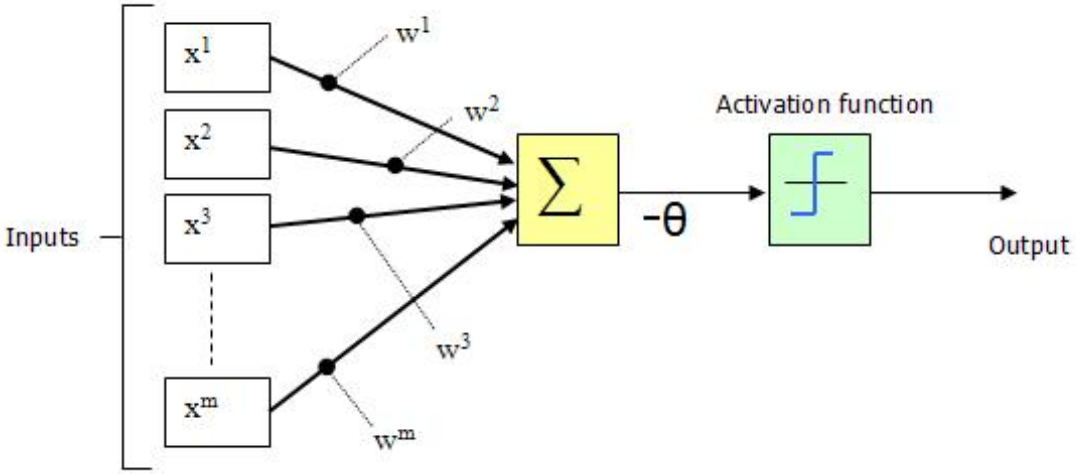

FIGURE 2.3: Artificial neuron. Source: http://69.10.233.10/KB/recipes/NeuralNetwork1.*aspx*

The subtraction of the neuron's threshold value θ from the weighted sum of inputs is a way to standardize the activation function - the neuron fires when this difference is greater than 0.

Although in biological neurons the activation function is a threshold function, such a function is rarely used in artificial neurons due to the fact that it is problematic for most network training algorithms because of its non-differentiability. Instead, the sigmoid function is used:

$$f(x) = \frac{1}{1 + e^{-\beta x}} \tag{2.5}$$

Learning

One of the more influential researchers of neurological systems (Donald Hebb) believed that the process of learning in biological neural networks occurs mainly through the effectiveness of the synapses, i.e. the weights of the information inputs in every neuron [36]. This approach is followed also in the learning of artificial neural networks.

Artificial Neural Networks learn by example. They cannot be programmed to perform a specific task. Instead, a neural network is fed a set of example objects,

and it modifies its weights accordingly in order to optimize some quality criterion. The learning process is done using a specified learning algorithm.

All learning methods used for adaptive neural networks can be divided into two major categories:

- *supervised learning* - for every training object, the desired output of the network is known. Therefore, an error function can be defined which measures the network's total divergence from the desired values for the entire training set. The aim is to obtain a set of weights which minimizes this error function.
- *unsupervised learning* - no explicit desired return values are known - therefore no error can be measured. This type of learning is sometimes referred to as *self-organisation*, in the sense that it self-organises data presented to the network and detects their emergent collective properties.

The learning process is usually iterative. Weights of the neurons' data inputs are systematically changed, and the network's performance is reevaluated using the new set of weights. This process continues until some stop criterion is met, e.g. the total error has decreased below a threshold value, or the difference in error in two subsequent iterations does not change significantly.

Architecture

In order to be usable in any real application, every artificial neural network must possess data inputs and data outputs. Therefore, an ANN contains at least two layers of neurons - the input layer and the output layer. Also, additional layers are usually present, referred to as *hidden* layers. The neurons in these layers have internal functions within the network, and they are not directly observable neither from the input, nor the output side (see Fig. 2.4).

There are two main classes of network architectures. The distinction is made with respect to whether or not we allow feedback connections, i.e. connections from the outputs of a neuron in layer l to the inputs of a neuron from layer k ¡= l.

- *feed-forward network* - feedback loops are not allowed. The signal flows in one direction only.
- *feedback network* - they allow the formation of connections in both directions. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations.

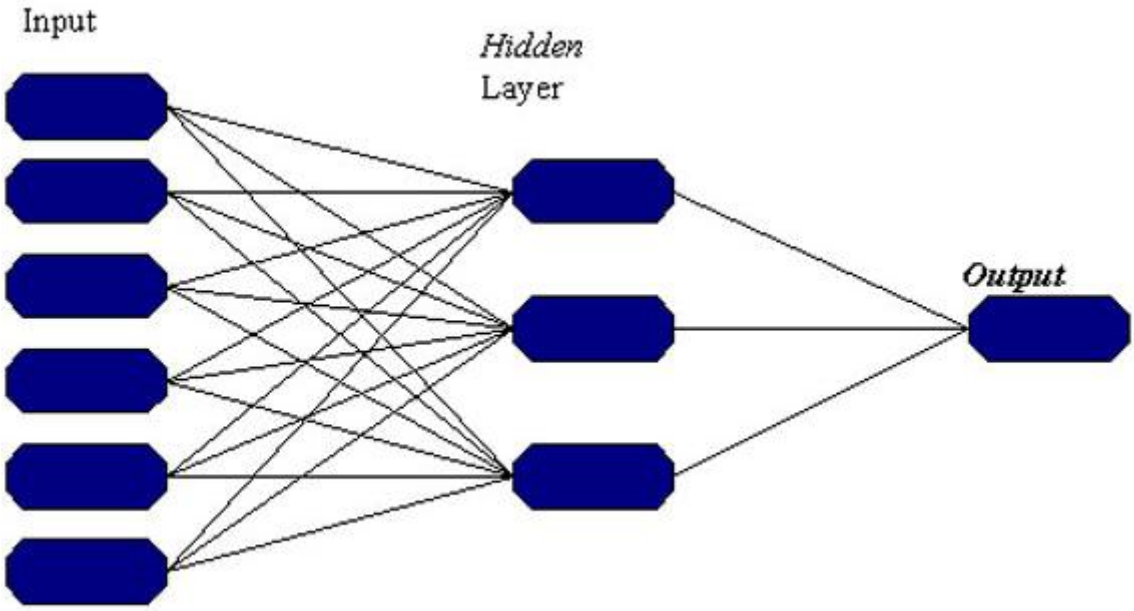

FIGURE 2.4: A simple feed-forward network. Source: http://www.hml.noaa.gov/ohh/images/margen/ann.jpg

Applications

Neural networks have proven to be a versatile tool, capable of modelling very complex functions. In particular, their non-linear characteristics make the spectrum of their applications very broad. Neural networks are well suited to tasks of data processing and analysis, prediction and classification. So far, they have been successfully applied in areas such as function approximation, pattern recognition, biomedical engineering, time series analysis, stock market prediction, neuropsychology, physics, geology, and many others [36].

2.4 Evolutionary algorithms

This family of metaheuristics is inspired by the biological processes of evolution, adaptation and Darwinian survival of the fittest. The algorithms perform computations by simulating the evolution of a population of individuals, which correspond to potential solutions in the solution space of the underlying problem.

2.4.1 Outline of operation

The simulation is carried out over a specified number of iterations. It may also end prematurely if some convergence criterion is met, e.g. the improvement in solution quality in k consecutive iterations has not exceeded Δf_{thr}. Every iteration consists of three general phases - *selection*, *recombination* and *mutation*.

Selection

This phase is a realization of the survival-of-the-fittest mechanism. During selection, solutions (individuals) currently available are evaluated with respect to their quality. The evaluation function, defined on the set of all possible solutions, is referred to as the *fitness function*, and it is highly domain-specific. This *fitness function* is analogous to an individual's abstract ability to survive in its environment. Once all individuals have been evaluated, some method is applied in order to select a subset of individuals which will proceed to the next phase. These are the individuals which managed to 'survive'. The purpose of this is to ensure that the better individuals within the population survive, and thus to make the entire population converge towards better solutions with each iteration. As in the case of real, biological organisms, selection is an important driving force of the evolutionary process. In the context of metaheuristics, it may also be viewed as a mechanism for solution intensification as defined in Section 2.2.1.

Many different selection methods have been developed for various applications of evolutionary algorithms - some deterministic, others randomized. Most of these methods make use of two values associated with every individual i: the probability of reproduction p_i and the expected number of copies e_i. They are defined as:

$$p_i = \frac{fitness_i}{\sum_j fitness_j}, e_i = N * p_i \tag{2.6}$$

where N is the population size. In this section, a couple of the most popular selection methods will be described.

- *fitness proportionate selection* - randomly draws N individuals from the population, selecting each individual i with probability p_i.
- *deterministic selection* - Define $[x]$ as the largest integer not exceeding x. First, every individual i is copied $[e_i]$ times into the new population. The remaining l spots are populated based on descending values of the fraction part of e_i.
- *random k-tournament selection* - randomly chooses k individuals from the population, and selects the one with the best fitness value. This procedure is repeated N times.
- *randomized remainder selection* - this is a combination of deterministic and fitness proportionate selection. Every individual i is copied $[e_i]$ times into the new population. Following that, the fitness proportionate principle is applied to select the remaining individuals, with fraction parts of each individual's e_i used as the probability of selecting that individual.

Recombination

The role of the recombination operator is to produce new individuals, or *offspring*, based on the information contained in two or more base individuals, or *parents*. This process is an analogy of mating in biological populations. The details of how the parents' structure is actually combined are dependent on the specific algorithm and the problem being solved. Designing adequate recombination operators is one of the greatest challenges in implementing evolutionary algorithms. In Genetic Algorithms (GA) and Genetic Programming (GP), which will be discussed in detail later in this chapter, recombination operators are usually referred to as *crossover* operators.

Mutation

Mutation is a unary operator, which changes the structure of an individual in a random fashion. One again, this operator is inspired by the biological process of mutation which occurs in the genotypes of living organisms.

Recombination and mutation of individuals are a means to ensure sufficient diversity of solutions within the population. This allows the evolutionary algorithm to get out of local optima which may be present in the fitness landscape.

2.4.2 History

The origins of evolutionary computation can be traced back to the 1950s. However, due to lack of adequate hardware, as well as some methodological problems present in those early attempts, the field remained relatively unknown to the scientific community [3]. With contributions from Holland, Rechenberg, Schwefel and Fogel in the 1970, a solid methodological foundation was established. The aforementioned authors introduced three strongly related, but independently developed methods, now considered mainstream - *Genetic Algorithms* (Holland, [18, 19]), *Evolutionary Strategies* (Rechenberg [32], Schwefel [34]) and *Evolutionary Programming* (Fogel [15]). In the 1980s, another class of evolutionary algorithms, *Genetic Programing*, was introduced through the publications of S.F.Smith[35], N. Cramer [9], J.Schmidhuber [33] and J.Koza [23].

2.4.3 Parameter encoding

Before the various evolutionary algorithms are discussed in detail, an important aspect of their design must be addressed - the representation of the individuals and the way they encode the problem-specific parameters.

Genotype and phenotype space

In real-world applications, the problem's search space is defined by a set of objects, each with unique characteristics. The subset of parameters which is subject to optimization forms the *phenotype space*. Conversely, genetic operators (mutation, recombination) often accept abstract mathematical objects (e.g. binary strings, arrays of integers etc.) as input parameters. These mathematical objects constitute the *genotype space*. Therefore it is necessary to provide a mapping function between both spaces.

The concepts of genotype and phenotype spaces are derived from analogous concepts present in the biological studies of heredity and development of organisms. The genotype of an organism is the class to which that organism belongs as determined by the description of the actual physical material made up of DNA that was passed to the organism by its parents at the organism's conception. For sexually reproducing organisms that physical material consists of the DNA contributed to the fertilized egg by the sperm and egg of its two parents. For asexually reproducing organisms, for example bacteria, the inherited material is a direct copy of the DNA of its parent. The phenotype of an organism is the class to which that organism belongs as determined by the description of the physical and behavioral characteristics of the organism, for example its size and shape, its metabolic activities and its pattern of movement.

Designing a problem representation

In general, two popular approaches exist. The first is to choose one of the standard algorithms and to design a mapping function in accordance with that algorithm's requirements. The second is to design such a representation which is as close to the phenotype space as possible, thereby avoiding the problem of mapping spaces altogether.

An important advantage of the first approach is that many empirical and theoretical results are available for the standard types of evolutionary algorithms. On the other hand, a complex coding function may introduce additional non-linearities and certain mathematical difficulties, significantly obstructing the search process.

2.4.4 Genetic Algorithms

Genetic Algorithms have been applied to many optimization problems, both discrete and continuous [16]. They remain the most efficient known ways of solving certain hard combinatorial problems.

Representation

Original genetic algorithms, as proposed by Holland, operated on individuals encoded as binary strings of fixed length. Therefore, they usually enforce the use of encoding and decoding functions of the form:

$$h : M \to \{0,1\}^l, h' : \{0,1\}^l \to M \tag{2.7}$$

which map solutions $\vec{x} \in M$ to binary strings $h(\vec{x}) \in \{0,1\}^l$. For problems where there are many parameters of different types/ranges, this can lead to quite complex mappings *h* and *h'*.

The reason why Genetic Algorithms adhere to binary representations of individuals is a result obtained in *schema theory*, a theory which can be applied to the study of the performance of genetic algorithms. The term *schema* refers to a string over the alphabet $\{0,1,*\}$. This string can be viewed as a template for binary strings, where the characters *0,1* are fixed, and the symbol * represents any of the two. Consider the template *00**. The two binary strings which match this template are *000* and *001*.

The *schema theorem* of genetic algorithms states that the classic genetic algorithm, with the assumption that the operators of mutation and crossover are detrimental, delivers a near-optimal sampling strategy for schemata in the course of evolution [16]. However, this result applies only to an optimization criterion where the collective fitness of all individuals processed during all iterations of the evolution is optimized - not to the desired criterion of finding the single optimal value. Moreover, certain experiments have shown that the binary encoding function can actually make the problem more complex by introducing an additional multimodality.

Mutation and recombination

Mutation in Genetic Algorithms was originally introduced as an auxiliary operator of small importance. It changes an individual's binary representation by randomly flipping every bit with a small probability p_m. The classic version of the mutation operator assumes that p_m remains constant over the entire simulation, however studies exist which reveal that introducing a variable, or even self-adapting, mutation rate, may be helpful in terms of convergence reliability and velocity [2].

As for the recombination operator, the original Genetic Algorithm performed a one-point crossover. This operation is carried out in the following manner: two parent individuals are randomly picked from the population, and a crossover point (position within the bit string) is chosen. The bit strings of both individuals are split at the crossover point, thus yielding 4 fragments, which are then assembled in such a way that the left substring of an individuals is concatenated with the right substring of the other individual. Therefore, one crossover operation produces 2

new individuals, while destroying the parents. The crossover is usually applied with a certain probability, which enables some individuals to pass to the next generation unaltered.

Certain modifications of the crossover operator have been proposed. These include increasing the number of crossover points, and the so-called *uniform crossover* method, where every bit of the new individual's string is randomly chosen from one of the parents. Another interesting extension is the participation of more than 2 parents in the crossover. This generalized crossover operator is reported to improve convergence properties of the genetic algorithm in some applications [1].

2.4.5 Evolutionary Strategies

This technique was developed by Rechenberg and Schwefel at the Technical University of Berlin for the purpose of solving difficult discrete and continuous optimization problems.

Representation

The classic version of Evolutionary Strategies uses a representation based on fixed-length vectors of real numbers. There is no need for a decoding function - the elements of the vector are phenotypic features of the individual. However, additionally to the object variables, every individual also keeps track of its set of *strategy parameters*. These parameters enable the algorithm to evolve not only the problem-specific variables, but also its own evolution parameters. More formally, every individual $\vec{i} = (\vec{x}, \vec{\sigma})$ consists of object variables $\vec{x} \in \Re^N$ and strategy parameters $\vec{\sigma} \in \Re_+^N$

Reproduction and self-adaptation

Mutation is performed independently on every element of the real vector. For an element x_i, the new value $x_i^{'}$ depends on the current value, and the corresponding element in the strategy parameters vector:

$$\sigma_i^{'} = \sigma_i * exp(\tau^{'} * N(0,1) + \tau * N_i(0,1)) \tag{2.8}$$

$$x_i^{'} = x_i + \sigma_i^{'} * N_i(0,1) \tag{2.9}$$

where the notation $N_i(.,.)$ indicates that the random variable is resampled for every index value i, whereas τ and $\tau^{'}$ are called *learning rates*.

Mutation is thus applied to the problem variables as well as the evolution parameters. This contrasts with the classic genetic algorithm template, where the mutation and crossover probabilities were fixed.

Evolutionary Strategies also make use of a recombination operator. New individuals are created based on ρ parents, $2 =< \rho <= \mu$, where μ is the population size. The actual operations performed on the object variable part and the simulation parameter part of the individual's representation are usually different. Types of crossover include *discrete recombination* (similar to uniform crossover in Genetic Algorithms), *intermediary recombination* (usually arithmetic averaging or geometric crossover) and others.

Selection

A (μ, λ)-Evolutionary Strategy uses a deterministic selection method such that μ individuals (parents) create $\lambda > \mu$ offspring using the aforementioned mutation and recombination operators, and the new population is filled with μ best offspring (with respect to the fitness function).

Another widely used, also deterministic selection scheme is the $(\mu + \lambda)$-strategy. It chooses μ best individuals from the union of the parents and their offspring, which guarantees that the best fitness function value per generation cannot deteriorate in the course of the simulation.

2.4.6 Evolutionary Programming

Evolutionary Programming (EP) was introduced by Fogel as an attempt to create artificial intelligence. The idea was to evolve Finite State Machines, which represented individual organisms in a population of problem solvers. The FSMs were tasked with the prediction of events based on former observations. The performance of every individual can then be measured by comparing its prediction with the actual event.

Method of operation

Evolutionary Programming is not associated with a single representation of individuals. Vectors of real numbers (both fixed length and variable length), binary strings and matrices are among the representations successfully used with EP.

Evolutionary Programming relies almost exclusively on mutation as its reproduction operator. Recombination is not used.

Originally, the mutation operator was used to modify the Finite State Machines. An FSM is an abstract machine which transforms a sequence of input symbols into a sequence of output symbols, based on a finite set of states and a finite set of transition rules. During mutation, one of the following changes to the FSM was performed: addition/deletion of a state, change of the initial state, change of a transition rule, change of an output symbol.

As Evolutionary Programming was adapted for use with different problem representations, the mutation operator underwent changes to reflect the new representations. Currently, when dealing with a real-vector representation, the mutation operator operates in a manner similar to that defined in Evolutionary Strategies.

Evolutionary Programing uses a selection method which is a probabilistic version of ES' $(\mu + \lambda)$-scheme.

2.4.7 Extensions

The last two decades have seen an outburst of applications and research concerning evolutionary algorithms. Many approaches have been developed which elude classification to any of the three aforementioned mainstream categories, while still following the general paradigm outlined in Section 2.4.1. Specifically, many new dedicated representations of individuals have emerged, along with special mutation and/or recombination operators. Today, designing problem-specific representations and operators is one of the key elements of successfully applying evolutionary algorithms to difficult optimization problems.

2.5 Genetic Programming

Genetic Programming is another realization of the evolutionary programming paradigm. However, in Genetic Programming, individuals do not represent explicit solutions to the problem under consideration, but rather they are *programs*. These programs accept input data and produce corresponding output values. In this context, Genetic Programming may be regarded as an automated procedure of evolving programs to solve a particular problem.

2.5.1 Representation

Canonical Genetic Programming represents the programs (individuals) as tree structures. The nodes, which are the building blocks of the tree, can be divided into two categories - *terminal nodes* and *non-terminal nodes*. The former corresponds to input variables of the problem. Terminal nodes simply return the value of a variable for the input object currently being processed, and take no formal parameters. The latter is a category which contains functional nodes. Every non-terminal node is associated with a function which accepts one or more parameters (see Fig. 2.5). The evaluation of a functional node is a recursive process. Because a non-terminal node uses the return values of their child nodes as values for its function's formal parameters, the child nodes must be evaluated first.

The terminal nodes are leaves in the tree, whereas the number of a non-terminal's child nodes is determined by the arity of its associated function. Together, the sets of defined terminals and non-terminals form the *primitive set* of a GP system. Both sets of nodes are problem-specific and they must be defined by the user.

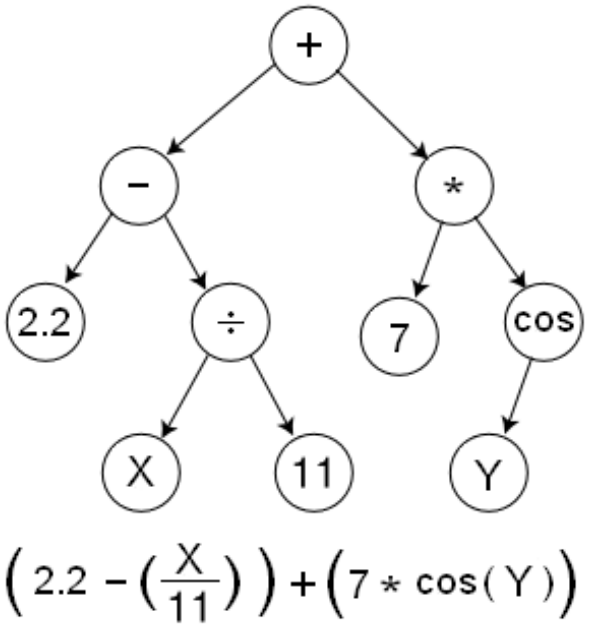

FIGURE 2.5: A sample GP individual and the expression that it implements

By using an explicit tree-like representation of programs, Genetic Programming avoids the problem of genotype-phenotype mapping, i.e. the mapping function is an identity function. However, other representations of individuals have also been applied to this methodology. One example of an alternative representation is the one used in *Linear Genetic Programming* [4], where programs are represented as linear sequences of instructions in an imperative programming language or machine code. This contrasts with the classic representation, which is more suited to functional programming languages such as *Lisp*. Another example is *Gene Expression Programming*, a technique proposed by Candida Ferreira [13], which makes a distinction between the genotype (a linear chromosome) and the phenotype (a tree expression).

2.5.2 Initializing the population

The population of programs is initialized in a random fashion. Among the many methods available in the literature, the two earliest are *full* and *grow*. Both methods generate trees with a depth not exceeding a user-supplied bound. The *full* method randomly picks nodes from the non-terminal set until the maximum allowed depth is reached. At this point, only terminal nodes are allowed. Such a procedure ensures that all the leaves in the generated tree are at an equal depth.

The second method, *grow*, does not restrict the node pool to non-terminals in the first phase. This may cause the process to end prematurely (i.e. before the maximum depth has been reached) if terminal nodes are selected too frequently. Therefore the *grow* method produces trees which vary in sizes and depths, as opposed to the *full* method. Also frequently used is a combination of both methods, referred to as *ramped half-and-half*, where half of the population is initialized using *grow*, and the other half using *full*, subject to some depth limits (hence the term 'ramped').

2.5.3 Genetic operators

Recombination

Genetic Programming uses *subtree crossover* as its main recombination operator. This type of crossover accepts two individuals as input. For each individual, a random node is picked, referred to as the *crossover point*. The subtree rooted at the crossover point in the first individual is replaced by the subtree rooted at the crossover point in the second individual (see Fig. 2.6). The modified first individual is returned as the result of the crossover process. It should be noted that the crossover is not performed on actual individuals from the population, but rather on their copies. This is done in order to preserve the parents' genetic material for use in potential future crossovers, as one individual may be selected for reproduction multiple times during one generation.

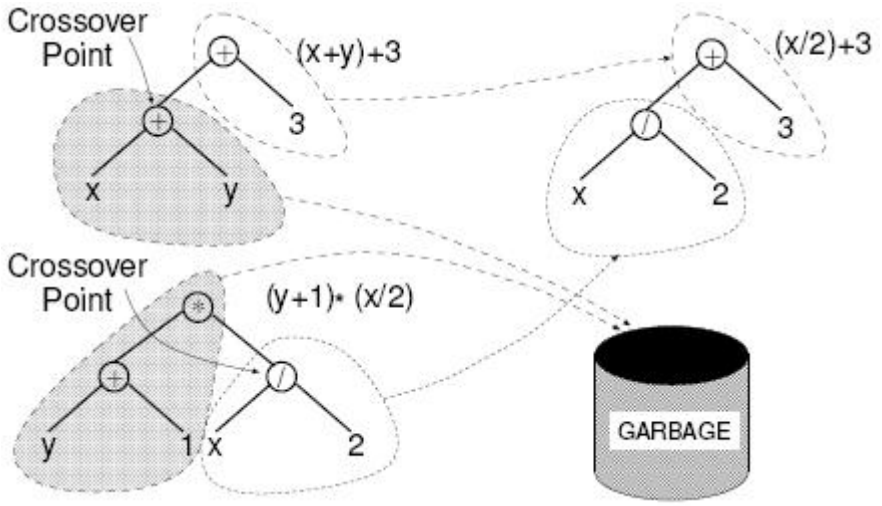

FIGURE 2.6: Subtree crossover. Source: [29]

Trees constructed based on a typical GP primitive set tend to have branching factors of two or more, which means that the majority of nodes in the tree are leaves. Therefore, choosing the crossover point with uniform probability will often result in only marginal exchange of genetic code (merely an exchange of leaves or small subtrees). To alleviate this effect, Koza introduced the method of selecting

functional nodes with 90% probability and terminals with 10% probability. Also, other crossover methods have been proposed, including methods which define their own probability distributions for selecting a node as the crossover point.

Mutation

There are at least 3 mutation operators in use with Genetic Programming. The first one is *subtree mutation*. It simply randomly selects a node in the tree and replaces the subtree rooted in it with a randomly generated subtree. A modification of this scheme is called *headless chicken mutation*, which is implemented as a crossover between the underlying individual and a newly generated tree. The third method is *point mutation*. It is analogous to the mutation operator defined in standard Genetic Algorithms. Every node is considered a candidate for mutation. With probability p_m, the node is substituted with a (randomly chosen) node of the same arity. That is, if the node is a non-terminal, it is replaced by a functional node with the same number of formal parameters, and a terminal is replaced by another terminal. If no node of the same arity exists within the primitive set, no action is performed.

Reproduction

Due to the fact that the mutation and crossover operators are applied with a certain probability, it is possible for an individual to pass to the next generation unaltered, i.e. without being subject to any of these two processes. This is referred to as *reproduction*.

2.5.4 Calculating fitness and selection

Evaluating an individual

Because individuals are programs, the evaluation process involves the execution of the program represented by the individual under consideration for some external input data. The obtained return values are then a basis for assigning some fitness value to the individual. There are many potential criteria to use when determining the individual's quality, some of which include [29]:

- *total error* between the obtained and the desired output values,
- *amount of resources* necessary to bring the system to a required state,
- *accuracy* of a program acting as a classifier or pattern recognition agent,
- *payoff* of a game-playing program,
- *compliance* of a structure with design criteria supplied by the user.

Selection

Usually, Genetic Programming makes use of the *tournament selection* method, which was discussed in Section 2.4.1. However, due to the modularity of evolutionary algorithms, any selection scheme can be applied.

2.5.5 Applications

Genetic Programming has produced some outstanding results in many unrelated areas, such as economic modelling, image and signal Processing, industrial process control, curve fitting, medicine, biology and others. Moreover, there have been several cases where GP was able to equal or even surpass the best known human solution. In 2 cases, this has led to a new patentable invention. Some of the human-competitive results include [29]:

- Creation of quantum algorithms, including a better-than-classical algorithm for a database search problem and a solution to an AND/OR query problem
- Synthesis of analogue circuits
- Creation of a cellular automaton rule for the majority classification problem that is better than all known rules written by humans

In this thesis, Genetic Programming has been applied to the task of learning boolean functions.

Chapter 3

Semantics of GP trees

The goal of this section is to introduce the concepts of *subtree contexts* and their *semantics*, which are vital to understanding the approach to evaluating individuals presented in chapter 4. Because this thesis deals with boolean functions, the aforementioned concepts will be defined for the boolean domain. However, they can be generalized to any finite domain.

3.1 Semantics of subtrees

Consider a GP individual representing a boolean function (see Fig. 3.1). Based on the ideas presented by Poli and Page in [30], the semantics of such an individual may be fully characterized by enumerating all possible input values and their corresponding outputs calculated by the function represented by the individual. Also, given a tree, the semantics of any of its subtrees may be calculated as the semantics of the function corresponding to the subtree.

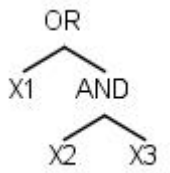

FIGURE 3.1: A tree representing the boolean function $f(x1, x2, x3) = OR(x1, AND(x2, x3))$

The number of possible inputs to a boolean function is determined by the number of its arguments. Thus, for a k-dimensional function, 2^k possible inputs exist. Table 3.1 illustrates the semantics of the function defined in fig. 3.1.

Following [30], we can view the semantics of a k-dimensional boolean function as a binary string of length 2^k, constructed by concatenating output values for all possible inputs processed in lexicographic order. The semantics contained in Table 3.1 can therefore be written as *00011111*.

TABLE 3.1: Semantics of function $f(x1, x2, x3) = OR(x1, AND(x2, x3))$

index	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

3.2 Subtree contexts

A *context* is defined as a tree with a temporarily removed (arbitrary) subtree, marked with a # character, called the *insertion point* [30, 26]. Given a GP tree of n nodes, it is possible to build *n-1* nontrivial contexts from it by removing particular nodes (we discard the trivial case of removing the entire tree). For instance, the three-node tree `(or x y)` gives rise to two contexts: `(or # y)` and `(or x #)`. A sample context is depicted on Fig. 3.2.

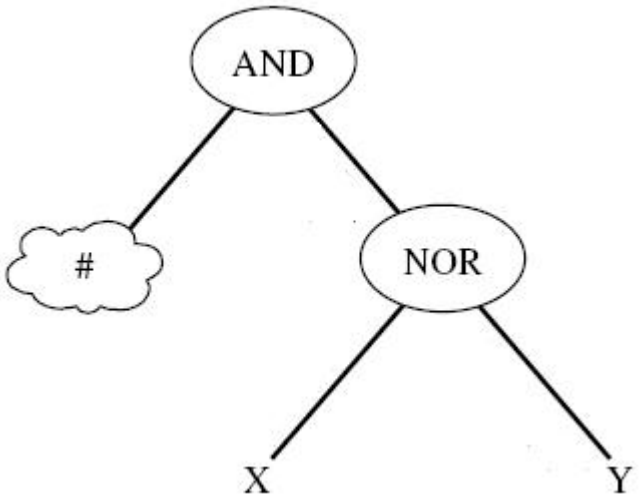

FIGURE 3.2: A sample context. # marks the insertion point. Source: [26]

3.3 Context semantics

The concept of semantics can be generalized to a context [26]. Some changes to the definition need to be made due to the fact that in a context, part of the tree is missing, and therefore the tree's response to an input usually cannot be calculated, as it depends on the missing subtree. However, there are cases when the entire context's output does not depend on the missing part. An example of such a context would be

`and(false #)`, which always returns *false*. It is known that Genetic Programming has inclinations towards the creation of such contexts [26].

A context is referred to as *fixed* for a particular fitness case (input object) if its output value is completely determined (*true* or *false*) and independent of what tree is appended at the insertion point. If a context is fixed for every possible combination of input values, it is called *fixed*.

For boolean functions, the semantics of a context is a word over the alphabet $\{0, 1, +, -\}$, of length 2^d, where d is the number of the function's arguments. The four elements of the alphabet correspond to 4 possible cases in the interaction between the output values of the missing fragment of the tree, and the output values of the entire tree, for a particular combination of inputs. The first case is the situation where the value of the context is *false* regardless of the missing subtree, e.g. `and(false #)`. The second case is a constant output value equal to *true*, e.g. `nand(false #)`. The third case is when the output of the tree is the same as the output of the missing subtree, e.g. `or(false #)`. The fourth and final case corresponds to the situation where the tree's output is the negation of the missing part's output, e.g. `nor(false #)`.

Computing the semantics of a context is more complex than computing subtree semantics. The latter is simply a function of the subtree root's operator and the values of its arguments, and it is completely independent of the subtree's position within the entire tree. Context semantics, on the other hand, is determined by three factors (see Fig. 3.3):

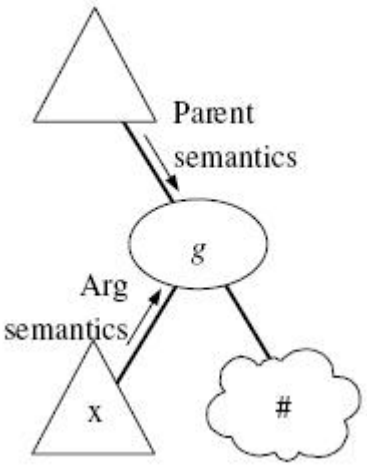

FIGURE 3.3: The factors which influence context semantics. Source: [26]

- the operator g immediately above the insertion point
- the semantics of the context obtained by removing the parent node
- the subtree semantics of any other arguments of the parent node's operator g.

TABLE 3.2: Semantics of functions *and, or*. Source: [26]

Parent semantics	Arg. semantics	and(x #)	or(x #)
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	1
+	0	0	+
+	1	+	1
-	0	1	-
-	1	-	0

There is one obvious exception - if the insertion point is the root of the entire tree, then there is no parent node and the context semantics is simply '+'.

In order to be able to compute the semantics of all contexts within a given tree, it is necessary to define a set of rules which indicate the value of the semantics for every possible combination of argument semantics and parent semantics. One set of rules must be defined for every primitive function present within the tree. If, for example, we were given the task of calculating all context semantics for the tree depicted by Fig. 3.1, the definitions of rules for the functions *or* and *and* would have to be provided (see Table 3.2).

Chapter 4

Potential fitness in Genetic Programming

4.1 Motivations

Most of evolutionary computation variants estimate an individual's fitness based on its actual behavior (phenotype). The fitness function usually does not explicitly consider the potential offspring of the evaluated individual. The rationale for this is at least twofold. Firstly, this is consistent with the biological evolution, which operates here and now, and, to our current knowledge, has no means of predicting the performance of individual's offspring. Secondly, also most of genotypic representations used in EC make it difficult to tell in advance if a specific individual has a chance for giving rise to a successful lineage. Having such an 'evolutionary foresight' could be profitable for the convergence of the evolutionary run. One can argue that the 'regular' fitness function already does this job - if it didn't promote the individuals that are likely to produce well-performing offspring, we would not observe any convergence at all. An individual performs well due to some fragments of its genetic code, so the evolutionary process promotes it in hope of re-using those portions in some of the individual's offspring. However, the code portions in question are not examined explicitly by the fitness function. The approach studied in this thesis investigates the fragments of individual's code in order to predict the chance of success for individual's potential offspring in a more precise way than the regular fitness function does. More precisely, it estimates the fitness of the best potential child of the individual, assuming that such a child is obtained from the evaluated individual by modifying it by means of a single-point mutation or a single-point crossover. In other terms, an individual is rewarded for its *potential* fitness rather than the actual fitness; hence the name of the approach. As it turns out, such a foresight is possible within the framework of genetic programming (GP) and boolean domains. Moreover, it may be done at a reasonable computational cost.

4.2 Learning boolean functions

Consider an n-dimensional boolean function $f(x_1, x_2, \ldots, x_n)$. There exist 2^n possible input combinations. Let us number the input combinations $1..2^n$, and let f_i denote the value of function f for the i-th input combination, i.e. $f_i = f(x_1^i, x_2^i, \ldots, x_n^i)$. The task for the learning agent may then be defined as follows: given the values f_i, a set of functional primitives and a set of terminals (representing the independent variables), construct a function $g(x_1, x_2, \ldots, x_n)$ such that the value:

$$p = |\{i : f_i = g_i \wedge 1 \leq i \leq 2^n\}| \tag{4.1}$$

is maximized. In other words, the function g should be a representation of function f whose values are the same as those of f for as many fitness cases (input combinations) as possible. Ideally, g should have the same values as f for *every* fitness case.

4.3 Potential fitness for boolean functions

Usually, when applying Genetic Programming to learning boolean functions, the value defined in Eq. 4.1 is not only the end criterion for the quality of the learning result (function g), but simultaneously it is the *intermediate* fitness function in every step of the evolution. The potential fitness approach is an alternative to such a methodology. Algorithm 1 shows an outline of the approach, whereas Algorithm 2 contains the general contract of the function *contextScore*.

Algorithm 1 Outline of the Potential Fitness approach for evaluating individuals. The parameter T represents the tree being evaluated.

```
treeFitness ← −2^n
for all contexts c ∈ contexts(T) do
  cFit ← contextScore(T,c)
  if cFit > treeFitness then
    treeFitness ← cFit
  end if
end for
return treeFitness
```

The method of operation of the Potential Fitness approach will be discussed in detail within this section.

In order for the *potential fitness* scheme to be applicable to a problem, two conditions must be met:

- the function to be learned is defined in the boolean domain, and
- the function's values for each fitness case are known *a priori*.

Algorithm 2 General contract of the *contextScore(T,c)* function. The notation *output(T,c,S,f)* represents the output of tree T for fitness case f with subtree S rooted at the insertion point of context c.

```
cF ← 0
for all fitness cases f do
  if ∀_{s1,s2} output(T,c,s1,f)=output(T,c,s2,f) then
    if output(T,c,*,f)=desired(f) then
      cF ← cF + 1
    else
      cF ← cF - 1
    end if
  end if
end for
return cF
```

The potential fitness function attempts to estimate the chance of the tree being modified (by means of crossover) so that its fitness is maximized. It does so by calculating certain information about every possible context of a tree. Each context corresponds to a situation where the root of the missing subtree (insertion point) is chosen as the crossover point. Therefore it is a convenient way to examine the possible impact of the crossover operation at that node on the entire tree.

The presented method makes use of an important distinction between two classes of elements within context semantics for boolean functions (see Section 3.3). It is the distinction between *fixed* elements (*0* and *1*) and *non-fixed* elements (*+* and *-*). The fixed part of the semantics indicates the fitness cases for which the performance of the context cannot be changed (improved or deteriorated). The non-fixed part of the semantics, on the contrary, indicates the fitness cases for which the performance of the context may still be changed by substituting an appropriate subtree in place of #.

The fitness function iterates over all contexts of the tree being evaluated. For every context, the algorithm keeps track of a special variable referred to as the context's *score*, with an initial value of 0. The semantics of the context is examined. For any fixed value of the semantics, the score is incremented if that value is consistent with the tree's desired output, and decremented otherwise. The non-fixed elements of the context semantics are ignored.

The scoring scheme rewards a context for hits and penalizes for misses. However, it does not care about the remaining fitness cases, i.e., the ones that could be correctly classified provided an appropriate tree would be substituted in place of #. Two contexts that are fixed to a different extent (e.g. one of them being fixed at 2 positions and the other one having 4 fixed positions), may still have the same score. Clearly, for n fitness cases, the worst possible score is $-n$ and the best possible score (ideal) is n.

The potential fitness of the entire tree is simply the maximum of scores of all its contexts. In other words, the potential fitness finds the best possible context with respect to evolvability, i.e., such a location in the tree that would result in the best fitness when substituted by an appropriate subtree. Note that the fitness value is not influenced by the number of maximum-score contexts within the tree.

As an example, consider a problem where the function to be learned, $f(x_1, x_2)$, is two dimensional and its values are defined as *1100*. Assume that a function $g = or(x_1, x_2)$ needs to be evaluated. In order to do this, 3 contexts need to be evaluated. The first one, #, can obviously be skipped. This leaves two valid contexts: $or(x_1, \#)$ and $or(\#, x_2)$. However, the function *or* is commutative, which means that only one of these two contexts needs to be considered, as the other one will yield the same score. We will calculate the score of the context $or(\#, x_2)$. The semantics of this context is *+1+1* (see Section 3.3 for instructions on calculating context semantics). As the non-fixed components are ignored during score calculation, only the second and the fourth elements of this semantics are relevant for score calculation. In the former case, context semantics is consistent with the tree's desired output, so the score is incremented and amounts momentarily to 1. However, context semantics is different from the tree's desired output for the last, fourth fitness case. This causes the score to be decreased and brings it back to its original value, i.e., 0. Thus, the overall score of this context is 0.

4.4 Calculating potential fitness

This section covers three methods of computing the potential fitness function. The first two methods are exact - they always return the maximum score of a context within the tree. They follow the pattern represented by Algorithm 1, because they iterate over all contexts of the tree and choose the maximum score. They differ only in the way they implement the function *contextScore*, i.e. Algorithm 2, which, as the name implies, is tasked with calculating the score of a single context. The third method is a randomized heuristic. It does not follow the schema represented by Algorithm 1, as it does not consider all contexts within a tree, but only a small subset of them.

4.4.1 Calculation using context semantics

This method (referred to as *Algorithm A*) is based on three simple steps. The first step is to compute the subtree semantics of the tree being evaluated - this means calculating the return value of every node within the tree. The next step involves the computation of the semantics of every context which may be obtained from the tree. This is a top-down process, starting at the root of the tree (see Section

3.3). This process makes use of the information gathered in step one (the subtree semantics). Finally, for every context, the semantics string is examined and the score value is calculated. The maximum score is memorized and returned as the result for the entire tree.

Algorithm 3 Computing the potential fitness of tree T using context semantics. Algorithm A's implementation of the *contextScore* function.

```
cFit ← 0
for i = 1 to 2^n do
  sem ← contextSemantics(T,c,i)
  if sem = 0 or sem = 1 then
    if sem = desired_i then
      cFit ← cFit + 1
    else
      cFit ← cFit - 1
    end if
  end if
end for
return cFit
```

Computational complexity

In step one, each node of the tree is examined exactly once, and for every node, all fitness cases must be enumerated in order to determine the node's output value for every one of them. Therefore the complexity of this part is $m * 2^n$, where m is the number of nodes within the tree and n is the dimension of the function being learned.

Phase two is similar in terms of complexity. It iterates through all nodes of the tree, and for each node it does a simple table lookup to determine the semantics for every fitness case (one of $0, 1, +, -$). The computational effort is thus again $m * 2^n$.

The final step once again iterates through all nodes and examines the semantics of the context whose insertion point corresponds to that node. In order to calculate the context's score, the semantics string (of length 2^n) must be fully checked.

From the above it may be concluded that the estimated total computational cost for an m-node tree is $3m * 2^n$

4.4.2 Exact calculation with Branch and Bound

Limits of *Algorithm A*

The second exact method was developed for two reasons. The first reason is extensibility. In order to use the first method (*Algorithm A*), the rules for calculating context semantics for every operator (function) within the primitive set must be

provided. That is, a table such as Table 3.2 must be defined for all the functions used. This would involve the necessity to modify the source code of the evaluating subsystem of the application every time a new function is added. The method presented in this section does not suffer from a similar problem.

The second reason is an attempt to decrease execution time, by designing a method which can benefit from the speedup resulting from packing individual boolean values (bits) into integer variables. This technique can be applied to calculating the subtree semantics of any tree. Since the values of any variables which are part of the problem are of boolean type, they only occupy one bit within an integer variable. Also, due to the fact that on most modern processors, bit operations operate on groups of bits (words) rather than single bits anyway, we can group many boolean values together in an integer variable whose length corresponds to the processor's word length. This way, the algorithm has to perform a lot less operations - if the word's length is k, only $\frac{1}{k}$ of the original workload has to be performed. Of course, if the total number of fitness cases is less than the word length, the gain in execution time is smaller.

However, this method cannot be applied to calculating context semantics. This is because the possible values of context semantics are no longer restricted to boolean values - there are 4 possibilities (0,1,+,-). Thus, the semantics of a context for a specific test case cannot be expressed by one bit (as it was the case with subtree semantics). Moreover, there is no straightforward way of representing the arbitrary rules of calculating semantics values (such as in Table 3.2) by means of the available bit operators (*and, or, xor, not*). Therefore, bit packing cannot be immediately applied to *Algorithm A*.

Definition

The method, referred to as *Algorithm B*, does not explicitly calculate context semantics. It is based on the observation that for the purposes of score calculation, we do not need the actual semantics string, but rather it is only required to identify those fitness cases for which the tree's output is independent of the missing subtree. This can be done in the following manner. First, the (subtree)semantics of every subtree in the tree are calculated and stored. Following that, every context is considered. The subtree located at the insertion point is replaced by an artificial node, referred to as *One*, which returns *1* for all fitness cases. The next step is to calculate the modified tree's output for every fitness case. However, we do not need to recompute the return values of every node within the tree. It is sufficient to update the values of the nodes on the path from the insertion point to the tree root, because the other values have not changed since they were last memorized. Therefore, the change induced by the artificial node needs to be propagated to the tree root. As an example,

consider the tree depicted by Fig. 4.1. Suppose the insertion point is node *10*. We would then replace node *10* with the artificial node *One*, and we would recalculate the values for nodes *7,3* and *1*. It should be emphasized that this time, once again all calculations are performed in the boolean domain, which means we can make use of bit grouping to obtain a significant speedup. Note that the computational effort associated with propagating the artificial node's values up the tree is proportional to the depth of the insertion point.

After the return value of the tree has been calculated, it is stored for use in the next phase of the algorithm. Then, this procedure is repeated, only with another artificial node, *Zero*, which returns zeroes for every fitness case, inserted in place of *One*. Once again the modified values are propagated to the root of the tree, and the tree's semantics are obtained.

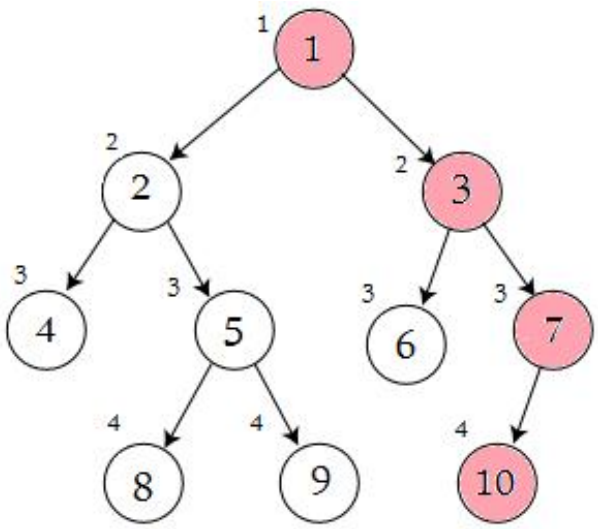

FIGURE 4.1: Updating the subtree semantics on the path to the root

Now that the semantics of the two versions of the tree (with *Zero* or *One* at the insertion point) have been calculated and stored, it is time to determine the context's score. This can be done in a manner which makes use of the bit grouping mechanism, which is key to preserving good execution speed. Assume that the length of the processor's word (natural group of bits used as operands for any arithmetic/logic operations) is d. On most modern computers, d is equal to 32 or 64. Thus, N boolean values (bits) can be stored in $\lceil \frac{N}{d} \rceil$ words. For the purpose of explaining this part of the algorithm, the following notation will be used:

- one_i: represents the i-th chunk (word) of the tree's responses obtained while the artificial node *One* was integrated into the tree
- $zero_i$: represents the i-th chunk (word) of the tree's responses obtained while the artificial node *Zero* was integrated into the tree

- $desired_i$: represents the i-th chunk (word) of the desired outputs of the tree
- *getNumBitsSet()* is a function which takes one word of length d as its sole argument and returns the number of bits in it which are equal to *1*. Of course, this function has to be implemented in an efficient way, i.e. the number of operations has to be significantly less than d. This can be achieved by storing the number of set bits for every possible word in memory. If the word length is too big, this information may not fit into the computer's physical memory, and the cost of calculating it may be too computationally expensive. Therefore, it is often beneficial to tabularize only a portion of the existing word space. As an example if the word length is *32*, it is sufficient to store the result of the function for every possible word of length *16*. Then, the computation of the function's value for a *32*-bit word requires only the access to two memory cells (for both *16*-bit halves) and their addition.
- N is the number of fitness cases, i.e. $N = 2^d$, where d is the dimension of the function f being learned
- d is the word length. For simplicity, it is assumed that N is a multiple of d, although the presented algorithm can be generalized to the case where this is not necessarily true.

The algorithm for computing the score of a context can be outlined in the following way:

Algorithm 4 Algorithm B's implementation of the *contextScore* function. Parameters T and c are respectively the tree and context under evaluation.

```
score ← 0
insertSubtree(T,c,One)
propagateToRoot(T,c)
one ← output(T,c,One)
insertSubtree(T,c,Zero)
propagateToRoot(T,c)
zero ← output(T,c,Zero)
for i = 1 to ⌈N/d⌉ do
  oneXzero ← one_i xor zero_i
  desiredXone ← desired_i xor one_i
  score ← score + d - getNumBitsSet(oneXzero or desiredXone)
  score ← score - getNumBitsSet(not(oneXzero) and desiredXone)
end for
return score
```

The first action is to set the *score* to *0*. Then, for every chunk of the solution, the *score* is updated accordingly. The variable *oneXzero* is the value of the `xor` operation performed on the chunks of vectors *one* and *zero*. Therefore, every position within

oneXzero with a value of *0* corresponds to a fitness case whose value does not depend on the missing subtree (the *xor* operator has a value of zero iff both its operands have the same values). Conversely, every bit in *oneXzero* with a value of *1* corresponds to a fitness case whose value *does* depend on the missing subtree.The variable *desiredXone* is the value of the `xor` operation performed on the chunks of vectors *one* and *desired*. Any bit of value *0* within this variable reflects the situation where the output value of the tree with artificial node *One* for the corresponding fitness case is the same as the desired value. A value of *1* means that the output and the desired value are different for this fitness case.

Consider a variable whose value is the bitwise `or` of *oneXzero* and *desiredXone*. Every set bit within this variable corresponds to a fitness case for which either the tree's output is different than the desired value, or the tree's output is not independent of the missing subtree. Therefore, if we denote the quantity of set bits as *q*, then *d-q* is the number of bits which do not fulfill any of the two aforementioned conditions - i.e. the number of fitness cases for which the context semantics are fixed and equal to the desired values. The *score* is incremented by *d-q*. In the next step, the *score* must be decremented by the number of fixed positions within the context semantics which *differ* from the desired values. This quantity can be easily calculated as the number of set bits within the result of the bitwise `and` of the negation of *oneXzero* and *desiredXone*.

Computational complexity

In contrast to *Algorithm A*, where the effort associated with calculating every context's score within the tree is the same, the cost of calculating a context's score using *Algorithm B* is proportional to the insertion point's depth within the tree. More formally, the cost, *c(x)*, may be expressed as follows:

$$c_B(x) = 2c_1 depth(x) n_{chunks} + c_2 n_{chunks} \tag{4.2}$$

where x is the insertion point, n_{chunks} is equal to $\lceil \frac{N}{d} \rceil$, c_1 is a constant representing the unit of time necessary to compute the output of one node for one chunk of input data, assuming that the values of all its arguments are known, and c_2 is an analogous constant representing the total cost of the operations performed inside the loop in Algorithm 4. The dominating component of this sum is the fragment $2 * c_1 * depth(x) * n_{chunks}$. The constant term *2* reflects the fact that the altered subtree's values must be propagated to the root twice (once for both of the artificial nodes *Zero* and *One*).

Consider a tree comprised of M nodes. We can thus express the mean effort of computing an average context's score as:

$$\bar{c_B} = \frac{2\sum_x (c_1 depth(x) n_{chunks} + c_2 n_{chunks})}{M} \approx 2c_1 n_{chunks} \frac{\sum_x depth(x)}{M} \tag{4.3}$$

The sum in the last term in Eq. 4.3 is simply the mean depth of all nodes within the tree, i.e.

$$\bar{h} = \frac{\sum_x depth(x)}{M} \tag{4.4}$$

The symbol $\bar{h}$ is used to denote mean node *depth* within the tree in order to avoid confusion with the length of the word d. The mean complexity can therefore be expressed as:

$$\bar{c_B} \approx 2c_1 n_{chunks} \bar{h} \tag{4.5}$$

Compare this to the cost of the same operation for *Algorithm A*:

$$\bar{c_A} = c_1^A N \tag{4.6}$$

where c_1^A is a constant representing the cost of computing a single value of semantics. The ratio of the mean costs for both algorithms is:

$$\rho = \frac{\bar{c_B}}{\bar{c_A}} = \frac{2c_1^B n_{chunks} \bar{h}}{c_1^A N} = \frac{2c_1^B \bar{h}}{c_1^A d} \tag{4.7}$$

If we assume that $c_1^B \approx c_1^A$, Eq. 4.7 becomes:

$$\rho \approx \frac{2\bar{h}}{d} \tag{4.8}$$

Therefore, *Algorithm B* will outperform *Algorithm A*, i.e. ρ will be less than 1, if $2\bar{h} < d$. A good intuition why this condition can often be satisfied in practice is a result from the field of discrete mathematics. It is known that the expected depth of a node in a randomly constructed binary tree of N nodes is $O(logN)$ (the proof can be found in [8]). While a GP individual(tree) subject to crossovers and mutations does not meet the formal assumptions which are necessary for this theorem to hold, the relationship $\bar{h} = O(logN)$ usually holds for GP primitive sets with a high (2 or greater) average branching factor. Moreover, the classic implementations of GP usually impose some constraints on the initial maximum depth of the constructed trees, and also on the maximum depth of any tree obtained through recombination or mutation. Combined with the fact that the word length, d, is up to 64 on modern computers, this shifts the performance ratio in favor of *Algorithm B*.

Applying Branch and Bound

Owing to the top-down nature of the computations, the performance of calculating the potential fitness can be enhanced even further by using the well-known *Branch and Bound* paradigm. This can be applied to both *Algorithm A* and *Algorithm B*.

At any node, apart from calculating the corresponding context's score, a quantity, denoted s^+, may be computed. This quantity represents the maximum gain in the score which may be attained by exploring the subtree rooted in the current node. The value of s^+ is simply the number of non-fixed positions within the semantics of this context. This is because once a position within the context semantics has become fixed, it will certainly not change by modifying any subtree whose root is a successor of the currently considered node. However, more non-fixed positions can become fixed. Ideally, all of the currently non-fixed nodes can become fixed in such a way that they correspond to the desired values.

Once s^+ has been computed, it is easy to identify a situation where there is no point in delving deeper into the subtree, because the maximum score of a single context will certainly not be improved. This is the case when the sum of the current context's score and s^+ is less than or equal to the maximum score of any context found so far. The cost of calculating s^+ is negligible when compared to the cost of calculating the context's score for both algorithms.

Although both *Algorithm A* and *Algorithm B* can benefit from this technique, it seems that the gain from performing the same cuts on a tree is greater for *Algorithm B*. Consider an example where the evaluated individual is the same as the one depicted by Fig. 4.1. We decide that the successors of node *3* do not need to be analyzed. For *Algorithm A*, the gain by doing so would simply be $\frac{3}{10}$ of the total workload, because the cost of evaluating every node is the same. For *Algorithm B*, however, the gain is $\frac{3+3+4}{4+4+4+3+3+3+3+2+2+1} = \frac{10}{29} \approx 0.34$. Similarly, if the cut were to be performed below node *7*, the respective gains of the two algorithms would be $\frac{1}{10}$ and $\frac{4}{29} \approx 0.13$.

4.4.3 Heuristic randomized calculation

There is one more advantage of *Algorithm B* over *Algorithm A*. Recall that context semantics must be calculated in a strictly top-down fashion, because the semantics of a context is a function of the semantics of its immediate parent context. *Algorithm B* is free from this constraint, and thus the score of every context can be computed independently and in any order. This gave rise to the randomized approach presented in this section.

This heuristic accepts a single input parameter which determines the maximum percentage of the full workload which may be performed (with respect to processing the entire tree). Following that, the algorithm picks random nodes from the evaluated tree with a probability distribution based on their depth within the tree, and calculates the score of every picked node's corresponding context. The number of nodes picked is such that the sum of the computational effort does not exceed the specified threshold. The maximum score among the evaluated context is returned

as the potential fitness of the tree. Of course, this method does not guarantee to find the actual maximum value within the tree.

This algorithm was designed to be a tradeoff between the quality of the obtained results and execution speed.

Chapter 5

Experimental results

5.1 Experimental setup

The potential fitness function was implemented as an extension to the *Evolutionary Computations in Java (ECJ)* framework. For all experiments, the standard parameter file *koza.params* found within *ECJ* was used. This involves the following settings:

- no mutation, only crossover and reproduction
- crossover probability *0.9*
- reproduction probability *0.1*
- tournament selection with tournament size *7*
- maximum tree depth for crossover operations: *17*
- uniform probability distribution for selection of crossover point
- every individual is a single tree (no Automatically Defined Functions)
- the elements of the primitive set are untyped, i.e. all nodes return a result of the same type
- population size: *1024* individuals
- number of generations: *100*
- during construction of a random tree, non-terminals are picked with a probability of *0.9* and terminals with a probability of *0.1*
- the *Ramped Half and Half* method is used for creating the initial population with a maximum depth limit randomly chosen from the range *[2; 6]*.

5.1.1 Benchmark problems

A total of 7 benchmark problems were used in this study. They can be classified into three problem groups, two of which are well-known and widely used as testbeds for learning boolean functions.

Odd Parity

The n-bit Odd Parity function $f_{op}(x_1, x_2, \ldots, x_n)$ assigns to every one of the possible 2^n input combinations the sum of each argument's value modulo 2, i.e. $f_{op}(x_1, x_2, \ldots, x_n) = (\sum_i x_i)$ *mod 2*. Therefore, the function returns *one* if the number of set bits is odd, and *0* if it is even. For the purpose of the experiment, the following primitive set was used:

- terminals: $x_1, x_2, \ldots, x_n$ representing the input variables
- non-terminals: two-argument functions *and(x,y), or(x,y), nand(x,y), nor(x,y)*

Three problems from this category were used in the study: *4-,5-,* and *6-Odd Parity*. The category is marked as k-OP on plots.

Multiplexer

The n-bit Multiplexer function's input can be classified into 2 groups. The first group corresponds to *information inputs*. The second group contains *address inputs*. The number of information inputs relates to the number of address inputs in the following way:

$$n_A = log_2 n_I \tag{5.1}$$

For any bit string of length n, the first n_I bits represent the values of their respective information inputs. The next n_A bits represent the number of the information input whose value should be returned as the function's output value. As an example, consider the *(4+2)*-bit Multiplexer function and a possible input *010111*. The values of information inputs *0,1,2* and *3* are respectively *0,1,0* and *1*. The address bits, *11*, point to input #3. Therefore, the value of the *6*-bit Multiplexer function for this input string would be *1*. The following primitive set was used:

- terminals: $x_1, x_2, \ldots, x_n$ representing the information inputs, and $a_1, \ldots, a_m$ representing the address inputs
- non-terminals: two-argument functions *and(x,y), or(x,y)*, one-argument function *not(x)*, three-argument function *if(x,y,z)*

The *if(x,y,z)* function returns the value of y if x is true; otherwise, it returns the value of z. It should be noted that in order to retain the ability to perform calculations

on groups of bits, or *words*, this function was expressed in terms of simpler boolean functions *and, or, not*:

$$if(x, y, z) = or(and(x, y), and(not(x), z)) \tag{5.2}$$

The functions *6*-Multiplexer and *11*-Multiplexer were used for testing. The category is marked as k-MUX on plots.

Comparator

The n-bit Comparator function, $n = 2i, i \in N$, treats the input string as two binary integers, each of length *n/2*. The return value is *1* if the number encoded by the first *n/2* bits is greater then the second number. Otherwise, the return value is *0*. The following primitive set was used:

- terminals: $x_1, x_2, \ldots, x_n$ representing the input variables
- non-terminals: two-argument functions *and(x,y), or(x,y), nand(x,y), nor(x,y)*

Two problems from this category were used in the study: *4*- and *6-Comparator*. The category is marked as k-CMP on plots.

5.1.2 Examined algorithm variations

In order to provide adequate material for a valid performance evaluation, a spectrum of algorithm variations was analyzed. The following basic algorithm types were used:

- standard Genetic Programming (GP) - the canonical form of boolean function learning, with a fitness function as described in Section 4.2.
- Genetic Programming with Subtrees (GPS) - this variation is similar to standard GP in that it uses the same fitness function, however, instead of returning the number of correctly processed instances for the tree root, it returns the maximum number of hits calculated over the set of all subtrees. Therefore, an individual's evaluation calculated by the GPS method is not less than its GP counterpart.
- Potential Fitness, exact calculation (PF) - as described in Section 4.4.2.
- Potential Fitness, randomized approach (rPF) - as described in Section 4.4.3.

In addition, each one of the base types listed above was examined in two configurations. The difference between them was the presence or absence of *lexicographic parsimony pressure*[25]. If parsimony pressure is enabled, the fitness function is modified in such a way that in case of equality between two individuals, the decisive criterion is tree size - the individual with fewer nodes is considered superior. In case of a draw on both criteria, a random individual is designated as the winner.

5.2 Results

5.2.1 Overview

In Section 5.2.2, the algorithms listed in 5.1.2 are examined with respect to their success rate and mean quality of the best individual per generation.

Section 5.2.3 deals with the comparison of running times of all the aforementioned algorithms.

In Section 5.2.4, the impact of using the Branch and Bound algorithm with Potential Fitness calculation is examined. In particular, the percent of total workload performed by the algorithm is presented on a per-problem basis. The base algorithm used in this part of the study is *Algorithm B* described in Section 4.4.2.

Finally, Section 5.2.5 outlines the details of the distribution of the Potential Fitness function across a population of individuals. Averaged histograms of Potential fitness are presented for each considered benchmark problem. Also, spectrograms demonstrating the relationship between relative potential fitness and relative depth of a node within the tree are shown.

5.2.2 Solution quality

In this section, the performance of the Potential Fitness approach will be characterized in terms of success rate (frequency of finding the optimal individual), mean generation in which the ideal individual was found (calculated only over those runs that found the ideal), mean hit rate of the best-of-run individual (calculated over both ideals and non-ideals), and mean size (number of nodes) of the best-of-run individual. The results have been calculated based on 400 independent evolutionary runs. Two algorithms which employ Potential Fitness (PF and rPF) along with two standard algorithms (GP and GPS) are compared (see Section 5.1.2 for a description of the algorithms). The GPS method is used in order to make the comparison between GP and PF more fair, because by definition standard GP considers far less solutions than PF. GPS makes up for this difference by considering all subtrees of the given individual when calculating the fitness.

The results indicate that in most cases, the deterministic Potential Fitness approach outperforms both GP and GPS in a statistically significant manner in terms of average hit rate. This seems to be true regardless of whether parsimony pressure is enabled or not. Also, for some cases, PF arrives at optimal solutions earlier than the other algorithms.

The main area where PF is clearly superior, though, is the success rate. For most of the more difficult problems (as *5-,6-Odd Parity*, *11-Multiplexer*), the success rate is effectively doubled. For one problem (*6-Comparator*), it is nearly tripled.

On the other hand, GPS does only slightly better than GP. This clearly indicates

that the superiority of PF should be attributed to the concept of score, and not to the fact that all (strictly: almost all) tree nodes are tested against the desired values.

The effect of enabling lexicographic parsimony pressure on the performance of the algorithms is twofold. Apart from reducing the average tree size and thus shortening the average run time, parsimony pressure usually increases the success rate and the best-of-run's hit rate. In particular, all the methods become able to occasionally find the optimal solution for the *6-Odd Parity* problem. PF is still superior in terms of success rate and best-of-run hit rate, though on average it benefits less from parsimony pressure than GP and GPS: its trees are smaller only about 35% than those evolved without parsimony pressure, whereas GP's and GPS's are reduced by 50% and 60%, respectively. As a result, PF trees are now approximately 60% larger than those produced by the other methods. This may be a side-effect of the potential fitness function that promotes individuals which would get worse evaluation from the regular fitness function and lose tournaments with equally-well performing yet smaller trees.

As for the randomized version of the PF algorithm, rPF, its performance is disappointing. This approach was designed as a tradeoff between execution speed and quality of obtained solutions, however the results suggest that using rPF is not worth the computational overhead, since the gain in solution is, if any, marginal. In fact, in some cases rPF actually performs worse than standard GP. Only in three cases was it able to produce results which are significantly better than those obtained with GP: twice for the *6-Comparator* problem (with parsimony pressure enabled/disabled) and once for the *6-Odd Parity* problem (with parsimony pressure disabled).

Figures 5.1 through 5.7 show the dynamics of the mean fitness function. In most cases, PF shows faster convergence from the very beginning of the evolutionary run. Only for Mux-11 (see Fig. 5.1), PF initially lags behind GP, but around the 35th generation starts to overtake it. Of course, for a very easy problem like Mux-6 (Fig. 5.2), all methods almost always converge quickly to the ideal solution and their comparison is inconclusive.

Table 5.1: Success rate and mean generation number when ideal individual is found, averaged over 400 evolutionary runs - parsimony pressure disabled

	Success rate				Ideal found in generation			
	GP	GPS	PF	rPF	GP	GPS	PF	rPF
Odd-4-parity	0.755	0.755	0.918	0.780	33.1	33.1	23.8	29.03
Odd-5-parity	0.048	0.055	0.220	0.010	74.3	67.1	69.9	92.0
Odd-6-parity	0.000	0.000	0.000	0.000	—	—	—	—
Mux-6	0.995	0.998	1.000	1.000	9.2	11.4	11.1	10.89
Mux-11	0.283	0.145	0.530	0.230	61.1	68.4	63.0	69.7
Cmp-4	0.990	1.000	1.000	1.000	7.6	8.6	6.2	7.2
Cmp-6	0.390	0.178	0.740	0.470	53.8	57.6	44.3	44.76

Table 5.2: Hit rate and tree size of best-of-run individual, averaged over 400 evolutionary runs - parsimony pressure disabled.Statistically significant differences in hit rate marked in bold (t-test, significance level 0.01).

	Hit rate				Tree size			
	GP	GPS	PF	rPF	GP	GPS	PF	rPF
Odd-4-parity	15.67	15.69	**15.90**	15.73	243.8	260.8	242.7	245.2
Odd-5-parity	28.39	28.49	**30.05**	28.66	402.8	416.4	452.2	406.1
Odd-6-parity	50.43	50.88	**53.45**	**51.28**	455.8	461.8	523.3	471.8
Mux-6	63.99	63.99	64.00	64.00	49.8	107.7	83.8	73.8
Mux-11	1950.87	1919.51	**1996.8**	1952.1	275.2	324.2	357.4	307.4
Cmp-4	15.99	16.00	16.00	16.00	67.1	87.7	69.6	72.43
Cmp-6	62.97	62.46	**63.68**	**63.28**	230.1	245.2	256.9	236.1

Table 5.3: Success rate and mean generation number when ideal individual is found, averaged over 400 evolutionary runs - parsimony pressure enabled

	Success rate				Ideal found in generation			
	GP	GPS	PF	rPF	GP	GPS	PF	rPF
Odd-4-parity	0.830	0.878	0.928	0.87	26.4	27.2	23.1	25.1
Odd-5-parity	0.103	0.140	0.295	0.14	69.9	70.3	73.1	74.6
Odd-6-parity	0.005	0.010	0.018	0.005	71.0	72.0	85.4	82.0
Mux-6	0.998	0.998	1.000	1.000	8.1	7.5	10.5	10.0
Mux-11	0.328	0.278	0.535	0.340	54.0	48.8	61.3	66.3
Cmp-4	0.910	0.843	0.998	0.931	7.2	7.3	6.3	7.1
Cmp-6	0.250	0.153	0.738	0.480	52.7	52.3	43.9	48.1

TABLE 5.4: Hit rate and tree size of best-of-run individual, averaged over 400 evolutionary runs - parsimony pressure enabled.Statistically significant differences in hit rate marked in bold (t-test, significance level 0.01).

	Hit rate				Tree size			
	GP	GPS	PF	rPF	GP	GPS	PF	rPF
Odd-4-parity	15.73	15.82	**15.91**	15.81	117.6	116.3	157.8	132.17
Odd-5-parity	28.99	29.32	**30.37**	28.78	174.0	179.4	254.1	179.5
Odd-6-parity	51.70	52.83	**54.45**	52.24	239.0	234.5	338.4	248.0
Mux-6	63.98	63.99	64.00	64.00	26.7	33.2	60.6	48.5
Mux-11	1950.05	1922.34	**1988.0**	1940.8	132.7	115.1	217.9	168.3
Cmp-4	15.91	15.82	**15.99**	15.94	45.3	43.5	53.3	51.28
Cmp-6	61.95	61.12	**63.56**	**62.92**	76.2	64.6	137.8	108.0

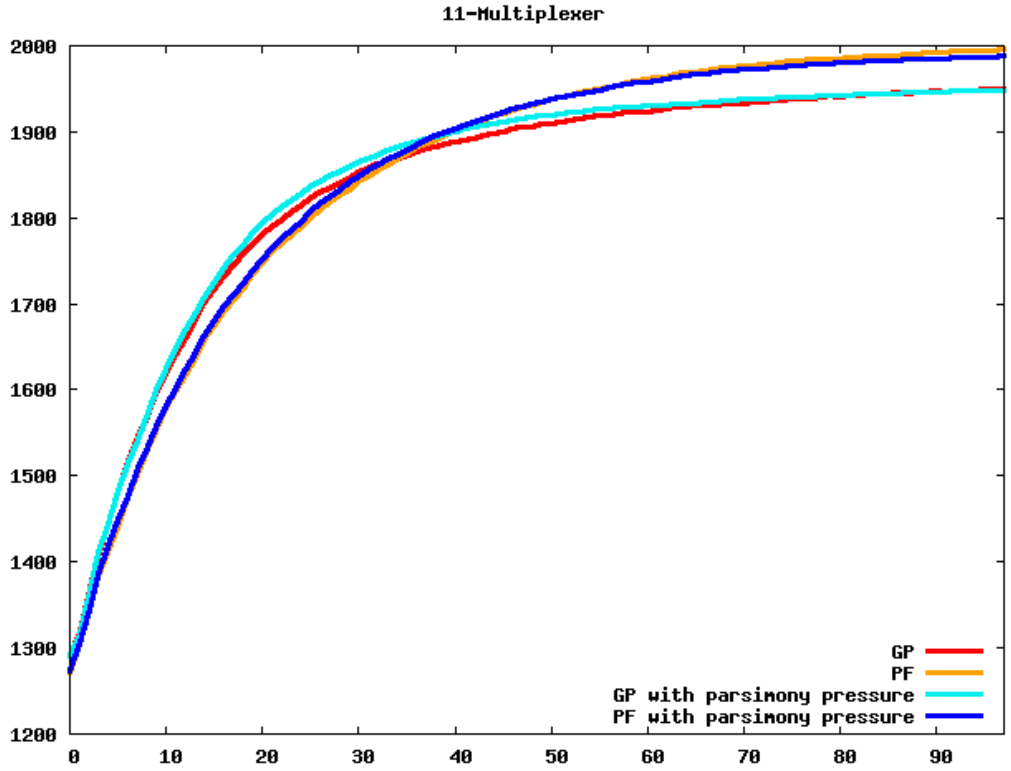

FIGURE 5.1: 11-Multiplexer. Average fitness as a function of generation number.

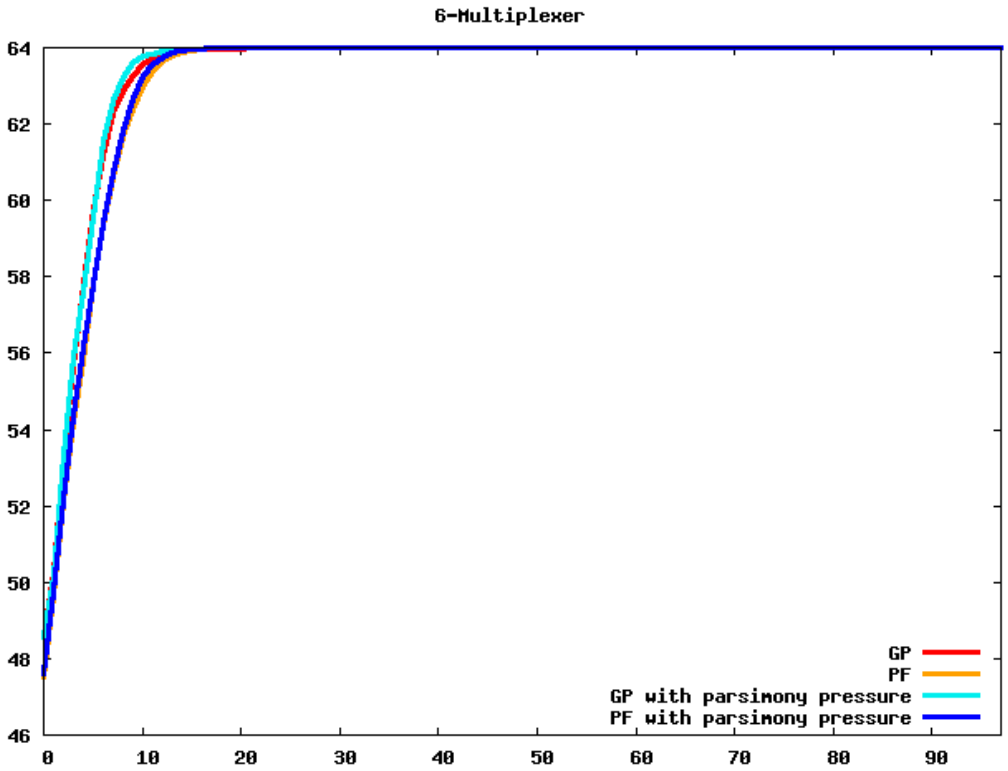

FIGURE 5.2: 6-Multiplexer. Average fitness as a function of generation number.

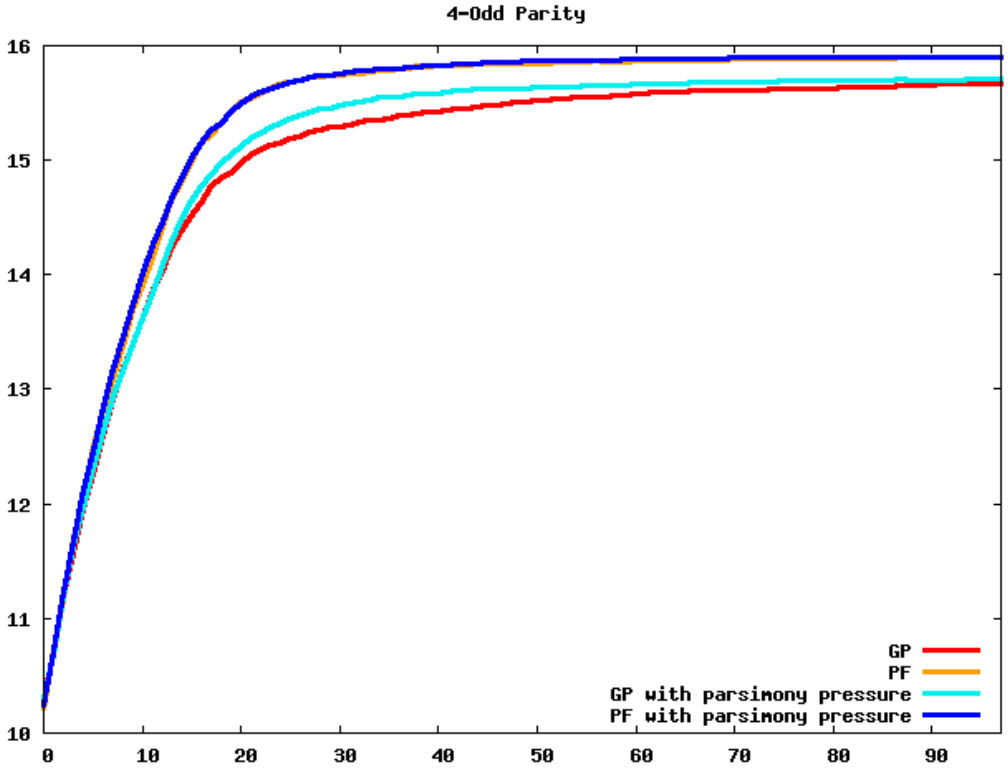

FIGURE 5.3: 4-Odd Parity. Average fitness as a function of generation number.

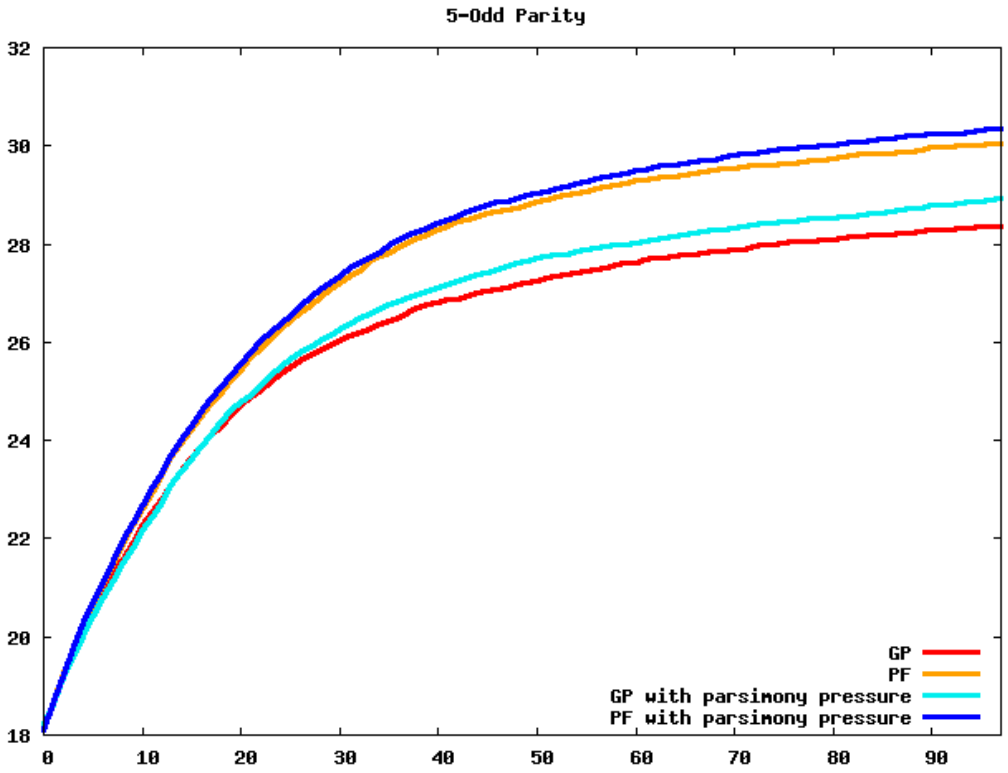

FIGURE 5.4: 5-Odd Parity. Average fitness as a function of generation number.

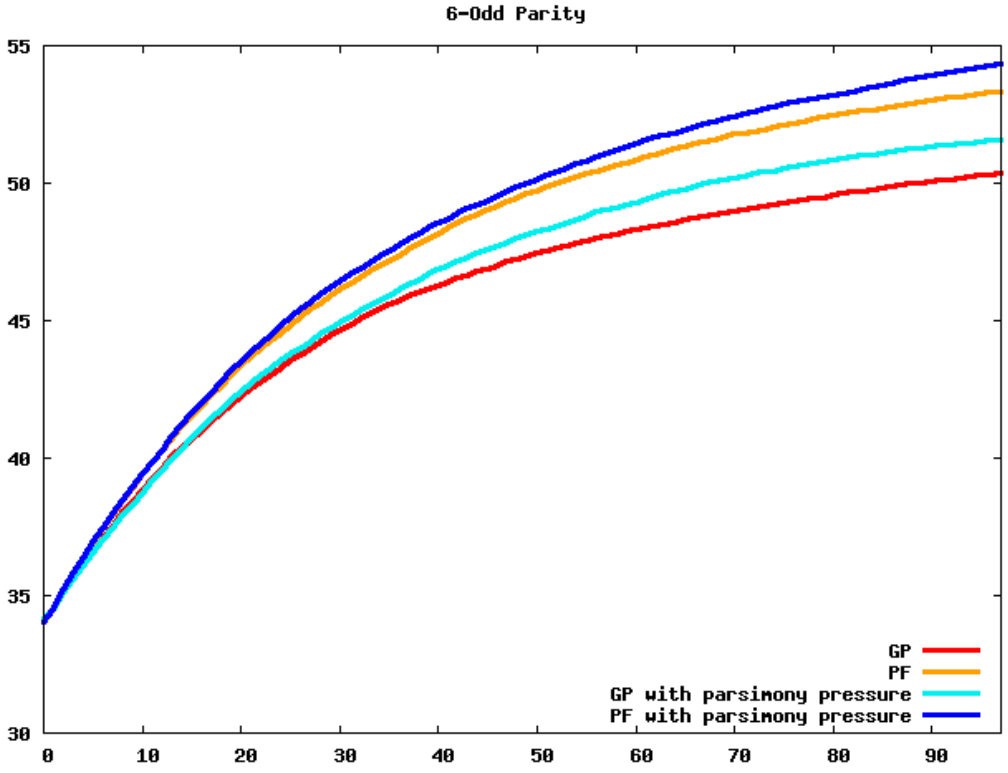

FIGURE 5.5: 6-Odd Parity. Average fitness as a function of generation number.

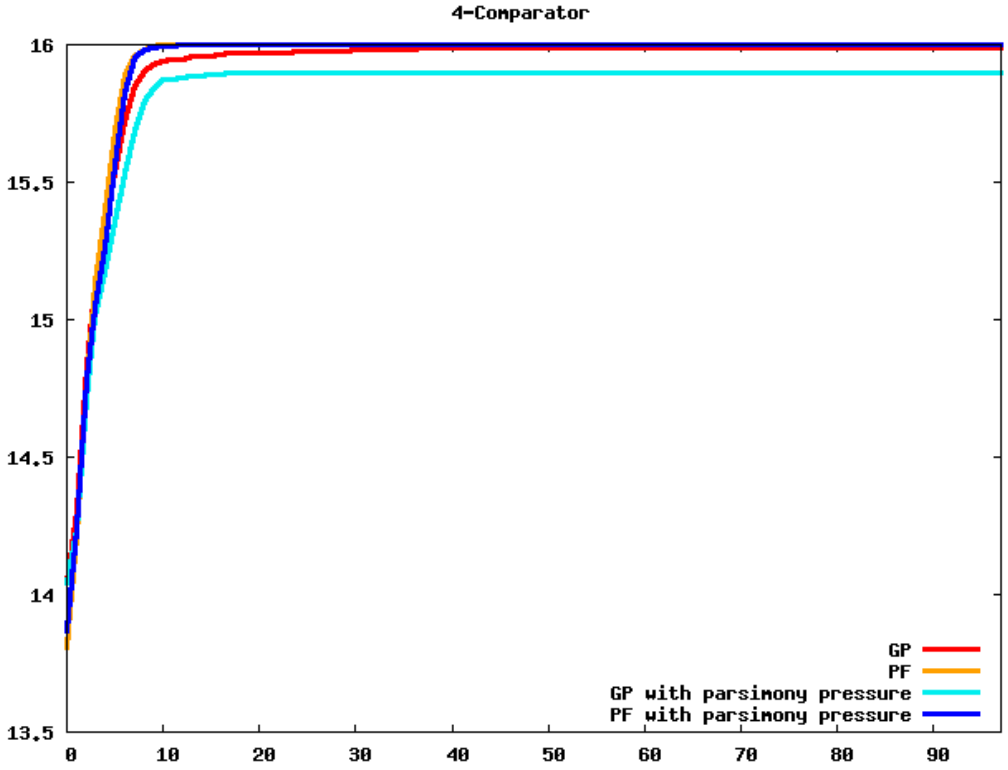

FIGURE 5.6: 4-Comparator. Average fitness as a function of generation number.

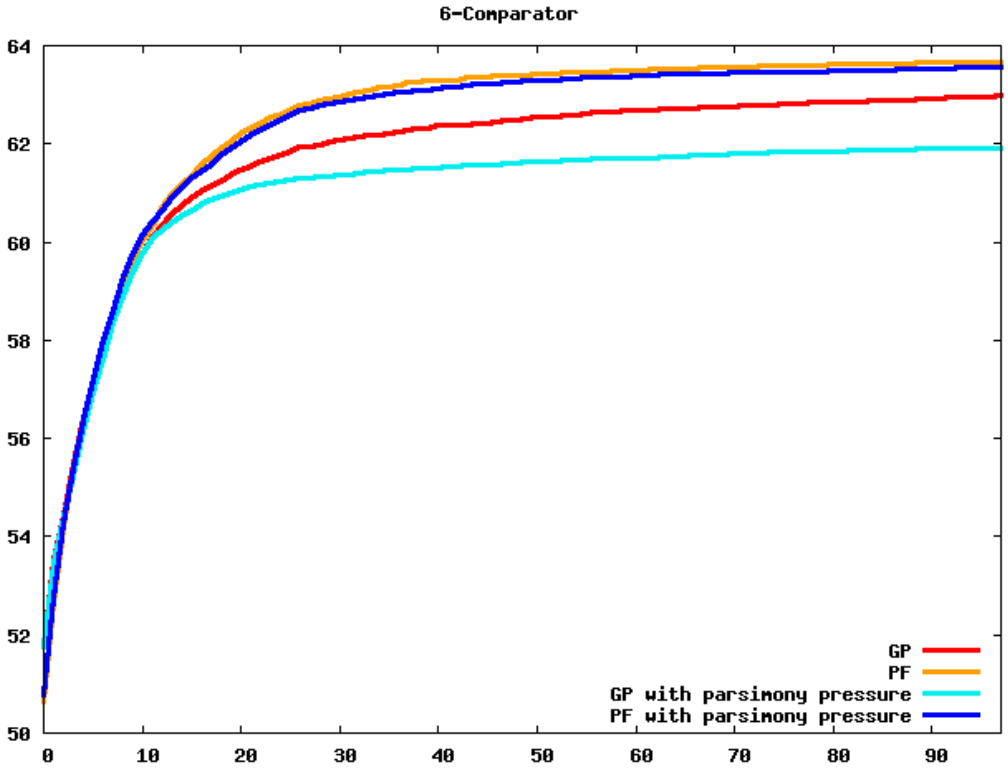

FIGURE 5.7: 6-Comparator. Average fitness as a function of generation number.

5.2.3 Running times

This section presents the running times of the various algorithms used for calculation of potential fitness, which were described in Section 4.4. Also, the running times of the standard GP evaluating algorithm are provided to enable a comparison.

Each algorithm is examined in two configurations - with and without lexicographic parsimony pressure. Two types of timing information are collected - the average evaluation time and the average total time of the evolutionary run.

In particular, the following algorithms are used for the comparison:

- standard GP evaluation with bit packing, denoted *GP*
- *Algorithm A* with Branch and Bound (see Section 4.4.1), denoted *A*
- *Algorithm B* with Branch and Bound (see Section 4.4.2), denoted *B*
- randomized algorithm for calculating potential fitness (see 4.4.3), denoted *rPF*.

When parsimony pressure is enabled, the relative total running times averaged across all 7 problems for algorithms *A,B* and *rPF* are, respectively, 6.04, 3.01 and 1.62, where the value 1.00 represents the total running time of the evolutionary run when using the standard GP algorithm. This means that, just as it was expected, Algorithm B performs significantly better than Algorithm A for most problems. The gain in execution speed becomes more apparent as the dimension of the problem increases - the most spectacular example is the *11-Multiplexer* problem, where Algorithm B outperforms Algorithm A in a nearly 5 to 1 ratio, in terms of mean evaluation time. For the smaller and simpler problems, Algorithm A seems to have the upper hand, although only by a small margin. Also, the randomized heuristic for calculating potential fitness, *rPF*, needs considerably less computation time than both exact methods. However, as it was shown in Section 5.2.2, the small gains in solution quality do not justify the computational overhead of 60% over standard GP. This suggests that it is hard to find a randomized heuristic which would produce solutions of a quality similar to the deterministic algorithm, while having significantly lower running times.

As for the case where parsimony pressure is disabled, the respective average relative total running times are 4.23, 2.49 and 1.30. Once again *Algorithm A* is inferior to *Algorithm B*. It is interesting that the differences in running times between GP and the two exact methods are smaller than it was the case when parsimony pressure was enabled. This can probably be attributed to the better performance of Branch and Bound on the larger trees produced by the evolutionary process when there is no pressure to reduce tree size. The performance of Branch and Bound is examined in detail in section 5.2.4.

There is one more interesting phenomenon which occurs in the two simple problems - *4-Odd Parity* and *4-Comparator*. For these two problems, the potential-fitness based algorithms actually have average running times which are smaller than those of standard GP. This is probably the result of the fact that for 4-bit problems, the differences in evaluation times are still small enough that they can be compensated by the superior success rate of the potential fitness approach - the ideal individual is found considerably earlier and the evolutionary process can be ended without running through a full 100 generations.

TABLE 5.5: Mean evaluation and total times, averaged over 100 evolutionary runs - parsimony pressure enabled

	Evaluation time				Total time			
	GP	A	B	rPF	GP	A	B	rPF
Odd-4-parity	2.3	7.1	8.1	4.1	10.6	21.4	22.3	13.5
Odd-5-parity	3.9	15.2	15.89	8.4	18.2	37.6	38.1	24.3
Odd-6-parity	6.5	35.1	26.0	11.8	26.6	63.6	54.9	33.12
Mux-6	0.69	9.5	6.1	3.7	2.6	15.2	12.5	6.1
Mux-11	22.5	746.0	161.6	53.1	30.1	766.0	177.6	63.1
Cmp-4	1.0	2.7	2.5	1.6	4.4	6.5	6.7	5.2
Cmp-6	1.8	12.5	8.8	5.6	7.1	21.6	18.0	13.2

TABLE 5.6: Mean evaluation and total times, averaged over 100 evolutionary runs - parsimony pressure disabled

	Evaluation time				Total time			
	GP	A	B	rPF	GP	A	B	rPF
Odd-4-parity	3.0	2.6	3.1	3.1	15.0	9.8	12.3	14.2
Odd-5-parity	8.1	16.4	16.9	15.5	42.3	51.9	52.6	51.2
Odd-6-parity	9.1	35.7	27.5	15.9	39.8	74.0	65.9	47.7
Mux-6	0.13	2.6	1.9	0.53	0.51	4.3	4.2	1.1
Mux-11	38.3	804.2	166.2	69.2	53.06	826.3	188.3	85.3
Cmp-4	0.08	0.16	0.21	0.23	0.46	0.42	0.5	0.48
Cmp-6	4.8	9.2	7.3	7.6	22.3	21.9	20.4	22.1

5.2.4 Performance of Branch and Bound

Percentage of performed workload and tree growth dynamics

In this section, the performance results of the Branch and Bound algorithm are presented. The plots show the average percent of the total workload done by the algorithm (100% represents the cost of processing the entire tree) as a function of generation number. Also, the dynamics of average tree size changes are shown. The

results are averaged over 100 independent evolutionary runs of 1024 individuals, 100 generations each.

Except the two easy problems (*4-CMP* and *6-MUX*), a pattern can be discerned regarding the behaviour of *BnB*. In the cases where parsimony pressure is disabled, the performed workload percentage drops almost exponentially with increasing generation number, reaching a minimum of 10% at generation *100*. It is interesting that in almost all cases, the curve which illustrates the mean tree size also has a similar shape. For the first 20-30 generations, the growth is faster than linear (with respect to generation number). Then, the growth rate decreases and becomes sub-linear. This behavior causes the curve to resemble an $x^{\frac{1}{2}}$ plot.

As for the situation where parsimony pressure is enabled, the overall efficiency of the *BnB* algorithm deteriorates a little, with an optimum usually located between 30% and 40% of the total workload. As the workload is relative, this decrease in performance cannot be attributed directly to the fact that there are simply fewer nodes within the tree, since the evolutionary process now favors smaller trees. It is rather a result of more complex interactions within the population caused by the modified fitness function (which includes tree size). It seems that two groups of problems can be isolated based on the shape of their respective workload and tree size curves. The first group includes the problems *11-Multiplexer, 4-Odd Parity,* and *6-Comparator*. The distinguishing feature of this group is that in each case, a threshold generation number exists which is simultaneously the point where the tree size achieves its maximum, and the workload percentage has its minimum (e.g. Fig. 5.17). After that point, a sharp drop in tree size and a significant, yet weaker increase of workload percentage occur. The two curves converge towards each other. This provides some insight about the mechanics of evolving the solutions. The sharp drop in tree size indicates that at a certain point, it becomes difficult for the evolutionary process to find better individuals in terms of congruence with the function to be learned, so instead it focuses on decreasing the tree size. This is particularly visible for *4-Odd Parity* and *6-Comparator*, for which the overall increase in solution quality between generation *20* and *100* is very small (see Section 5.2.2). It should be noted, however, that the drop in performance of *BnB* is not nearly as severe as that of the tree size - the most spectacular case is when tree size drops from 100% to 50%, while the workload percentage increases from 30% to 40%.

The second group, represented by the two *Odd Parity* problems of higher order (*5*- and *6*-bit), seems more resistant to the influence of parsimony pressure. Although the workload percentage is still significantly higher that in the parsimony pressure-free cases (25% vs. 10%), there is virtually no decrease in performance. Once the workload percentage reaches ca. 30% around generation *40*, this level is maintained through the remainder of the evolutionary process. Also, the drops in tree size are much less dramatic, reaching about 75% as a minimum value.

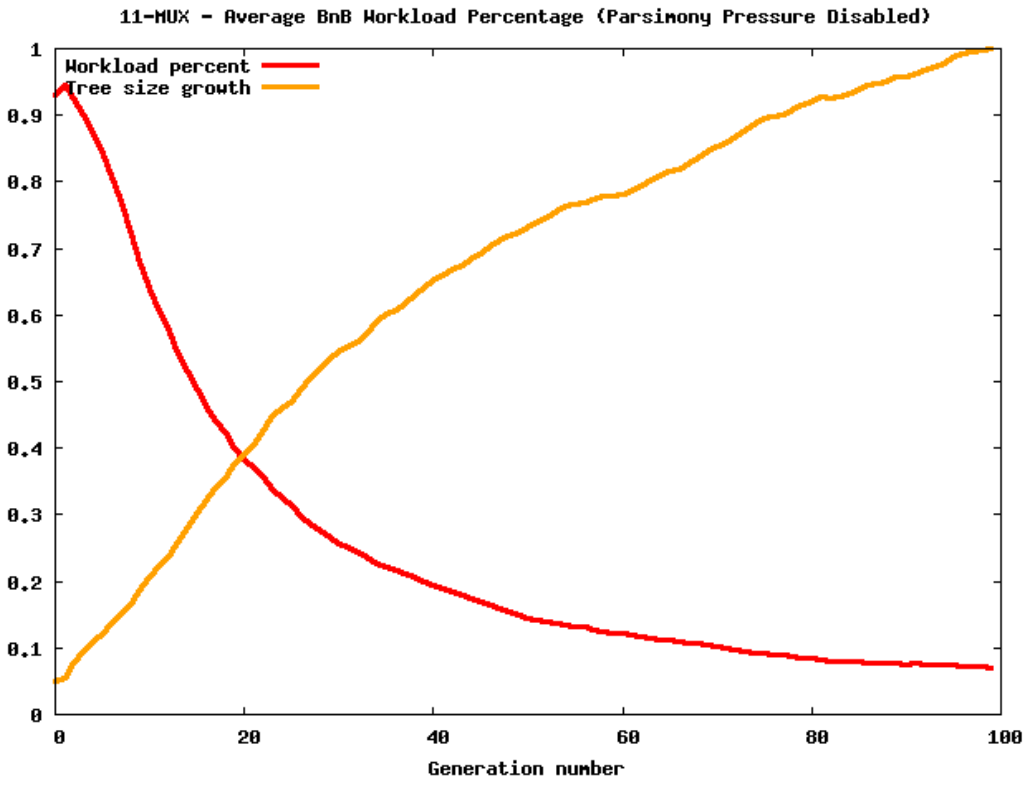

FIGURE 5.8: 11-Multiplexer without parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

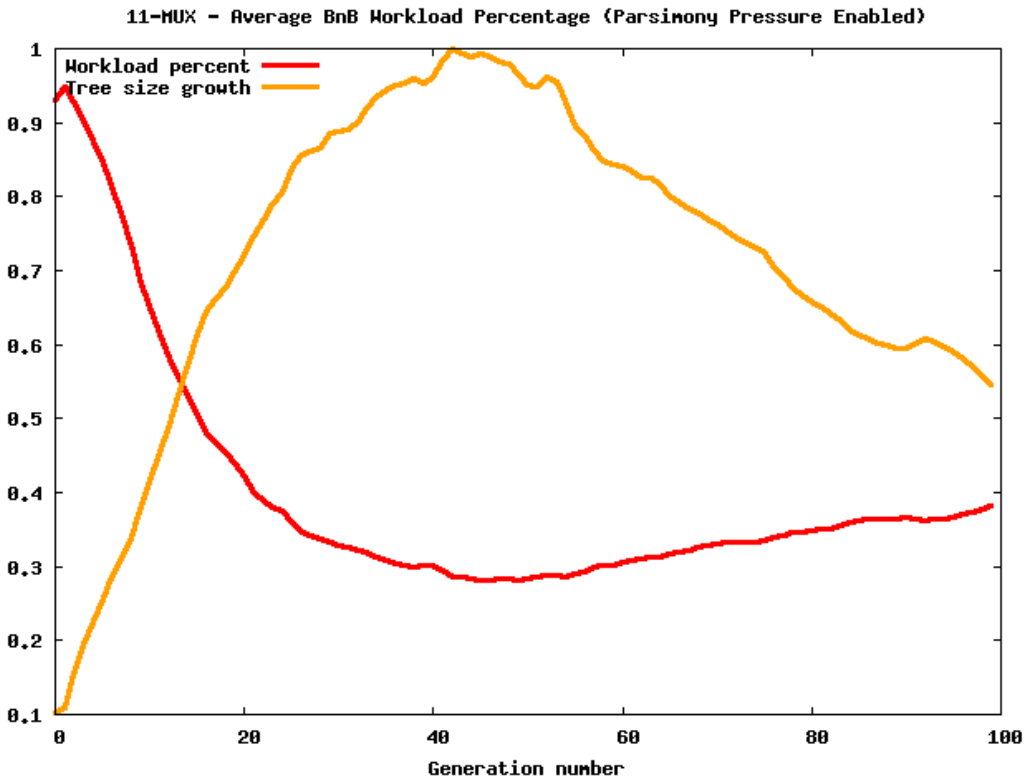

FIGURE 5.9: 11-Multiplexer with parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

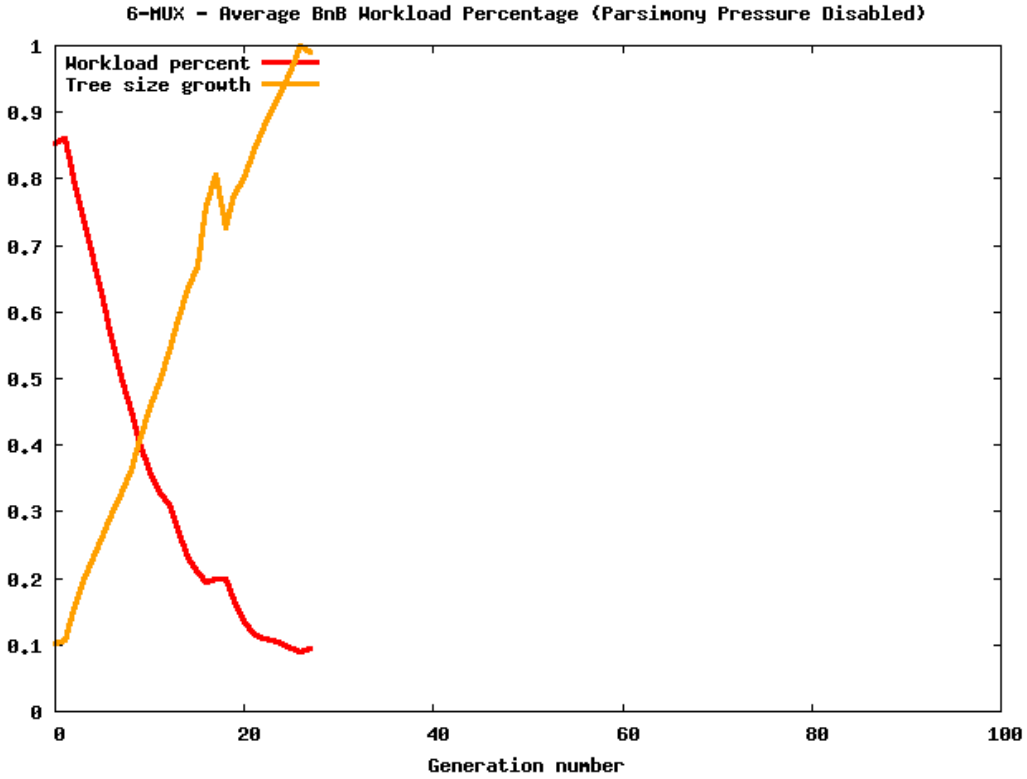

FIGURE 5.10: 6-Multiplexer without parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

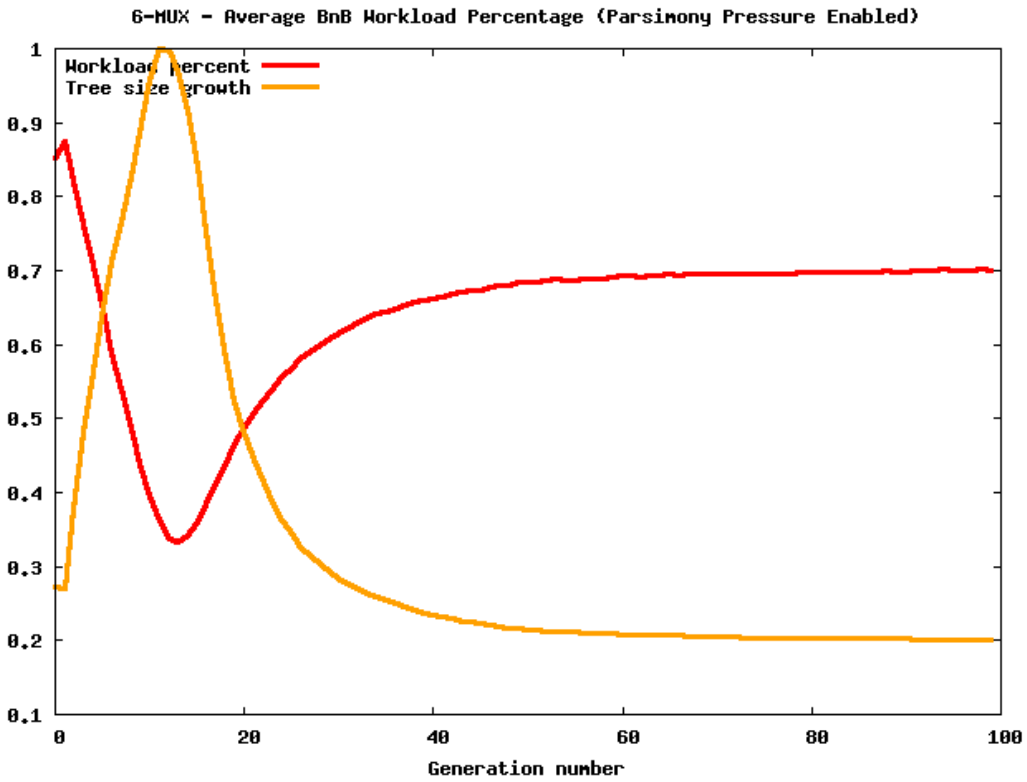

FIGURE 5.11: 6-Multiplexer with parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

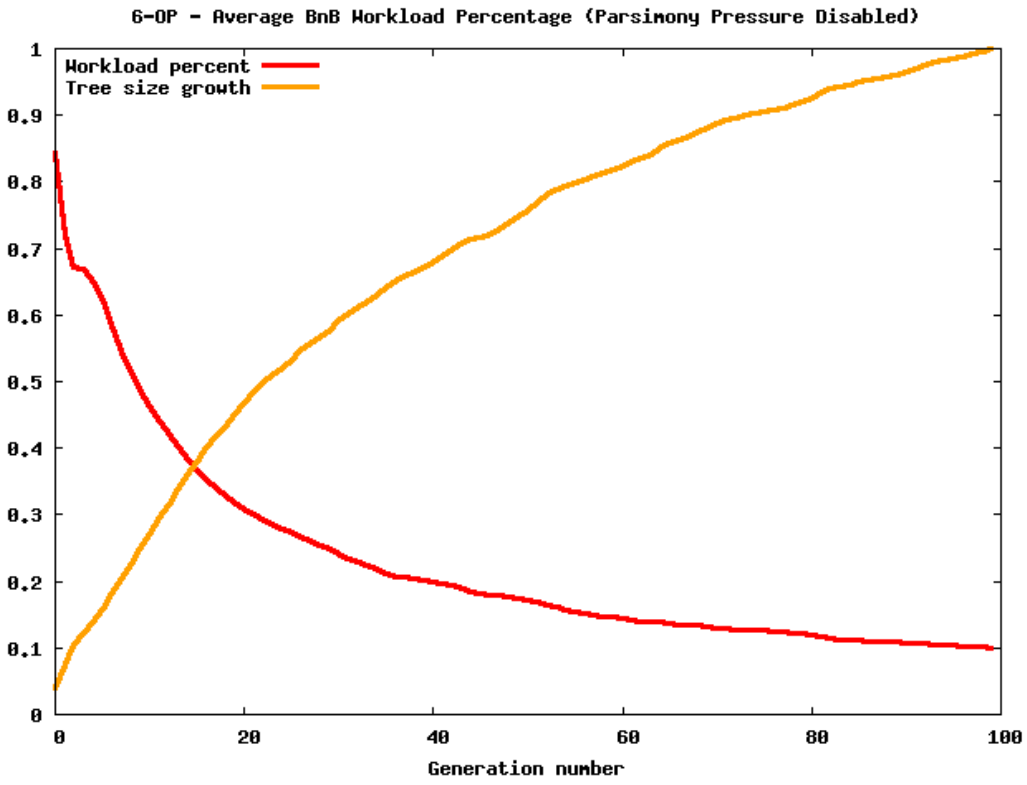

FIGURE 5.12: 6-Odd Parity without parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

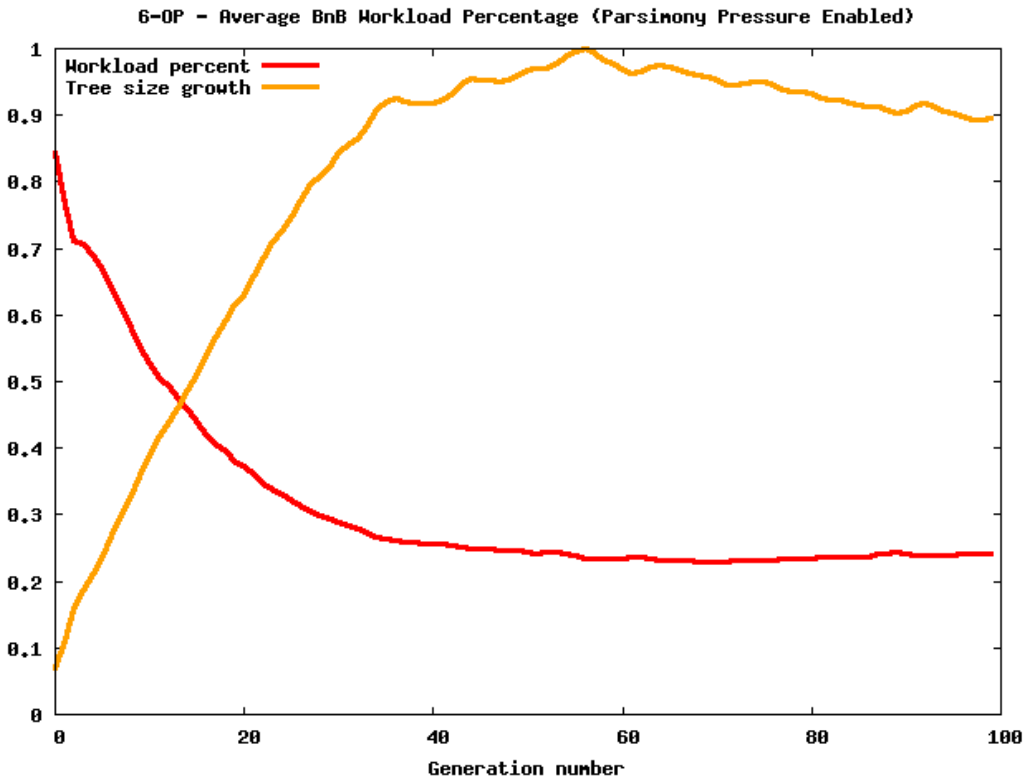

FIGURE 5.13: 6-Odd Parity with parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

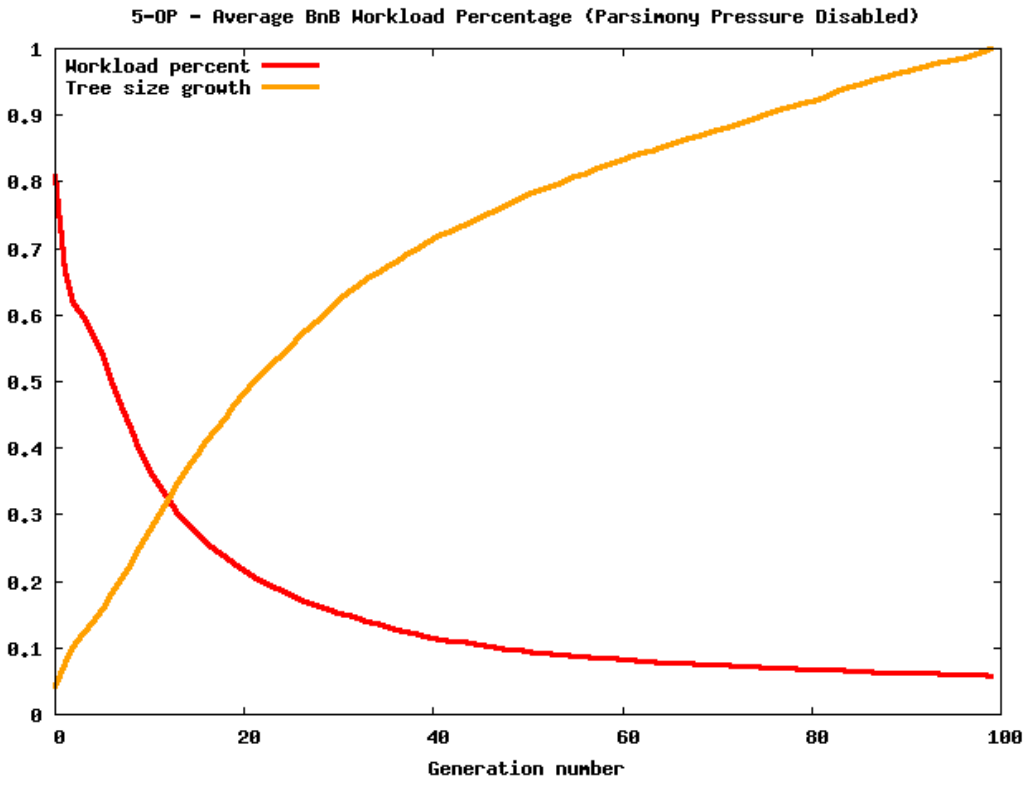

FIGURE 5.14: 5-Odd Parity without parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

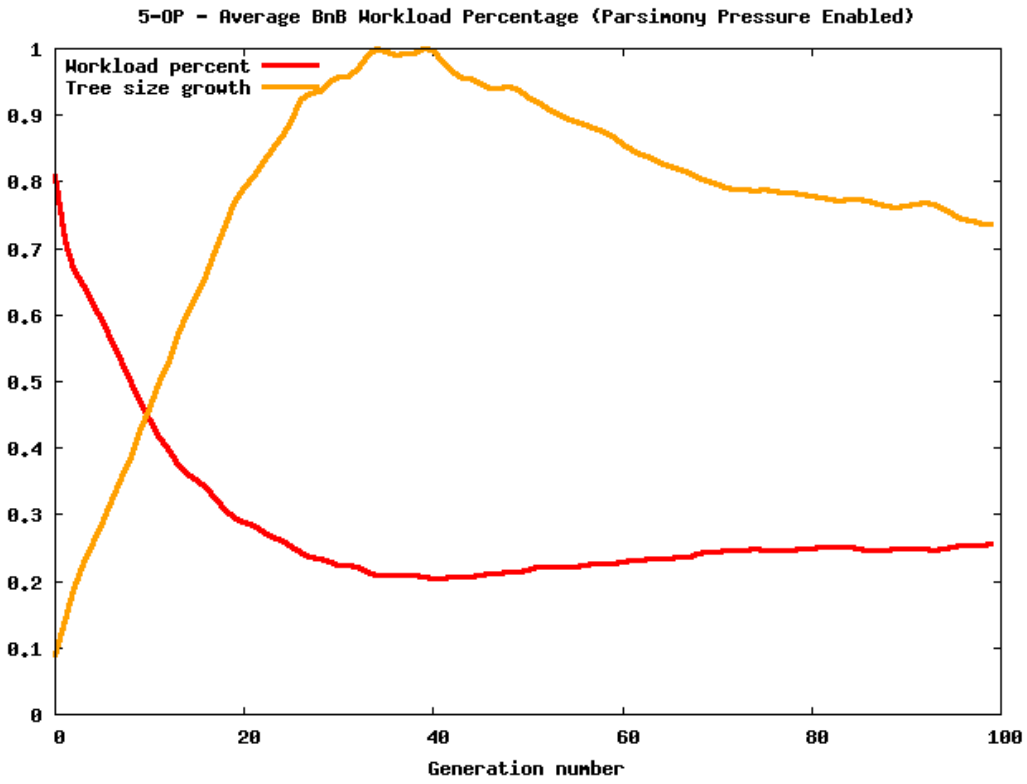

FIGURE 5.15: 5-Odd Parity with parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

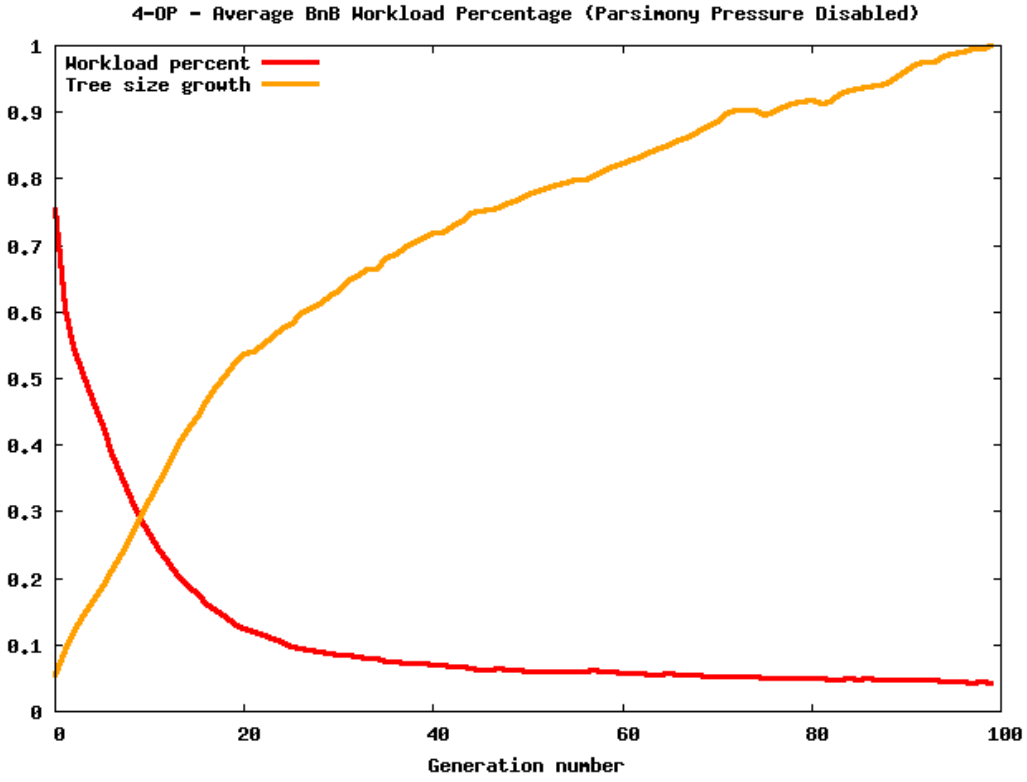

FIGURE 5.16: 4-Odd Parity without parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

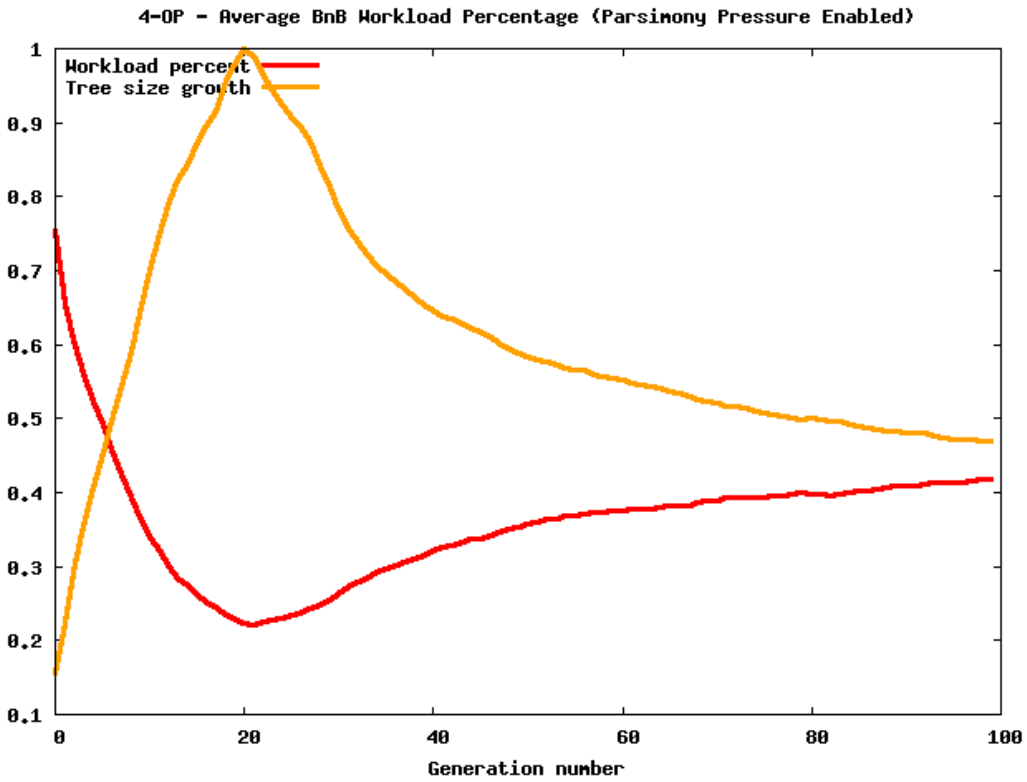

FIGURE 5.17: 4-Odd Parity with parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

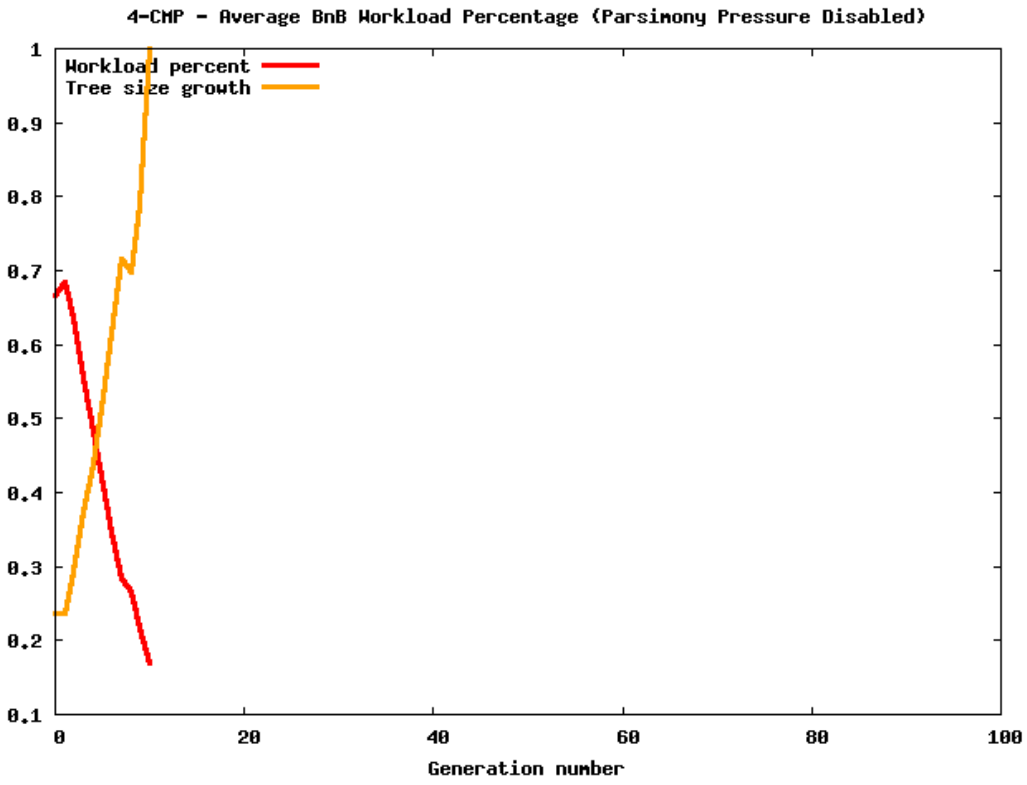

FIGURE 5.18: 4-Comparator without parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

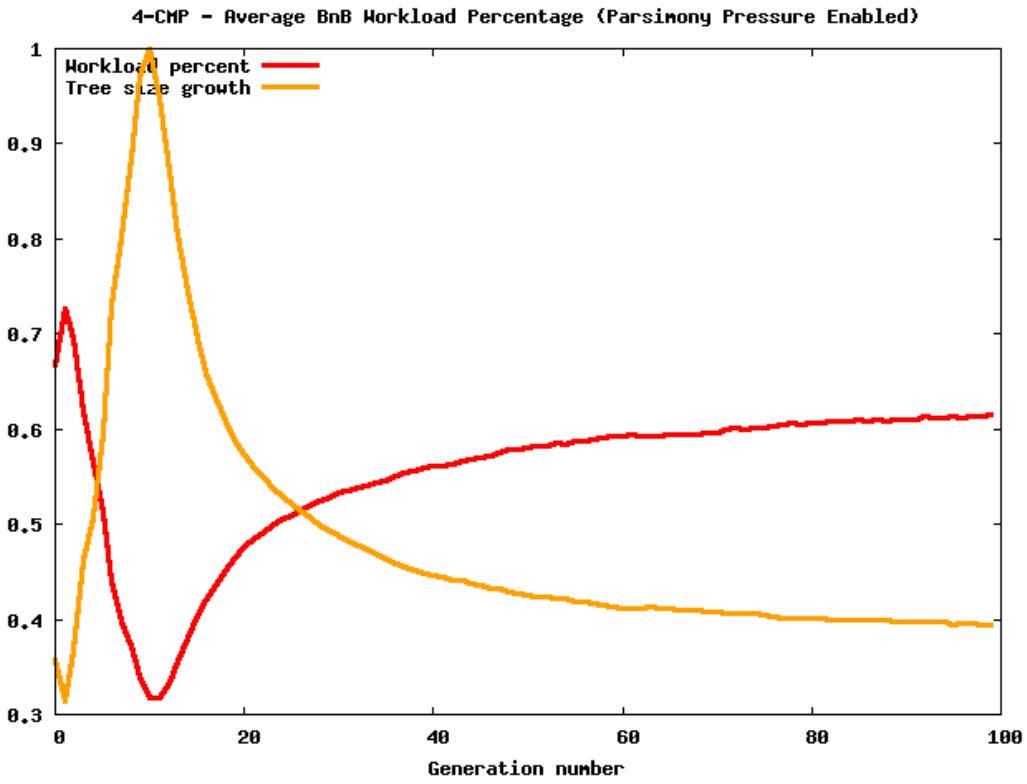

FIGURE 5.19: 4-Comparator with parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

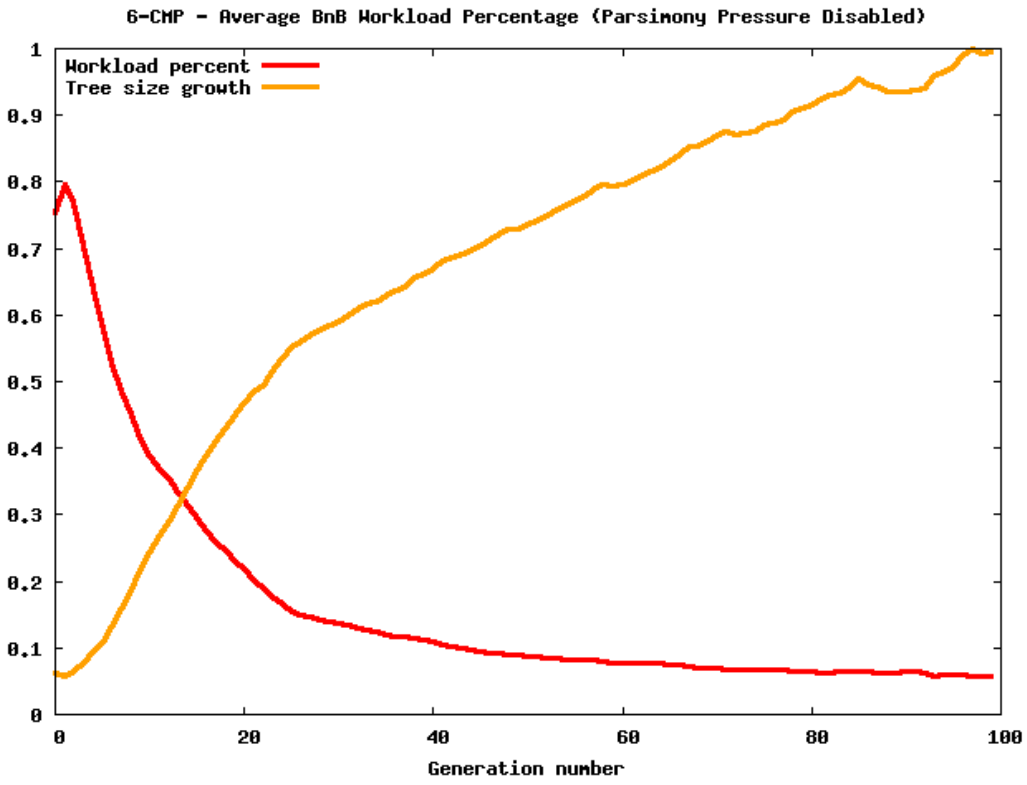

FIGURE 5.20: 6-Comparator without parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

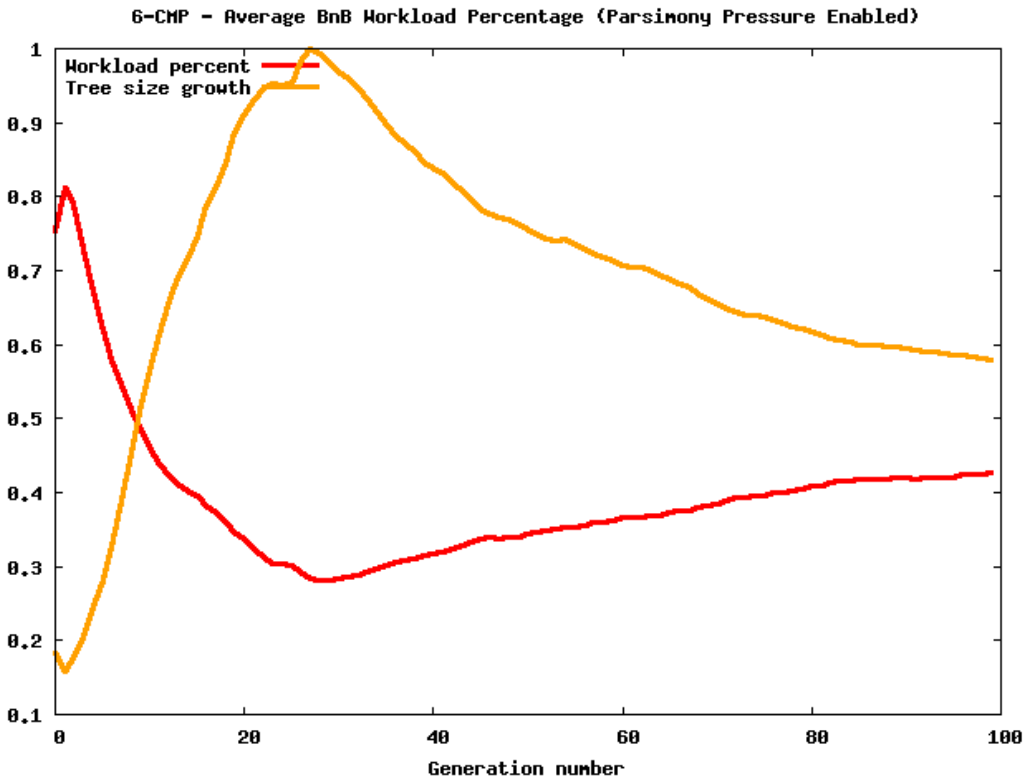

FIGURE 5.21: 6-Comparator with parsimony pressure. Average tree size and percentage of performed workload as a function of generation number

Average relative depth of best potential fitness

This section presents the data concerning the relative depth within the tree where the node with the maximum potential fitness is found. The depth is measured as the percent of the entire tree's depth, i.e. the length of the longest path from the root to a leaf. The plots are based on 100 independent evolutionary runs of 1024 individuals through 100 generations each. The width of the bars on the charts represents the averaged standard deviations of the relative height for each generation.

Finding a good (ideally, optimal) potential fitness quickly is instrumental in achieving good performance by the *Branch and Bound* algorithm, as it leads to performing a lot of cutoffs at high levels of the tree. Therefore, the average relative depth of the optimal node has a direct influence on how well the *BnB* can constrain the solution space. The results shown in this section can help explain the phenomenons described in Section 5.2.4.

It is apparent that the problems with parsimony pressure enabled generally require the evaluating agent to delve deeper into the tree to obtain the maximum potential fitness value than in case of their counterparts which do not employ parsimony pressure. Moreover, when the parsimony criterion is in effect, the curves seem to have a minimum located somewhere before the $50th$ generation, after which they turn towards the deeper regions of the tree. This may be a hint as to why the performance of *BnB* is worse for the cases where parsimony pressure is enabled. In contrast, the parsimony-free instances seem to follow the same pattern as it was the case for workload percentage (see Section 5.2.4) - the relative depth decreases exponentially.

It is also worth noting that the variances of the relative depths appear to be different depending on whether parsimony pressure is disabled or enabled. Although for both cases the variances generally stabilize after constantly decreasing for about 20 generations, in the former case they seem to be substantially smaller.

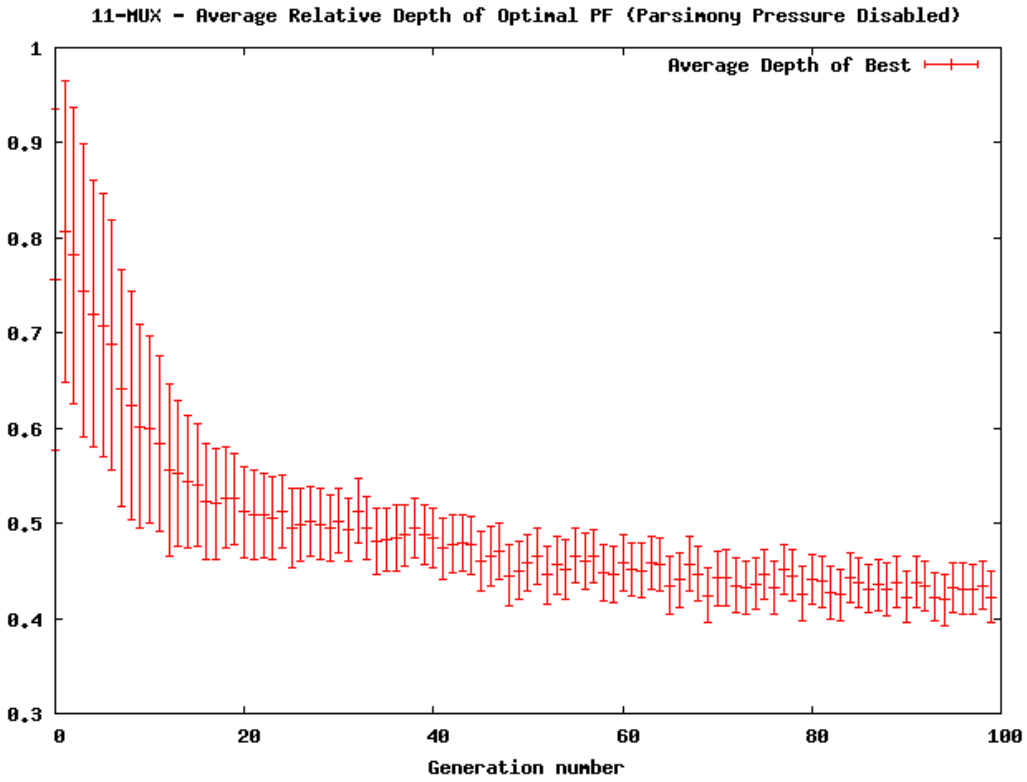

FIGURE 5.22: 11-Multiplexer without parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

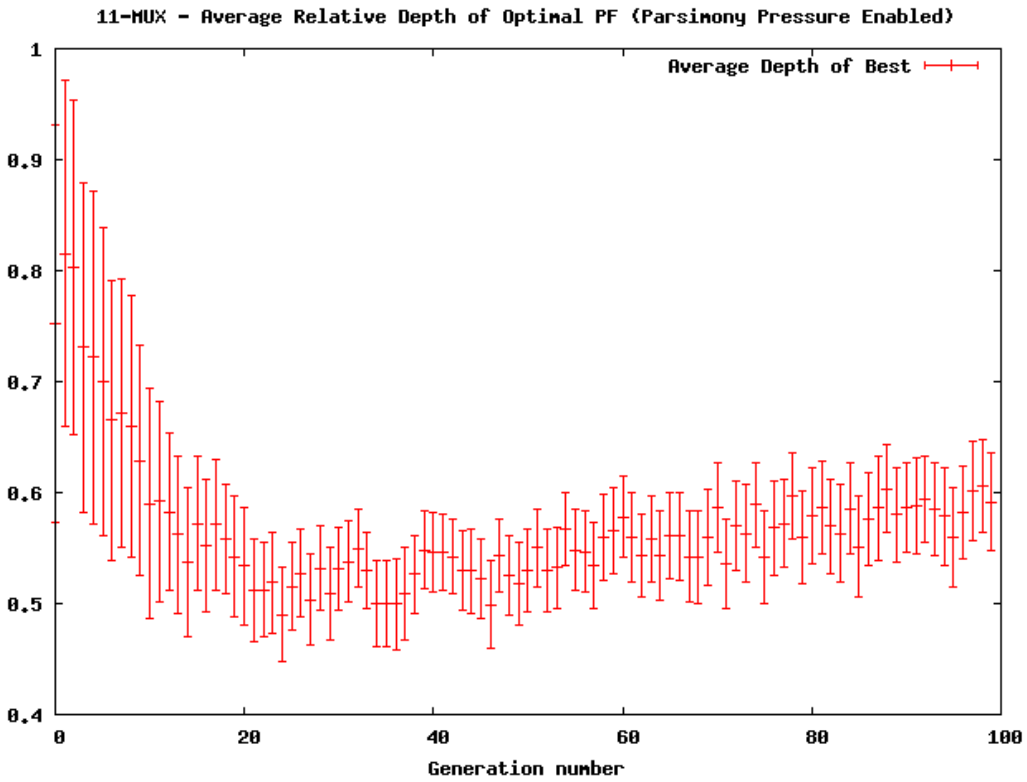

FIGURE 5.23: 11-Multiplexer with parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

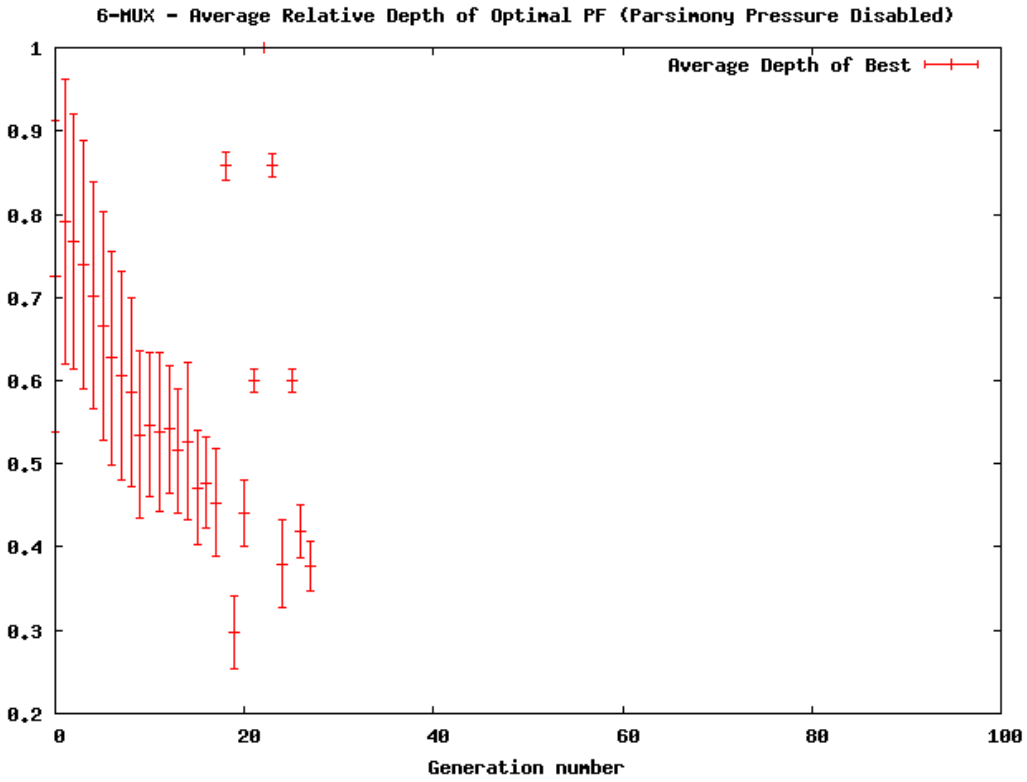

FIGURE 5.24: 6-Multiplexer without parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

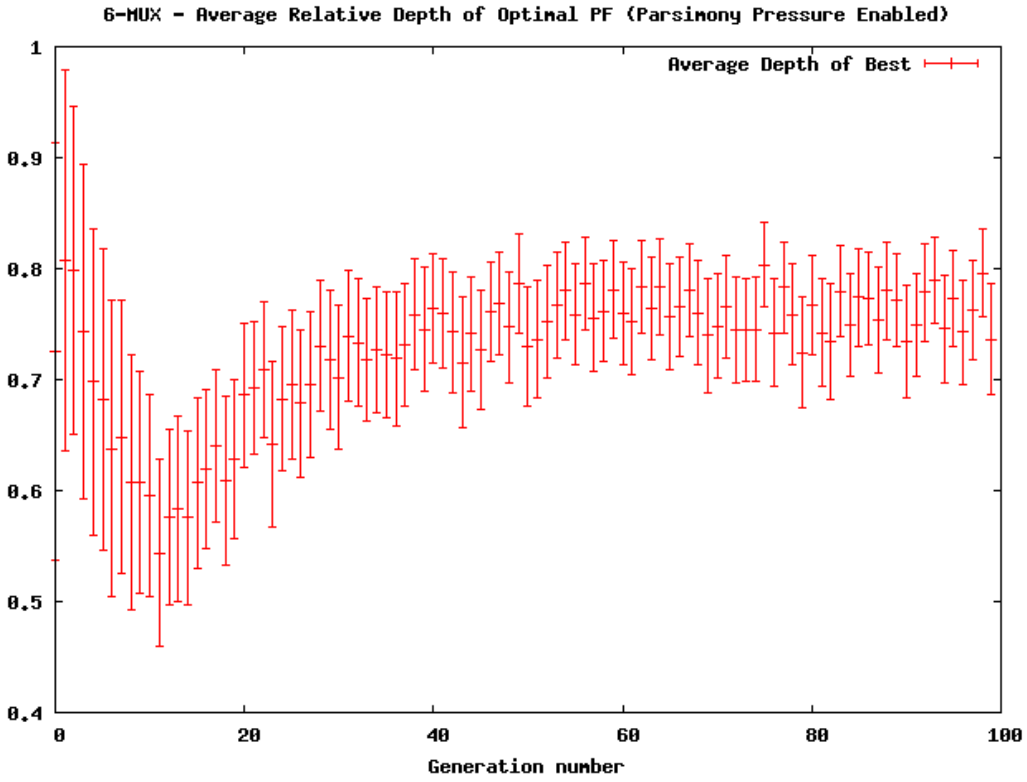

FIGURE 5.25: 6-Multiplexer with parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

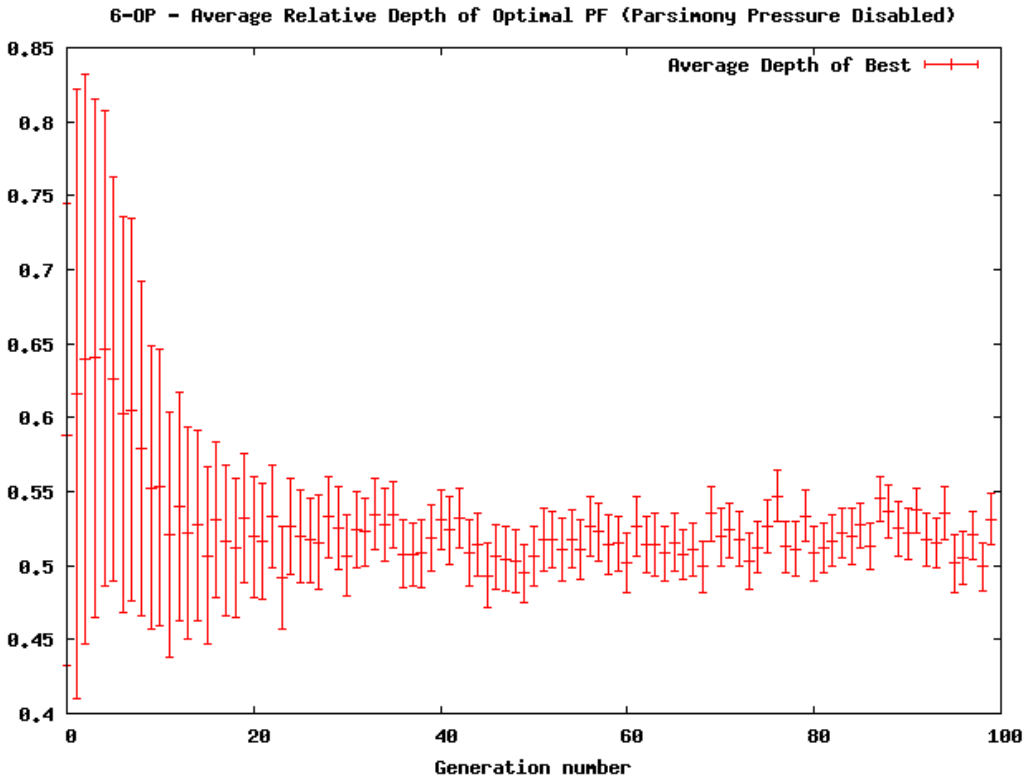

FIGURE 5.26: 6-Odd Parity without parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

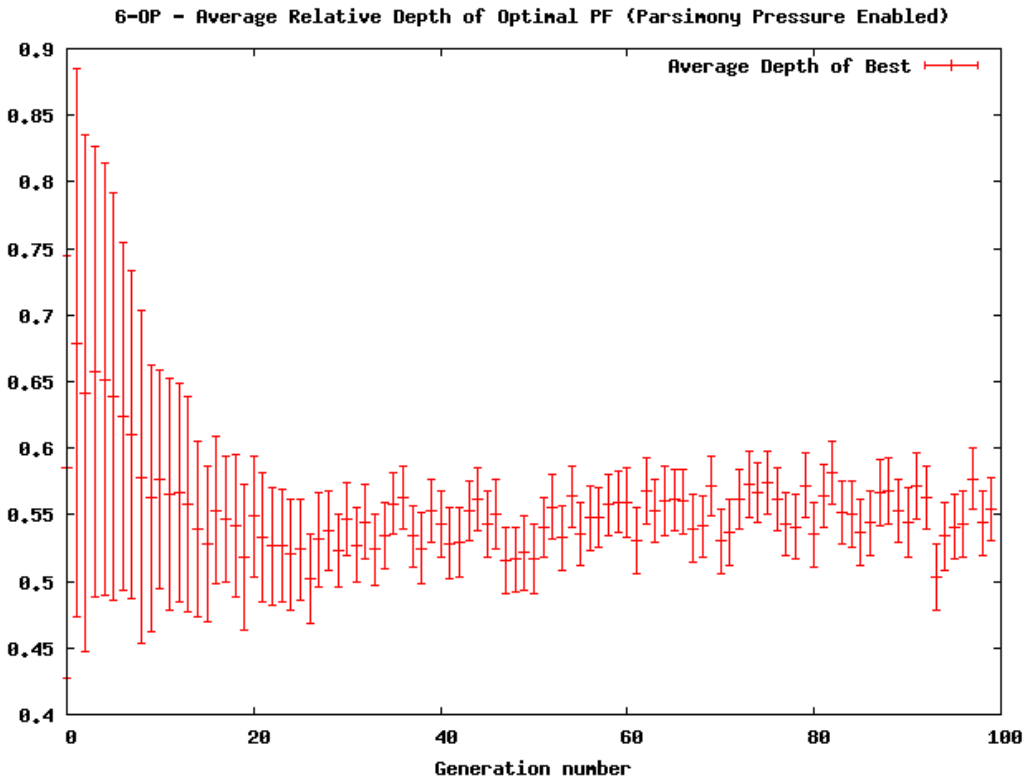

FIGURE 5.27: 6-Odd Parity with parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

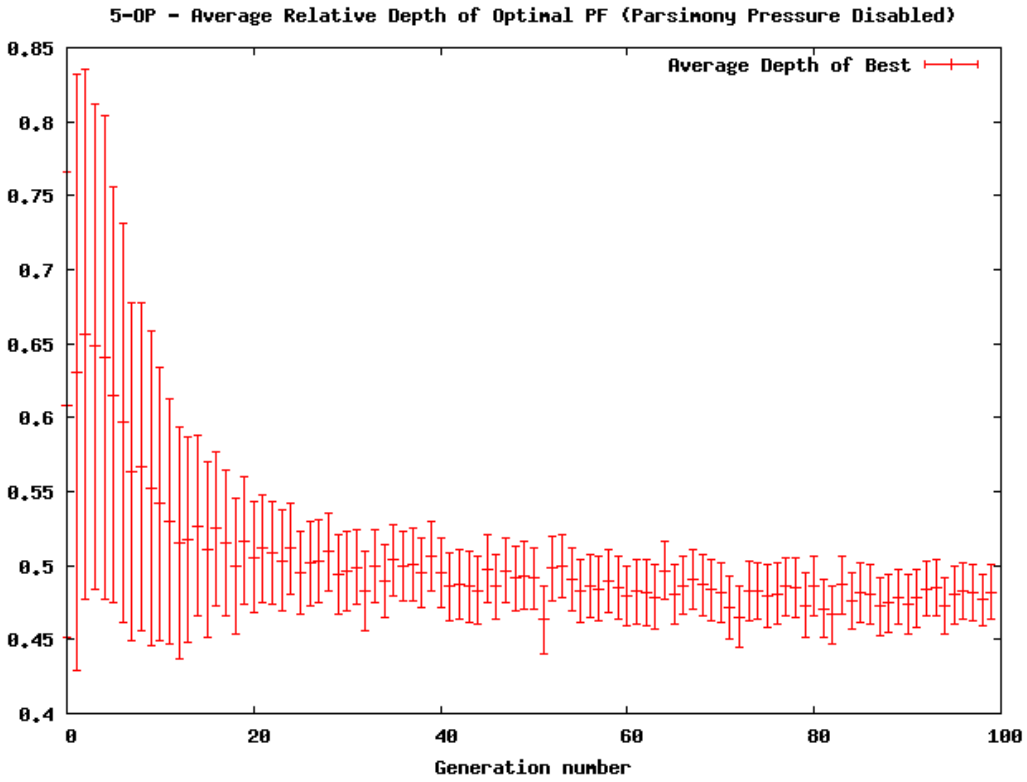

FIGURE 5.28: 5-Odd Parity without parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

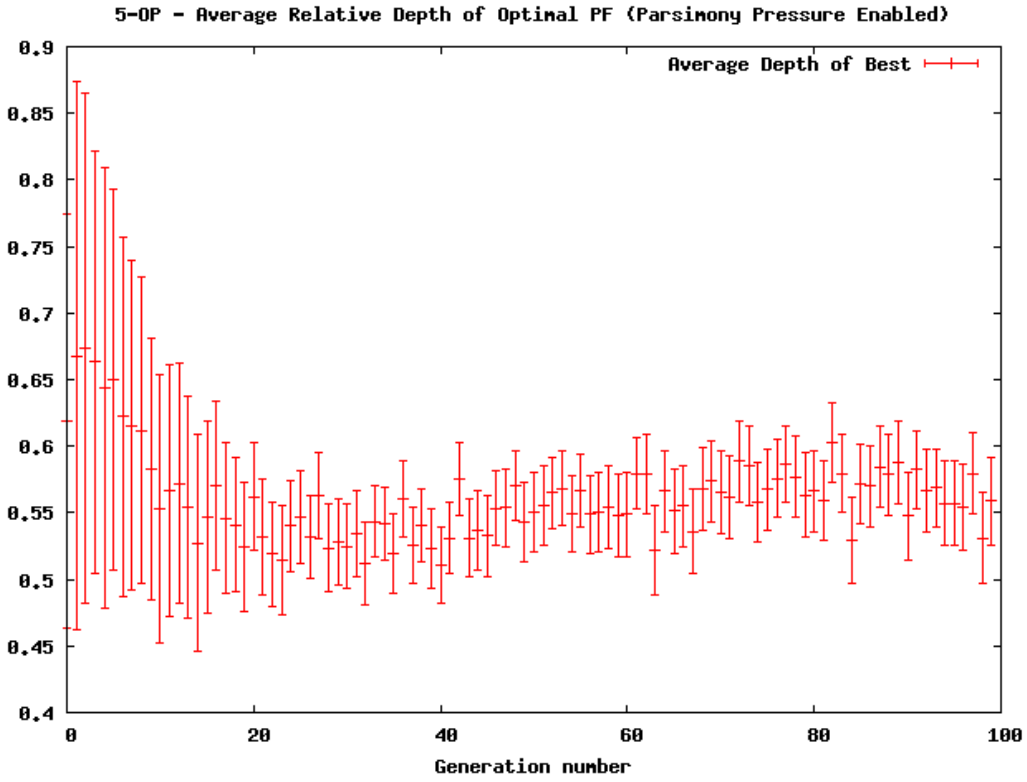

FIGURE 5.29: 5-Odd Parity with parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

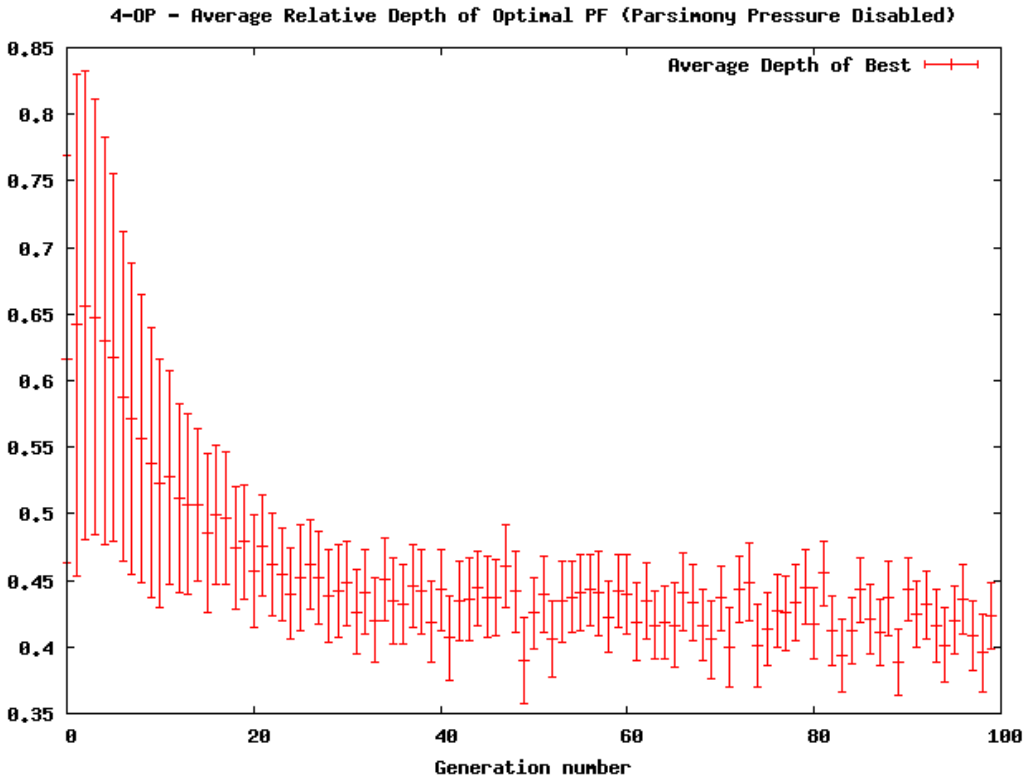

FIGURE 5.30: 4-Odd Parity without parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

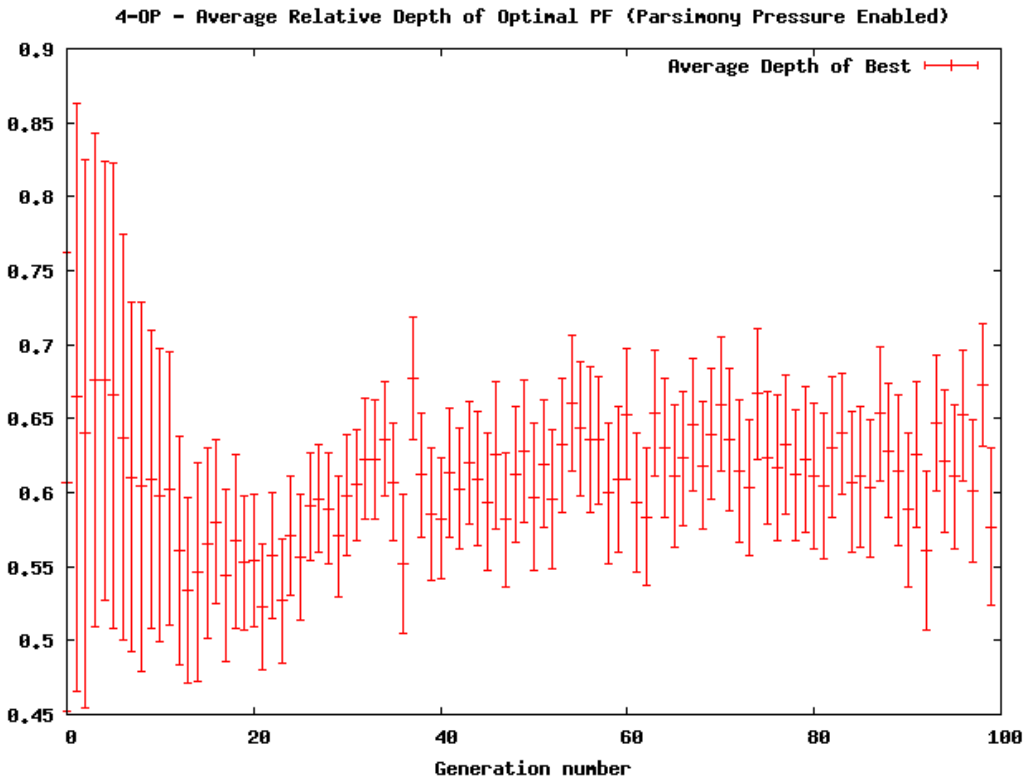

FIGURE 5.31: 4-Odd Parity with parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

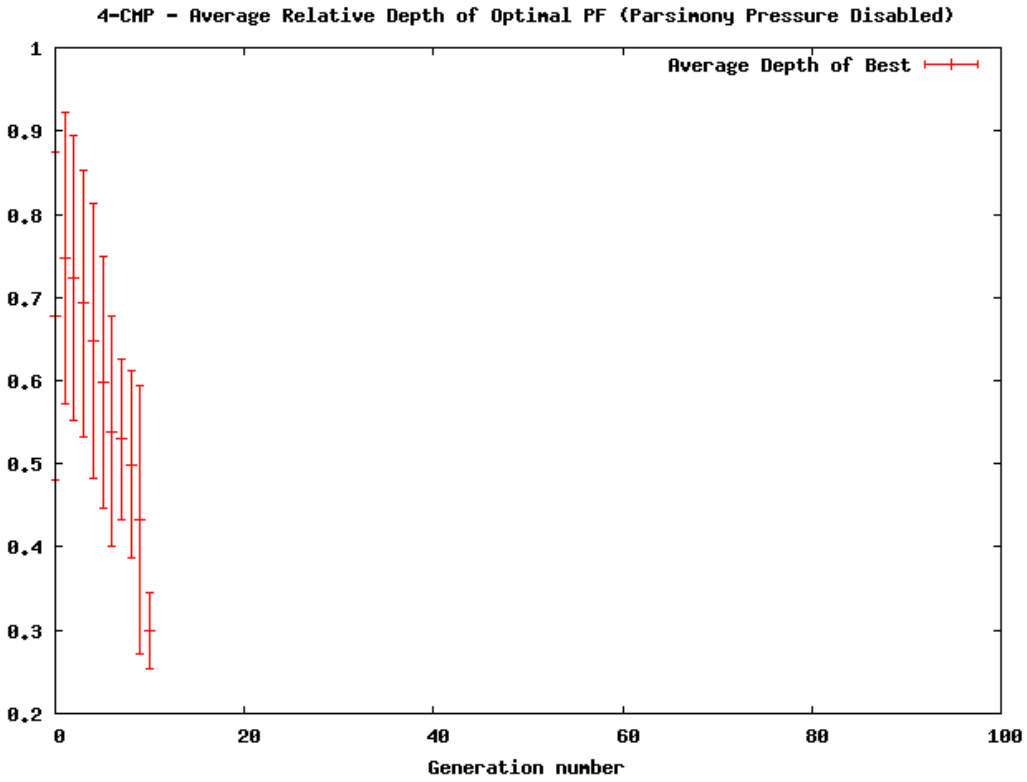

FIGURE 5.32: 4-Comparator without parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

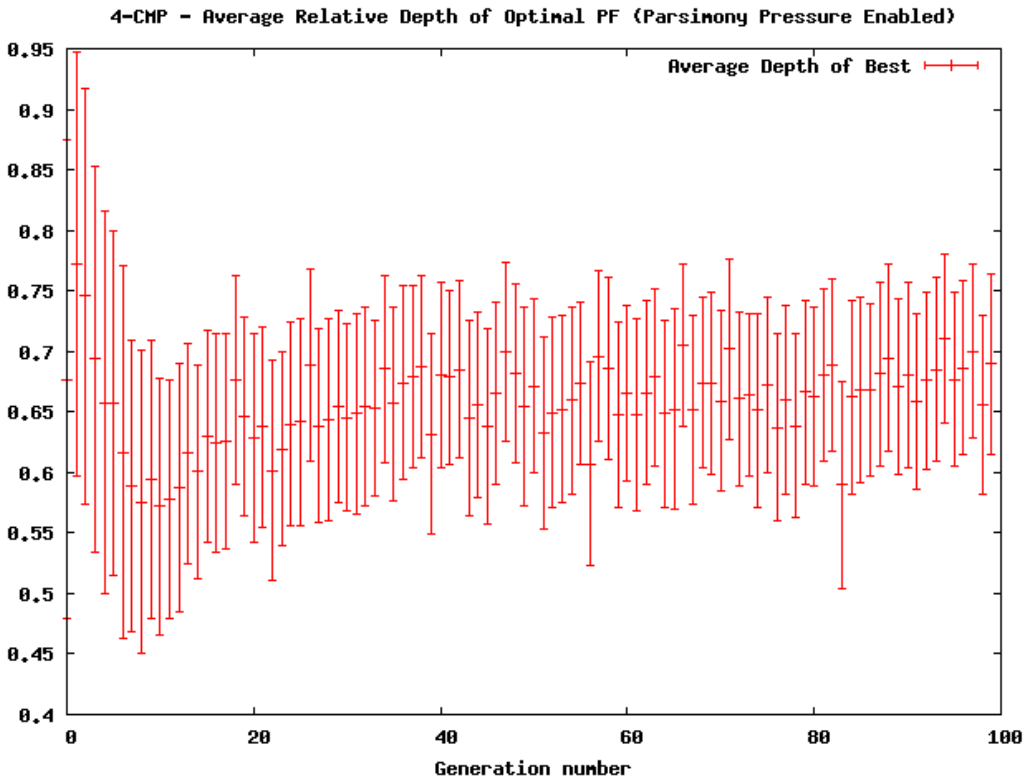

FIGURE 5.33: 4-Comparator with parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

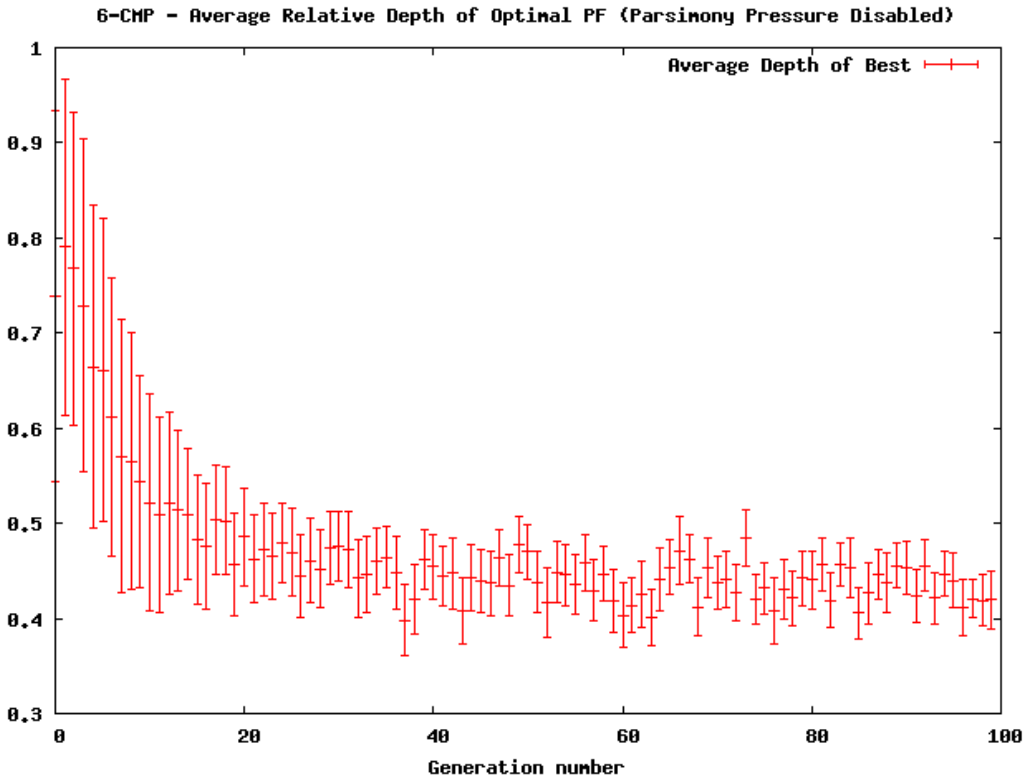

FIGURE 5.34: 6-Comparator without parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

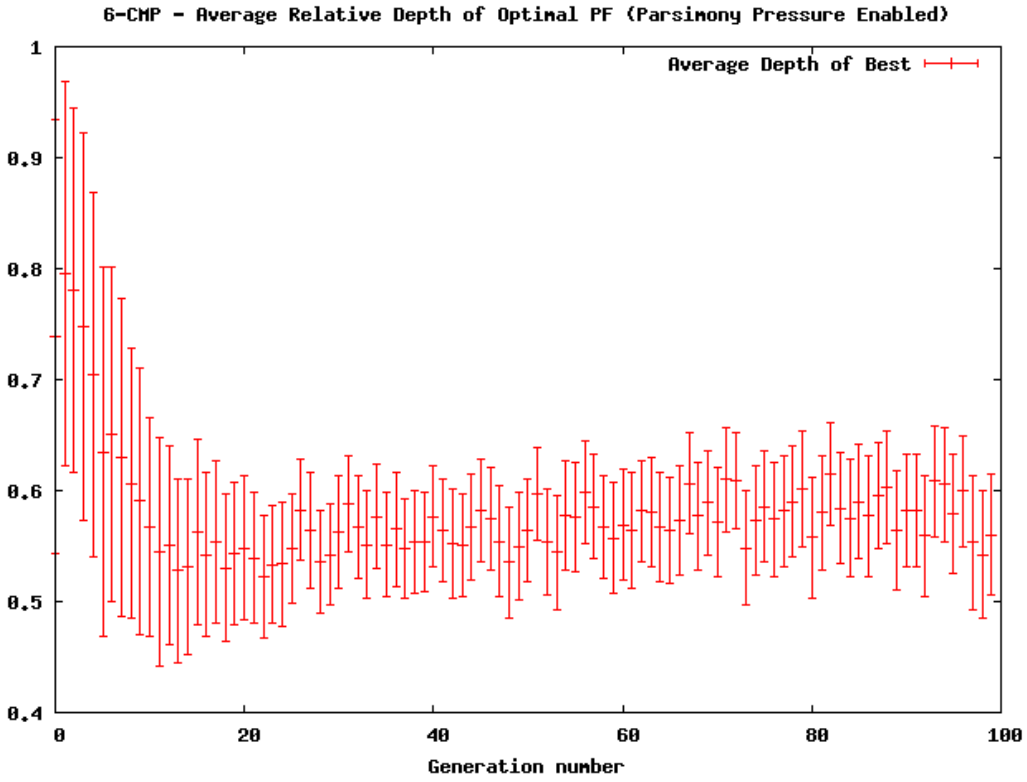

FIGURE 5.35: 6-Comparator with parsimony pressure. Average relative depth at which the optimal Potential Fitness is found, plotted against generation number. Error bars indicate averaged standard deviation.

5.2.5 Distribution of potential fitness

Histogram of potential fitness

This section concerns the distribution of potential fitness values within an average individual (tree). The diagrams show histograms of relative potential fitness (percent of maximum) as a function of generation number. Each point of the plot is averaged over 1024 individuals times 100 independent runs. Because the potential fitness value can be negative, the value 0% corresponds to -2^d rather than 0 (where d is the problem dimension).

The most important information which can be elucidated from the results presented in this section is the dynamics of the maximum potential fitness within the average individual, i.e. the way the percentage of nodes having maximum potential fitness within the tree changes in the course of the evolution. This relates directly to the convergence rate of the evolutionary process. A brief theoretical analysis of this relationship will be performed to facilitate the understanding of the results' significance.

Consider a population P comprised of N individuals. The next generation will be created with the use of tournament selection of size k, crossover with probability p_{xo} and reproduction with probability $1 - p_{xo}$. Assume that there are exactly b individuals having the optimal potential fitness within population P. Also, note that the potential fitness of the entire tree is the maximum potential fitness of any of its contexts, and during the crossover operation, the insertion point is picked randomly with uniform probability. In order for the evolutionary process to progress quickly in the right direction, it would be best if at least one of the best-valued individuals were picked for crossover and its crossover point corresponded to the context with the maximum potential fitness.

The probability that at least one of the b optimal nodes will be picked to participate in a single tournament during selection is:

$$p_s = 1 - \frac{\binom{N-b}{k}}{\binom{N}{k}} = 1 - \frac{\prod_{i=N-b-k+1}^{N-k} i}{\prod_{j=N-b+1}^{N} j} \tag{5.3}$$

Of course, since the individual under consideration has the maximum value within the population, it will win any tournament and be selected for crossover. The crossover operation itself takes two individuals (parents) as arguments and creates only one new individual (offspring); the other one is discarded. Therefore, although two parents are picked through selection for crossover, the desired effect can only be achieved if the optimal individual is picked as the first of the two parents. This means that the expected number of cases where the best individual is considered the base tree in a crossover operation can be expressed as:

$$n_{xo} = N p_{xo} p_s \tag{5.4}$$

Moreover, we require that the correct context be chosen as the crossover point. In order for this to happen at least once, it is necessary that:

$$n_{xo} * p_{max} \geq 1 \tag{5.5}$$

where p_{max} is the probability that the optimal context will be chosen, which may be interpreted as the percentage of all nodes within the tree which have the maximum potential fitness. Therefore,

$$p_{max} \geq \frac{1}{n_{xo}} \tag{5.6}$$

Equation 5.6 can now be applied to the actual experiment. As stated in Section 5.1, the parameter values are as follows: N=1024, k=7, p_{xo}=0.9. As for the assumed number of optimal solutions, two cases will be considered: *b=1* and *b=2*. The corresponding p_{max} values are:

$$p^1_{max} \approx 0.16, p^2_{max} \approx 0.08 \tag{5.7}$$

The results show that for all considered problems and their variations, the average percentage of maximum-valued nodes does not drop below 11%. This means that as long as there are at least two individuals with the maximum fitness in every generation, the desired effect should occur at least once. However, this does not hold for the case where there is only one individual with the maximum potential fitness. Such a situation may occur in the closing generations for the harder problems like *11-Multiplexer* or *6-Odd Parity*. Therefore it is hypothesized that these problems could benefit from an increase in population size (in terms of average solution quality and/or success rate).

Aside from the percentage of optimum nodes, the potential fitness histogram appears to be heavily shifted towards the higher values. As an example, consider the *6-Comparator* problem. The vast majority of the nodes has an average quality of over 95% of the maximum.

Also, it seems that when parsimony pressure is enabled, the percentage of optimum nodes decreases considerably. This is especially visible in the second half of the evolutionary process, i.e. generations 50-100.

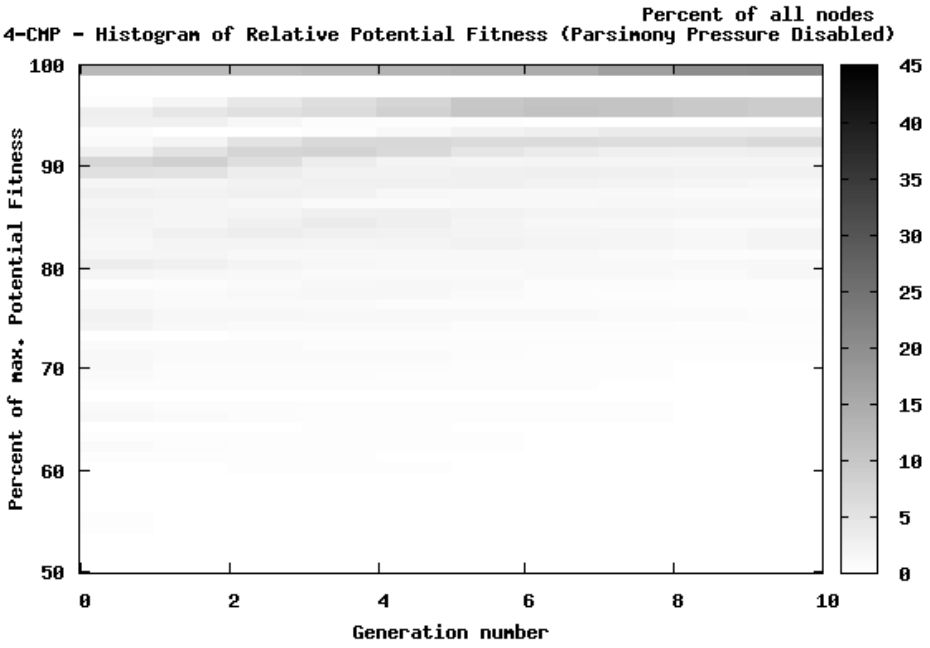

FIGURE 5.36: 4-Comparator without parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

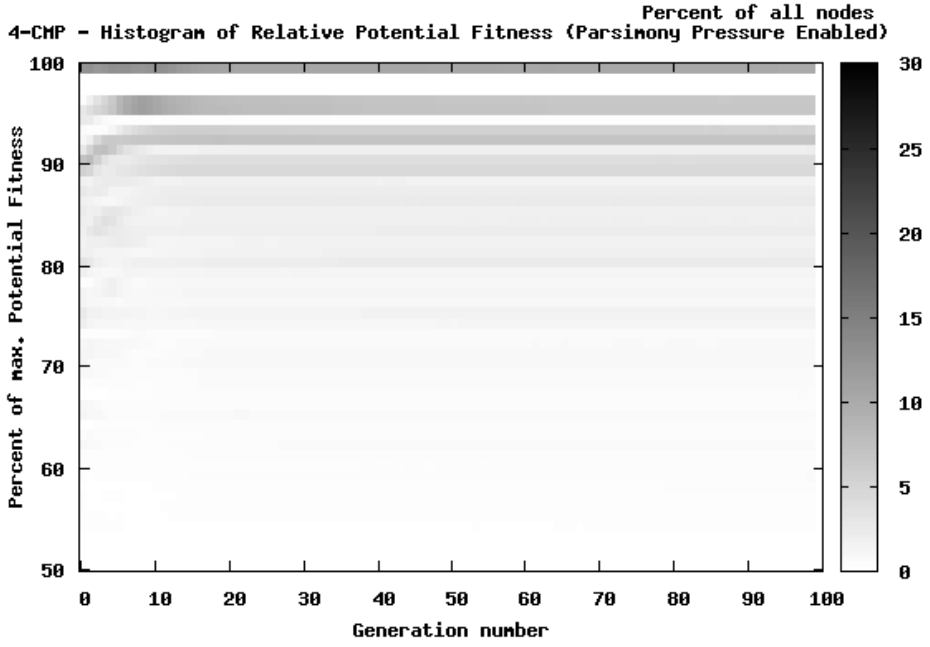

FIGURE 5.37: 4-Comparator with parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

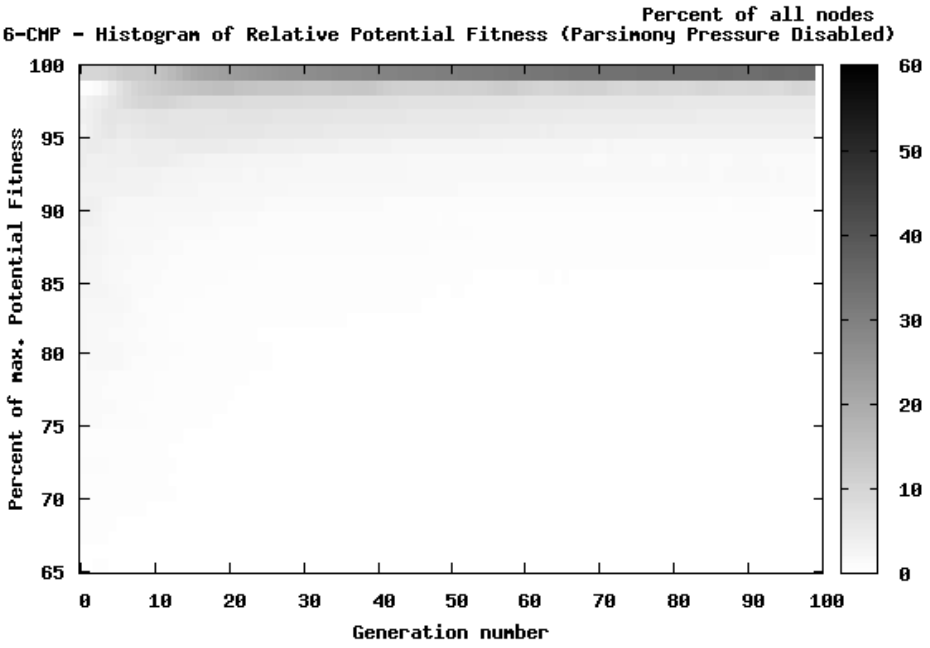

FIGURE 5.38: 6-Comparator without parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

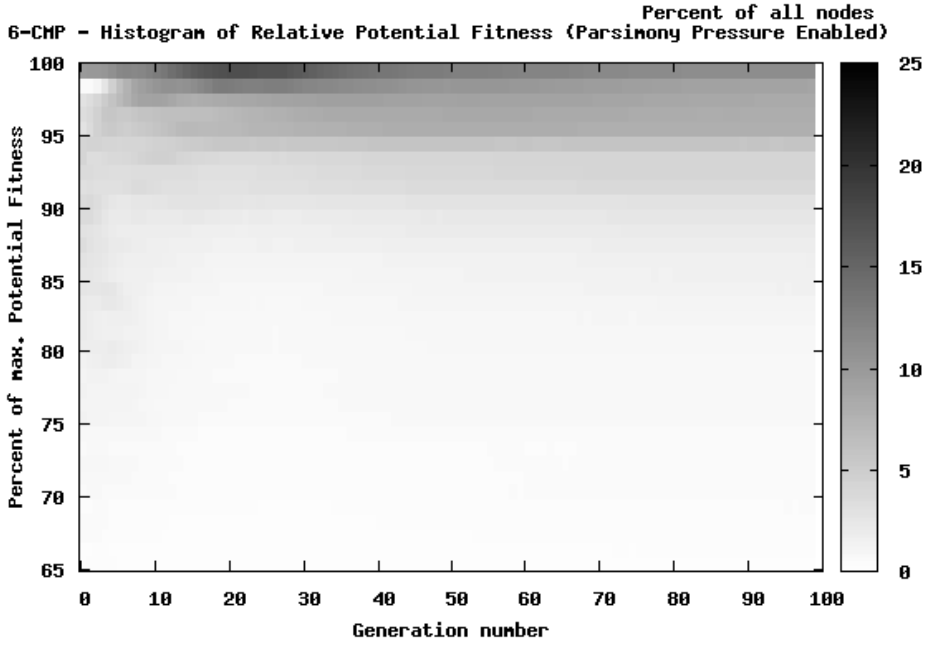

FIGURE 5.39: 6-Comparator with parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

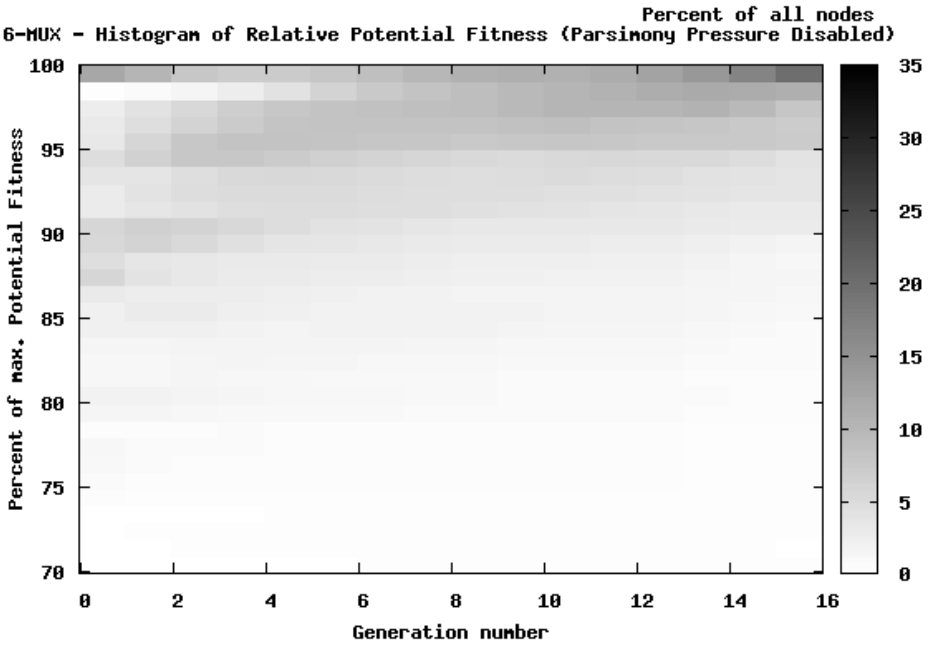

FIGURE 5.40: 6-Multiplexer without parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

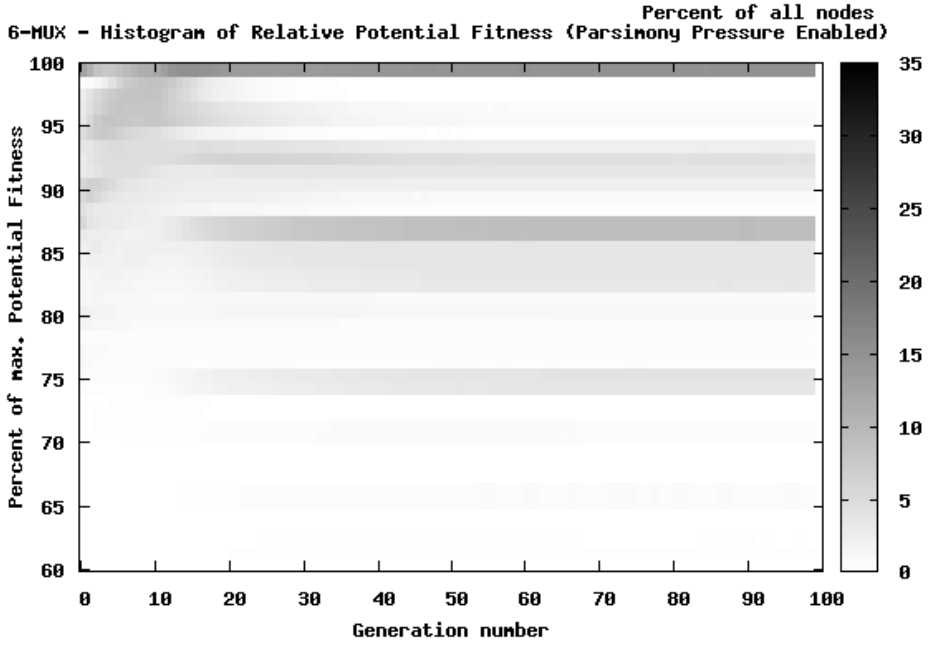

FIGURE 5.41: 6-Multiplexer with parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

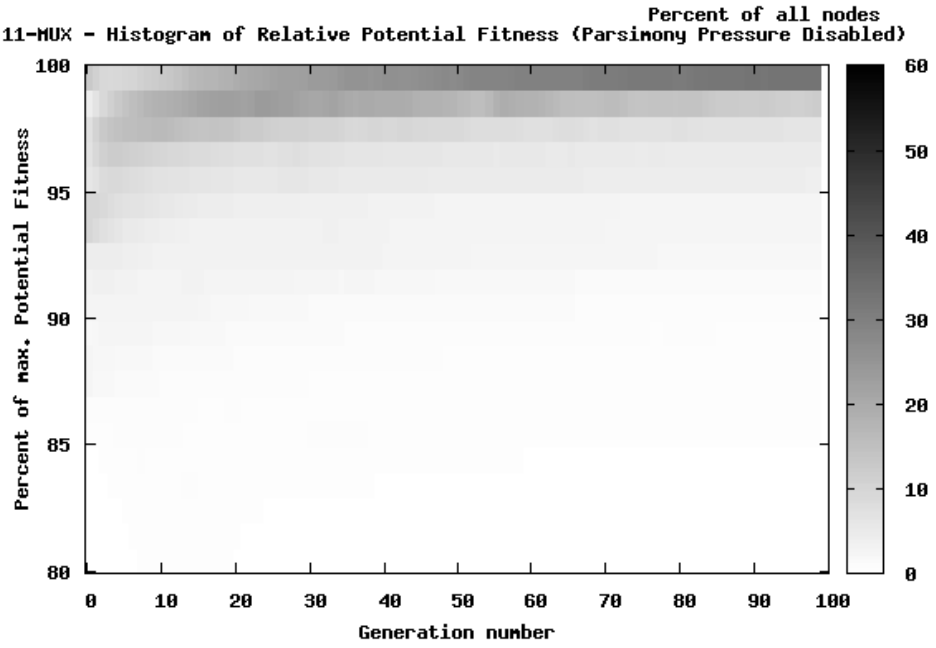

FIGURE 5.42: 11-Multiplexer without parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

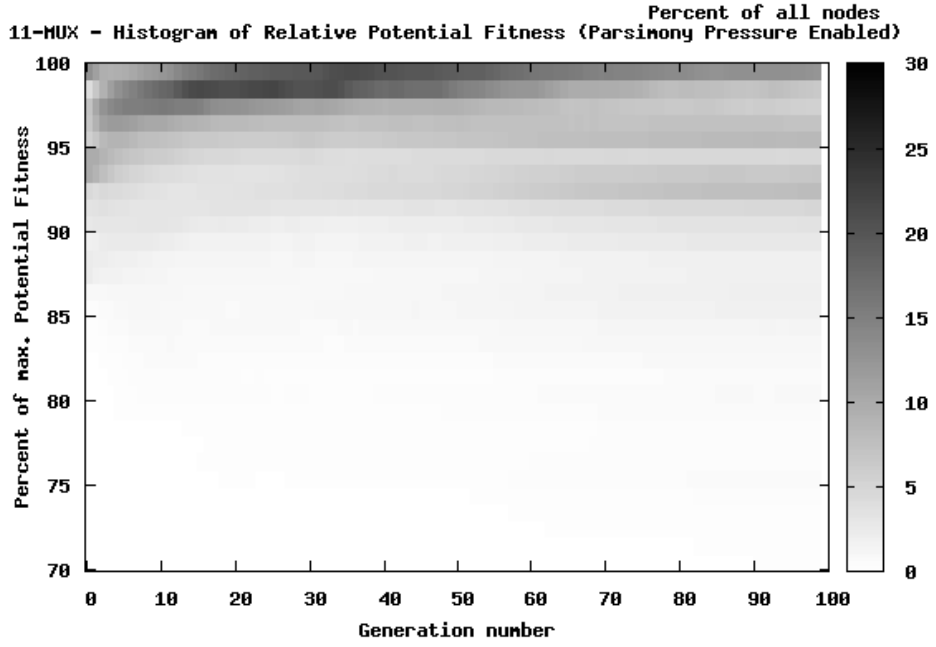

FIGURE 5.43: 11-Multiplexer with parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

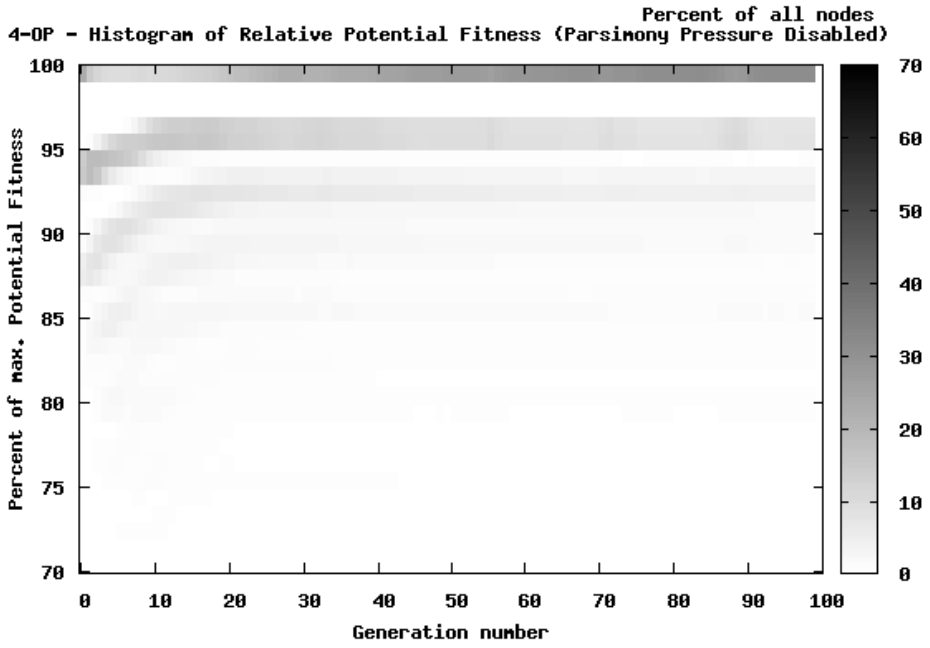

FIGURE 5.44: 4-Odd Parity without parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

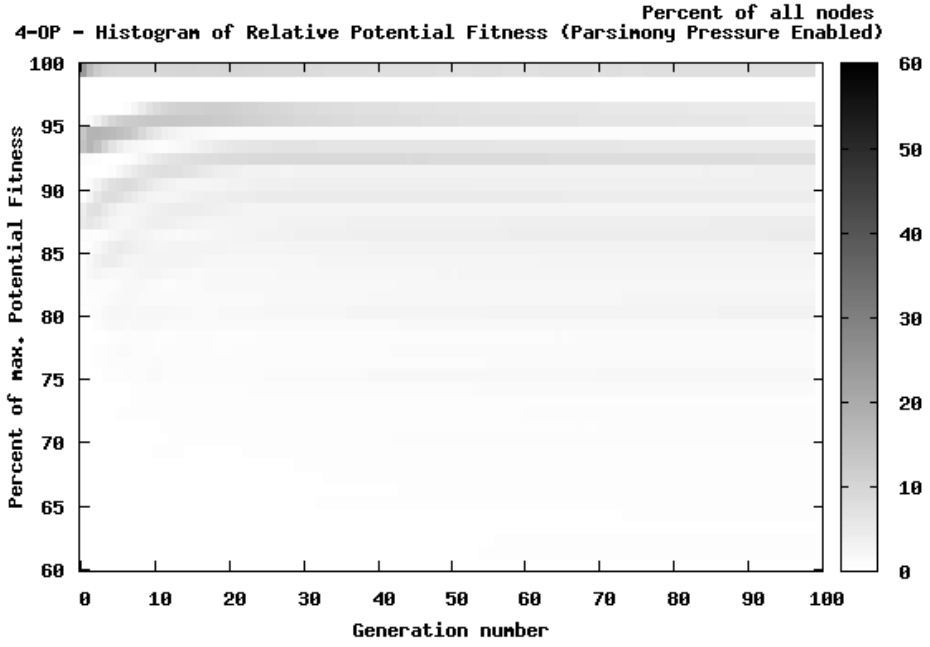

FIGURE 5.45: 4-Odd Parity with parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

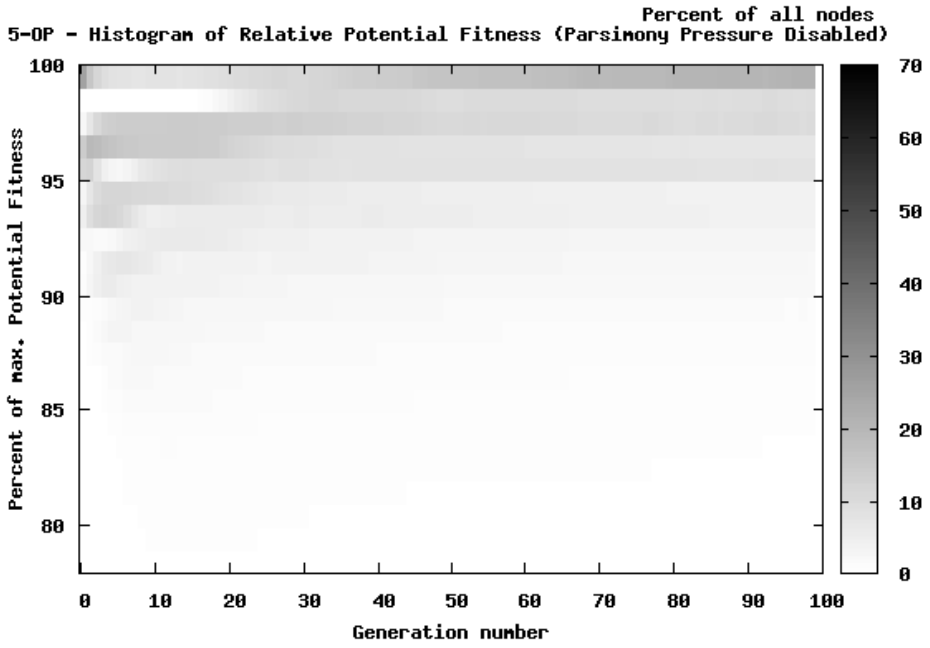

FIGURE 5.46: 5-Odd Parity without parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

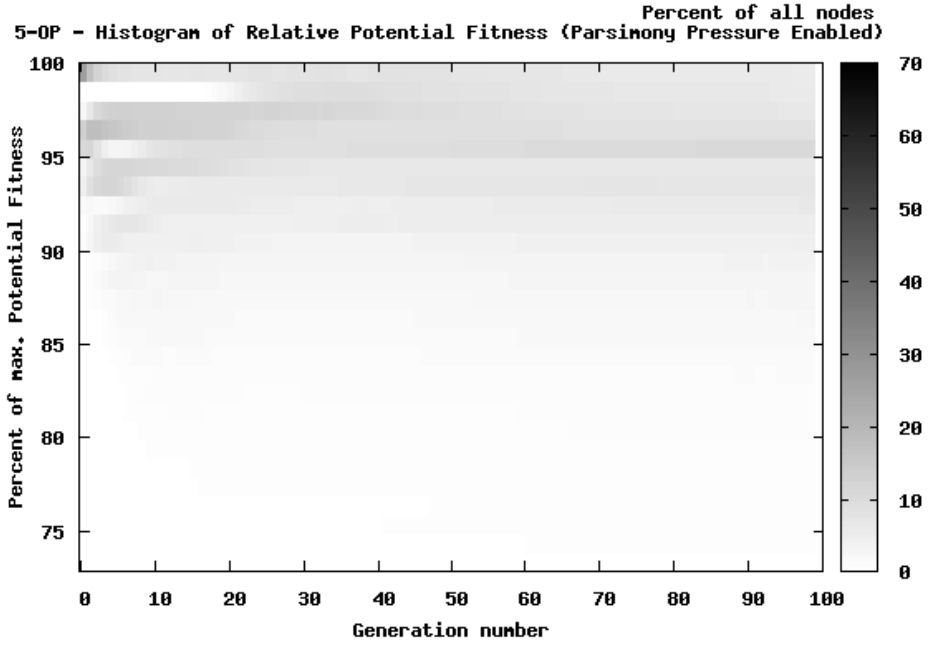

FIGURE 5.47: 5-Odd Parity with parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

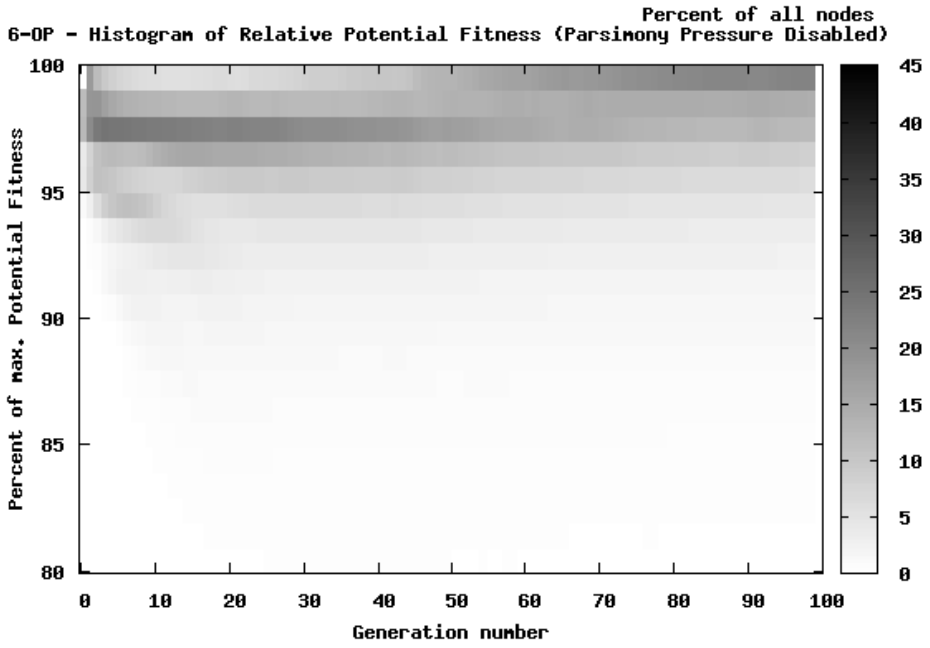

FIGURE 5.48: 6-Odd Parity without parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

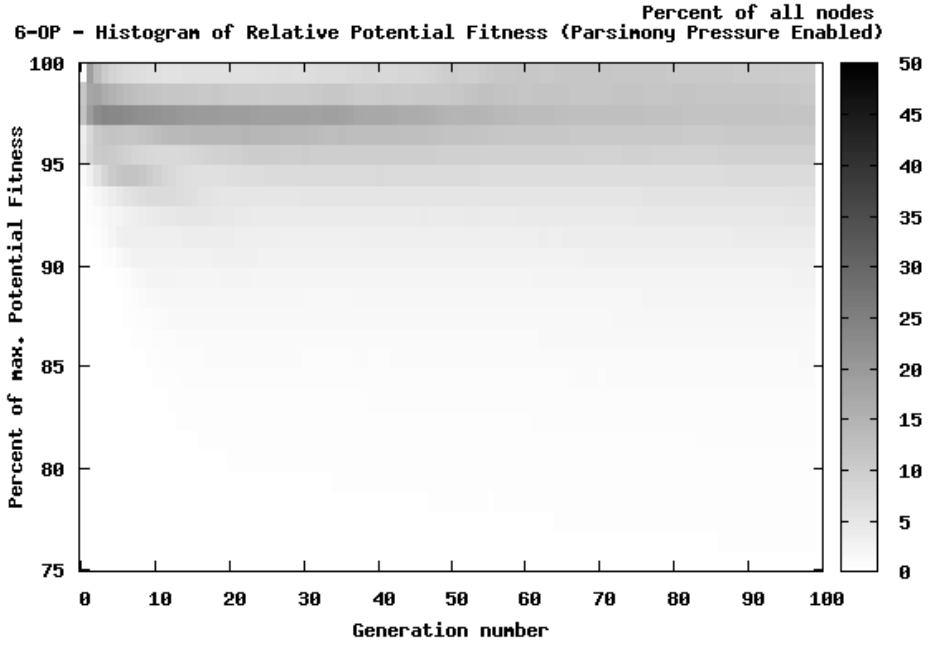

FIGURE 5.49: 6-Odd Parity with parsimony pressure. Histogram of Potential Fitness across all generations. Average from 100 independent runs.

Spectrogram of relative potential fitness vs. relative depth in tree

This section contains the results which show the distribution of average relative potential fitness as a function of average depth of the node within the tree and generation number. The results are averaged over 100 independent runs of 1024 individuals by 100 generations each. The color of a point on the plot corresponds to the percentage of the maximum fitness within the tree for a given percentage of the tree's height at a specified generation number. The purpose of this data is to answer questions of the type 'what is the average quality of nodes which lie within $\frac{1}{3}$ of the tree height towards the end of the evolution ?'.

The results indicate that the average quality of nodes in the lower portions of the tree (at greater depth) is generally better that that of the nodes situated near the root. However, at about 30-40% of the tree's height, this no longer applies and the differences are minimal for the remainder of the depth range (40-100%).

An interesting phenomenon can be observed when comparing pairs of corresponding problems for enabled/disabled parsimony pressure. In case of disabled parsimony pressure, the diagrams become blurred around generation *50*, which continues until the end of the evolution. In contrast, the diagrams representing problem variations with parsimony pressure enabled are comprised of solid, sharp lines which show little or no change throughout the entire generation span. The reason this occurs is probably the vastly superior mean tree size of the first case compared to the second - the basic algorithm (not employing parsimony pressure) can produce trees up to three times as large (on average) as the ones constructed by the parsimony-enhanced version. Therefore, more data contribute to every point on the plot, which makes it more random.

The results suggest that there is no optimal relative depth where it is best to search for solutions of excellent quality (near 100%), although there are extensive regions within the tree where the quality exceeds 90% of the best potential fitness, and therefore such a value can be achieved without much computational effort. However, the results for the randomized heuristic presented in Section 5.2.2 show that this is not enough to significantly outperform standard, canonical Genetic Programming.

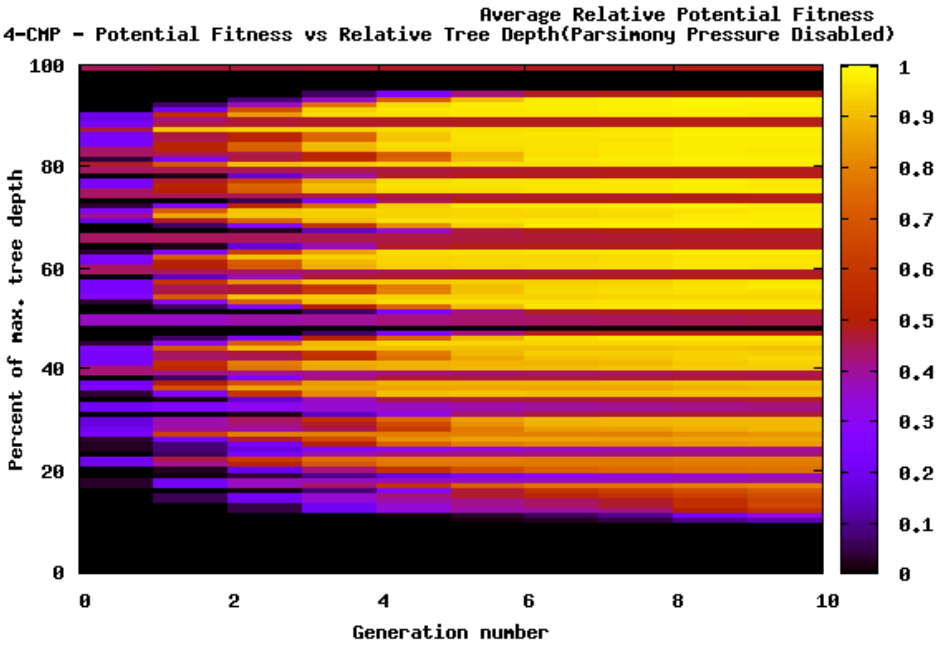

FIGURE 5.50: 4-Comparator without parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

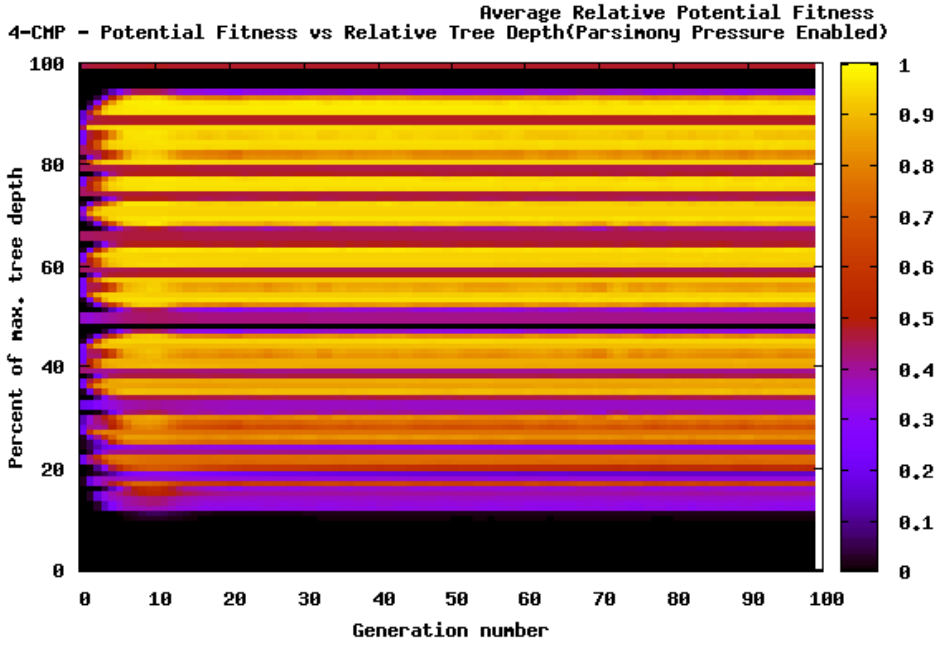

FIGURE 5.51: 4-Comparator with parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

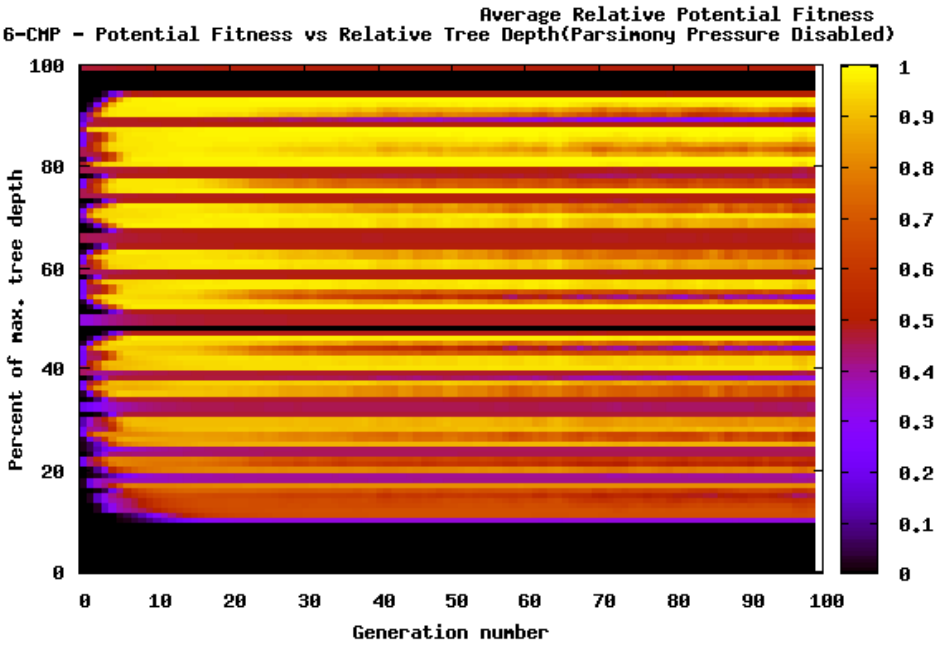

FIGURE 5.52: 6-Comparator without parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

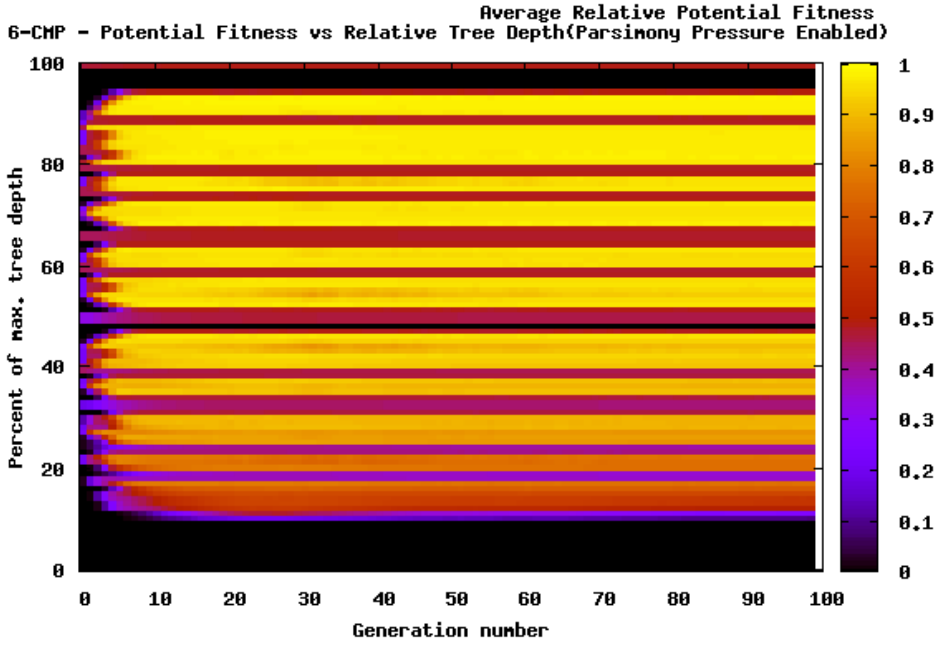

FIGURE 5.53: 6-Comparator with parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

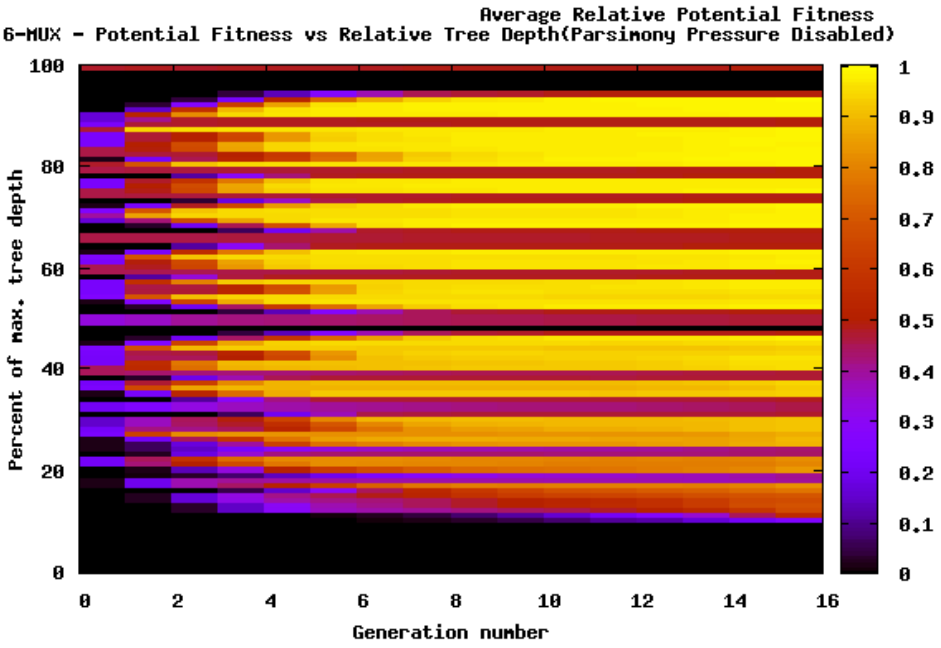

FIGURE 5.54: 6-Multiplexer without parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

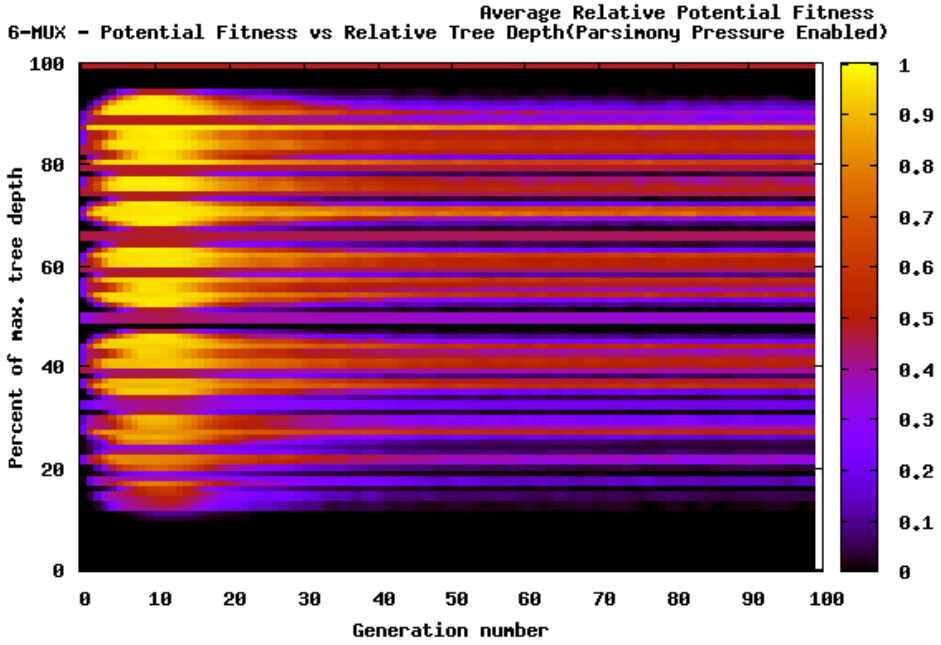

FIGURE 5.55: 6-Multiplexer with parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

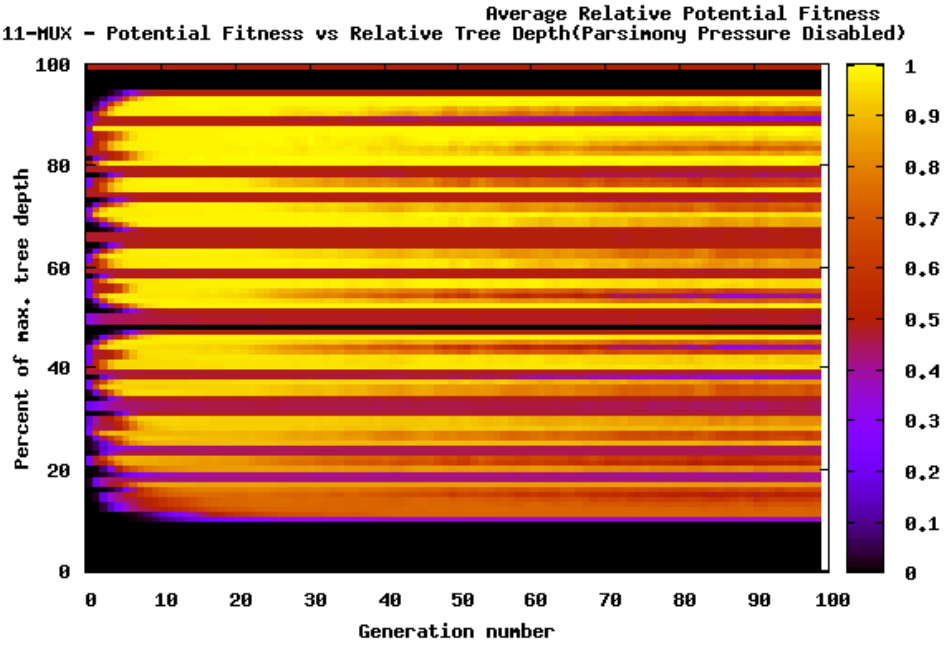

FIGURE 5.56: 11-Multiplexer without parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

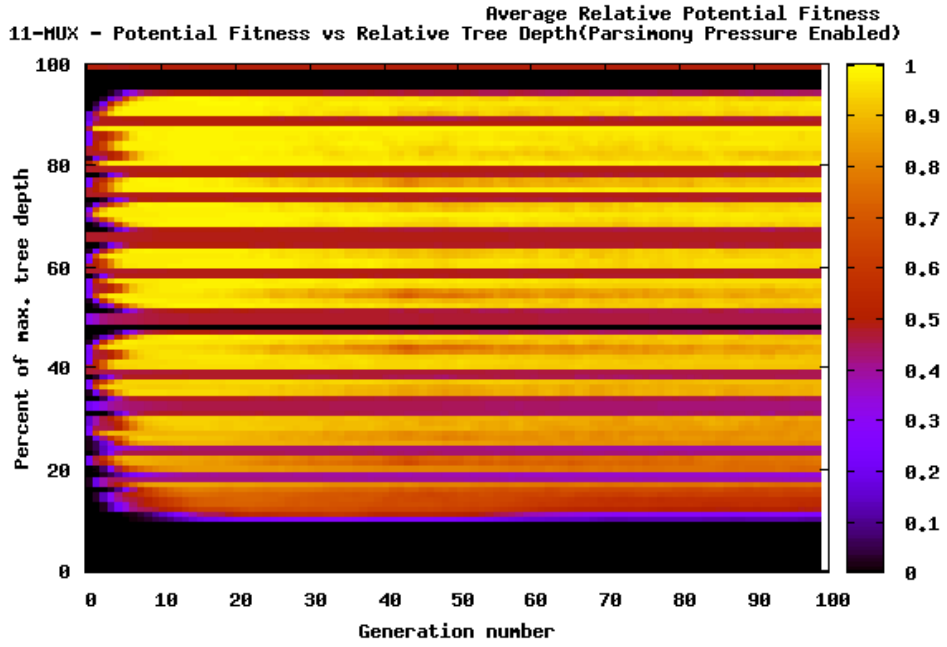

FIGURE 5.57: 11-Multiplexer with parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

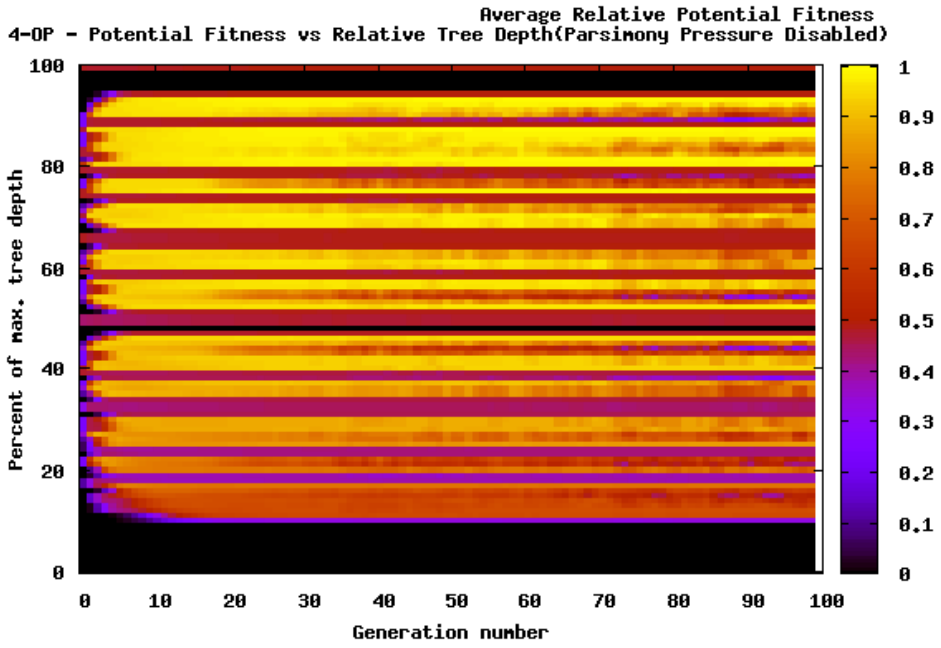

FIGURE 5.58: 4-Odd Parity without parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

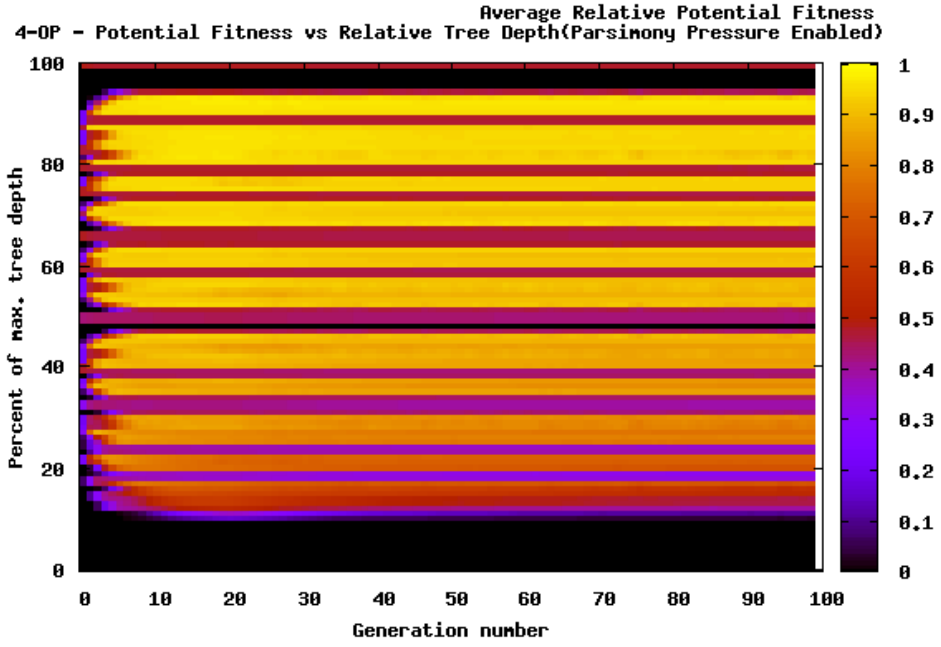

FIGURE 5.59: 4-Odd Parity with parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

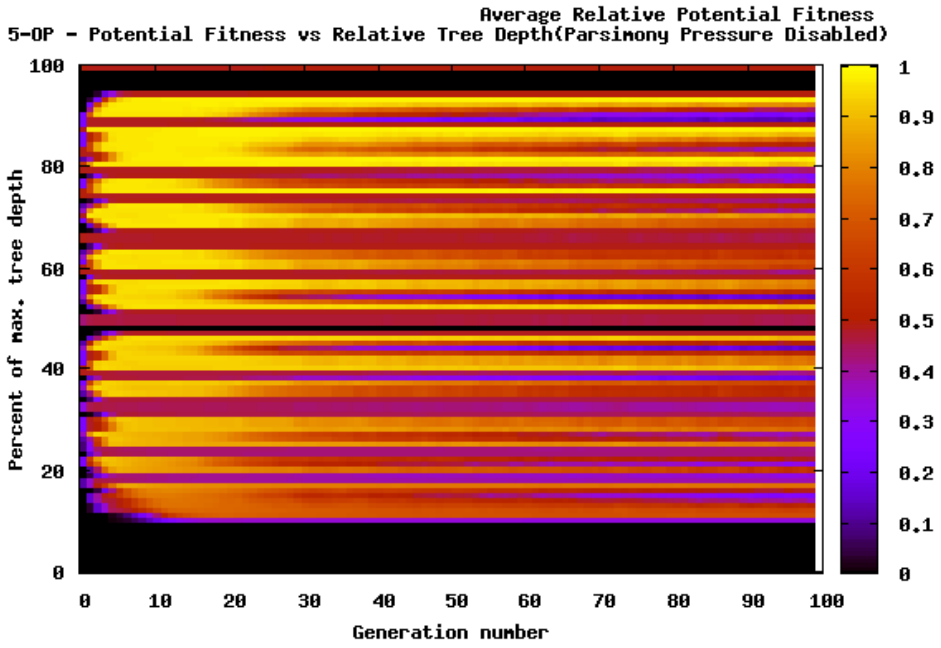

FIGURE 5.60: 5-Odd Parity without parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

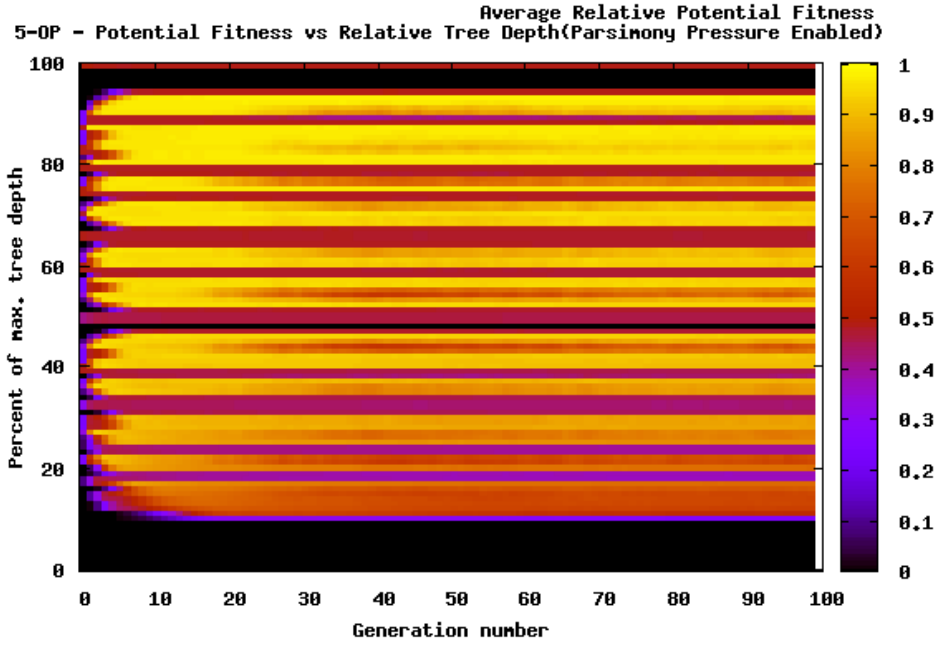

FIGURE 5.61: 5-Odd Parity with parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

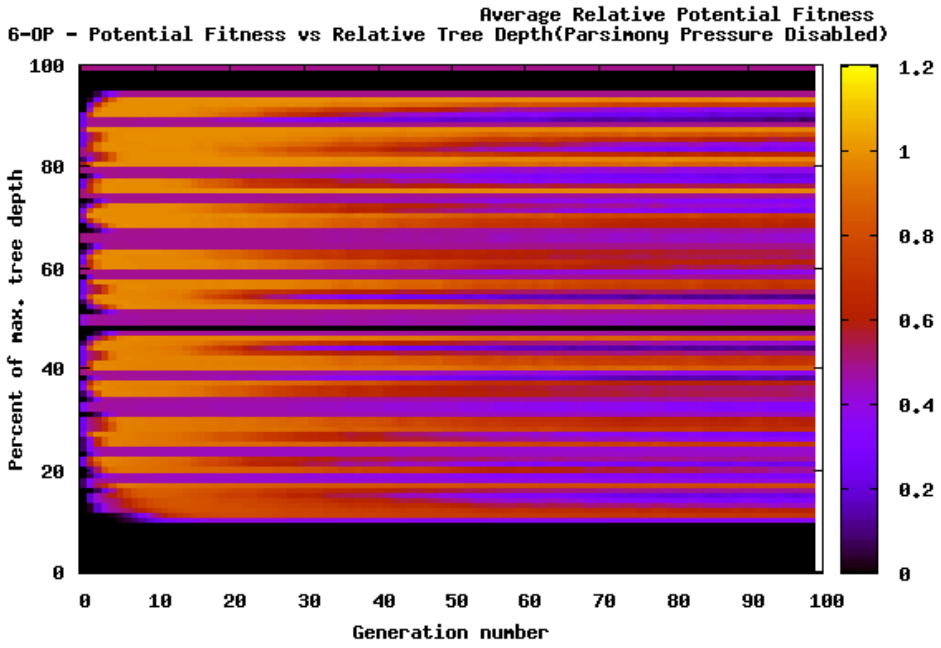

FIGURE 5.62: 6-Odd Parity without parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

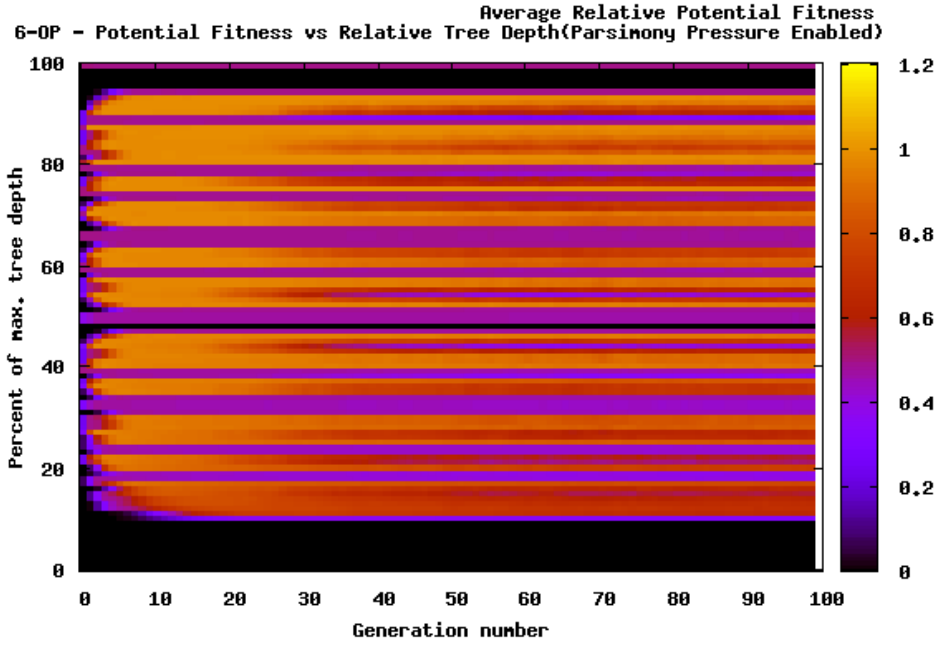

FIGURE 5.63: 6-Odd Parity with parsimony pressure. Spectrogram of Potential Fitness vs. Relative Depth in tree. Average from 100 independent runs.

Chapter 6

Conclusions

In this thesis, a new approach to evaluating individuals in Genetic Programming applied to boolean functions was studied. This approach, referred to as *Potential Fitness*, attempts to estimate the future fitness of the individual.

The method was implemented in the *Java* programming language and extensively investigated using the *Evolutionary Computations in Java (ECJ)* framework, and 7 benchmark boolean problems from the *Odd Parity, Comparator,* and *Multiplexer* families. The calculations were performed in two configurations: with or without the presence of lexicographic parsimony pressure.

The impact of using the Potential Fitness approach on the quality of the obtained solutions was studied and compared to the performance of standard Genetic Programming. It was determined that the new approach performs better in terms of success rate, average hit rate of best individual, and in some cases, average number of generation when the ideal individual is found, for the vast majority of the examined problems, both in the presence and absence of parsimony pressure.

A dedicated method for calculating the exact value of Potential Fitness for an individual was designed. It was found that it significantly outperforms the reference method of calculating Potential Fitness (through the definition of context semantics).

A randomized method for calculating an estimate of an individual's potential fitness was proposed. Its running times and various aspects of the solution quality were analyzed. It was determined that while it offers significantly lower running times than the exact method, the quality of the obtained results is seldom an improvement compared to standard GP. This suggests that it is difficult to find a heuristic algorithm which can yield an interesting tradeoff between solution quality and execution time.

The main goal of this thesis, which was the investigation of the properties of the Potential Fitness approach, and proposing a possibly efficient method of its calculation, has been achieved. Part of the results were published and presented at

the *Genetic and Evolutionary Computation Conference (GECCO) 2008* [24].

Further research

It seems that generalizing the Potential Fitness approach to other finite domains is difficult due to the dramatic rise in computational complexity. Also, for domains of higher order than *2*, the probability that the output value of a tree remains constant regardless of the subtree at a specific context decreases significantly, and therefore the number of fixed elements in the context semantics of a tree would be very low, effectively rendering the potential fitness useless.

It could be interesting to investigate another aspect of the evolutionary process. Perhaps if the crossover point of the tree were chosen with a probability of the potential fitness of the corresponding context (instead of a uniform probability for all nodes), the solution quality could be enhanced even further. This, however, would make it necessary to calculate the potential fitness of every node, which would make the use of Branch and Bound impossible, and thus cause the calculation times to increase significantly, perhaps beyond an acceptable threshold.

Bibliography

[1] A.E.Eiben, C.H.M. van Kemenande, and J.M.Kok. Orgy in the computer: Multi-parent reproduction in genetic algorithms. In *Advances in artificial life. Third international conference on international life*, 1995.

[2] Th. Back. Optimal mutation rates in genetic search. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, 1993.

[3] Th. Back, U. Hammel, and H.-P. Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1997.

[4] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming An Introduction. On the Automatic Evolution of Computer Programs and its Application.* Morgan Kaufmann, 1998.

[5] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Graduate School of Industrial Administration, 1976.

[6] D. Cliff. *Biologically-Inspired Computing Approaches To Cognitive Systems: a partial tour of the literature.* HP Digital Media Systems Laboratory, HP Laboratories Boston, 2003.

[7] S.A. Cook. The complexity of theorem proving procedures. In *Proceedings, Third Annual ACM Symposium on the Theory of Computing.* ACM, 1971.

[8] T. H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[9] N.L. Cramer. A representation for the adaptive generation of simple sequential programs. *Proceedings of an International Conference on Genetic Algorithms and the Applications, Grefenstette, John J. (ed.)*, 1985.

[10] W.G. Macready D.H. Wolpert. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1997.

[11] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, 1992.

[12] L.A. McGeoch D.S. Johnson. The traveling salesman problem: A case study in local optimization. *Local Search in Combinatorial Optimization*, 1995.

[13] C. Ferreira. *Gene Expression Programming: Mathematical Modelling by an Artificial Intelligence*. Springer, 2006.

[14] M. Laguna F.Glover. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[15] L. J. Fogel. Autonomous automata. *Industrial Research*, 1962.

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

[17] K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *DATALOGISKE SKRIFTER*, 1998.

[18] J.H. Holland. Outline for a logical theory of adaptive systems and other articles. *Essays on Cellular Automata*, 1970.

[19] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[20] R. Eberhart J. Kennedy. Particle swarm optimization. In *Proceedings, IEEE International Conference on Neural Networks*, 1995.

[21] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 1972.

[22] M. P. Vecchi Kirkpatrick S., C. D. Gelatt Jr. Optimization by simulated annealing. *J. Chem. Phys.*, 1983.

[23] J. R. Koza. A paradigm for genetically breeding populations of computer programs to solve problems. *Stanford University Computer Science Department technical report*, 1990.

[24] K. Krawiec and P. Polewski. Potential fitness for genetic programming. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, 2008. LNCS49740184.

[25] S. Luke and L. Panait. Lexicographic parsimony pressure. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of*

the Genetic and Evolutionary Computation Conference, pages 829–836, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[26] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. Semantic building blocks in genetic programming. In M. O'Neill, L. Vanneschi, S. Gustafson, A.I. Esparcia Alcázar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Genetic Programming*, volume 4971 of *LNCS*, pages 134–145. Springer, 2008.

[27] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 1953.

[28] H. Liu P. Ji, Y. Wu. A solution method for the quadratic assignment problem (qap). In *The Sixth International Symposium on Operations Research and Its Applications (ISORA06)*, 2006.

[29] R. Poli, W.B. Langdon, and N.F. Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[30] R. Poli and J. Page. Solving high-order boolean parity problems with smooth uniform crossover, sub-machine code gp and demes. *Genetic Programming and Evolvable Machines, 1(1/2)*, 2000.

[31] M. Queyranne. Performance ratio of heuristics for triangle inequality quadratic assignment problems. *Operations Research Letters*, 1986.

[32] I. Rechenberg. *Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Technical University Berlin, Berlin, Germany, 1970. (In German).

[33] J. Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook. Institut für Informatik, Technische Universität München, 1987.

[34] H.-P. Schwefel. *Numerische Optimierung von Computer–Modellen mittels der Evolutionsstrategie*. Birkhauser, Basel, 1977.

[35] S.F.Smith. *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, University of Pittsburgh, Pittsburgh, USA, 1980.

[36] R. Tadeusiewicz. Wsto sieci neuronowych. In *Biocybernetyka i inynieria biomedyczna 2000 - tom 6 Sieci Neuronowe*, 2000.