On an alternative approach to Evolutionary Programming

M.Tech Final Stage Report

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

by

Gautham Anil
gautham_anil@iitb.ac.in
Roll No: 04305013

under the guidance of

Prof. S. Krishna

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
Acknowledgments

I would like to thank my guide Prof. S. Krishna from the bottom of my heart for letting me do the M.Tech Project in this topic under her. I am also extremely thankful for her constant encouragement and invaluable guidance.

I also would like to thank my friends for their encouragement and criticism.
Contents

1 Introduction and Motivation .......................................................... 1
  1.1 Genetic Programming .................................................................... 1
      1.1.1 Disadvantages of mutation and crossover ............................. 2
      1.1.2 Bloat problem ................................................................. 2
 2 Literature Survey .............................................................................. 4
  2.1 Evolution of Evolutionary Algorithms ........................................... 4
      2.1.1 Experimental Results ......................................................... 4
  2.2 Estimation of Distribution Programming using Bayesian Network .... 5
      2.2.1 The basic EDP Algorithm .................................................... 5
      2.2.2 Probability Distribution Model .......................................... 5
      2.2.3 Program Generation .......................................................... 5
      2.2.4 Experimental Results ....................................................... 6
      2.2.5 Analysis ....................................................................... 6
  2.3 Estimation of Distribution Programming + GP ............................... 6
      2.3.1 Experimental Results ......................................................... 6
      2.3.2 Analysis ..................................................................... 7
  2.4 Techniques for Handling GP issues ............................................... 7
      2.4.1 Dynamic Maximum Tree Depth ........................................... 7
      2.4.2 Variations on Size and Depth ............................................. 8
      2.4.3 Analysis ..................................................................... 8
      2.4.4 Alternative bloat control strategies .................................... 8
      2.4.5 Plagues ..................................................................... 10

3 Design ............................................................................................ 12
  3.1 Formal Design ........................................................................... 12
  3.2 Working ................................................................................... 13
      3.2.1 The Controller ................................................................. 13
      3.2.2 Child Program ................................................................ 14

4 Implementation and Experimental Setup .......................................... 16
  4.1 Platform Details ....................................................................... 16
  4.2 Controller ................................................................................ 17
      4.2.1 Shell Listener .................................................................. 17
      4.2.2 Child Listener .................................................................. 17
      4.2.3 Main Message Queue ...................................................... 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.4</td>
<td>Controller Message Queue</td>
<td>17</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Credit Adjustment Function</td>
<td>17</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Global Settings</td>
<td>18</td>
</tr>
<tr>
<td>4.3</td>
<td>Search space and Mutator</td>
<td>19</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Symbol Space</td>
<td>19</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Syntax/Expansion Rules</td>
<td>20</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Mutator</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>Child Program and Evaluator (Child portion)</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluator (Controller portion)</td>
<td>22</td>
</tr>
<tr>
<td>4.6</td>
<td>Sample Problem</td>
<td>22</td>
</tr>
<tr>
<td>4.7</td>
<td>Seed Program</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Issues, Result and its Interpretation</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Issues faced</td>
<td>25</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Seed-Flatlining</td>
<td>25</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Long life</td>
<td>25</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Zero credit children</td>
<td>25</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Premature parent death</td>
<td>27</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Non-seed flatlining</td>
<td>27</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Miser</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Results</td>
<td>29</td>
</tr>
<tr>
<td>5.3</td>
<td>Interpretation of Result</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion and Future Work</td>
<td>32</td>
</tr>
<tr>
<td>6.1</td>
<td>Future Work</td>
<td>32</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Mutation Control</td>
<td>33</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Community Evolution</td>
<td>33</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Change Optimizations</td>
<td>33</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Other performance improvements</td>
<td>34</td>
</tr>
</tbody>
</table>
List of Figures

4.1 The mutation Algorithm ................................................. 21
4.2 Input to score mapping for sample problem (eid 1) .................. 23
4.3 Seed code .............................................................. 24

5.1 Successive credit adjustment steps with no child being created. ....... 26
5.2 Portions of Log showing zero credit children. ......................... 26
5.3 Portions of Log showing premature parent death. ..................... 27
5.4 Successive credit adjustment steps with no child being created. ....... 28
5.5 Section of the log showing misers in the populations .................. 28
5.6 Code of child with score 80 ............................................ 30

6.1 Future design diagram for a distributed system ....................... 35
Abstract

The field of Genetic Programming (GP) is a popular approach to automatic programming. However, it is plagued by many issues which prevent it from being scalable. These issues include bloat, need for custom search spaces etc.

We propose design for a system called Evolutionary Programming System or EPSys, a system in which we try to eliminate many problems plaguing Genetic Programming and hence preventing it from being scaled and being applied to larger real world problems. We then follow it up by a minimal implementation in which we solve a very simple problem. We then prove that this implementation is equivalent to GP. We also prove that this system is immune to bloat problem which has plagued GP since its inception. We also conclude that the current implementation is very much slower than GP due to huge overhead.

We then propose some basic improvements to EPSys which could potentially make it the far more scalable than GP.
Chapter 1

Introduction and Motivation

In the field of computer science, we come across many problems that needs to be solved. For example, the shortest path finding problem. A large number of these problems can be modeled as an optimization problem. An optimization problem is defined as one in which a solution is to be in the feasible region which has the minimum (or maximum) value of the objective function. Most of these problems can be solved using custom solvers created using algorithmic techniques. For example, Dijkstra’s method for shortest path finding problem.

However, if the solution can be represented as a string/list-of-symbols, most of these problems can also be solved using Genetic Algorithm. A genetic algorithm (GA) is a search technique used in computer science to find approximate solutions to optimization and search problems.

1.1 Genetic Programming

If the solutions to the problem is in fact program, then the technique is called Genetic Programming (GP). It uses an evolutionary algorithm to optimize a population of programs according to a fitness landscape determined by a program’s ability to perform a given computational task. It is quite similar to GA.

GP originated in the early 90s. The early landmark work on GP is commonly acknowledged to be John R Koza’s Genetic Programming: On the Programming of Computers by Means of Natural Selection. GP is very computationally intensive and so in the 1990s it was mainly used to solve relatively simple problems. However, more recently, thanks to various improvements in GP technology and to the well known exponential growth in CPU power, GP has started delivering a number of outstanding results. While this may be thought of as an encouraging development, we believe it also goes on to show the dependence of GP on computing power to scale and its inability to significantly improve otherwise despite one and a half decades of research. We will see how we can change this. First let us see the algorithm of GP.

The typical algorithm for GP can simply be as given below. However, note that the definition for GP allows for more leeway in changing the algorithm than it does for GA.

1. Generate an initial population of random compositions of the functions and terminals of the problem (computer programs).
2. Execute each program in the population and assign it a fitness value according to how well it solves the problem.

3. Create a new population of computer programs as follows.
   (a) Copy the best existing programs.
   (b) Create new computer programs by mutation.
   (c) Create new computer programs by crossover (sexual reproduction).

4. Return to step 2.

From the algorithm given above, we understand that GP is essentially two step performed on a population of programs over many generations viz., creation and selection. The former is seen in step 3 while the latter is performed in step 2.

### 1.1.1 Disadvantages of mutation and crossover

Let us examine the creation step closely. Other than copying, there are two procedures for creating new programs. These procedures take candidate solution program(s) as input and output a program (say, a child program). Due to the random nature of these procedures, there is no consideration of the semantics (if any) and logic in the program. In Koza’s implementation of GP, he designed the program syntax of the population in such a way that any combination of the symbols would lead to a valid program. It is true that this might avoid some syntactic even semantic errors prevalent in mutated and crossed over code. But it still does not solve the requirement of needing to create new children taking into consideration the logic of the input program.

If you think about this fact, you will realize that the limitation is in the ability of the child-creation procedures to program. To do incremental modifications to a program, some amount of intelligence is required and the mutation and crossover procedures don’t have any. In [10] we see rudimentary artificial intelligence in the form of bayesian networks used to create new members of the population in an effort to make this process of creation more efficient at generating valid code. We see that while the effort is laudable and the resulting performance improvement significant, it is not substantial and underlines the need for better strategies for child creation.

### 1.1.2 Bloat problem

Bloat is a problem prevalent in GP. It is a result of the random nature of child code creation mentioned in the previous subsection. Due to lack of understanding of logic, the created code sometimes include unnecessary computations and infinite loops which take up unnecessary CPU time during execution. Since the candidate can be evaluated only once it has been executed fully, this leads to extreme slowdown and sometimes virtual halting of the GP run. While many solutions have been suggested to solve this problem some of which have been reviewed in chapter 2, none of these tackle the fundamental reason bloat exists. And that is because GP does not treat time as a scarce resource.

---

1This technique is reviewed in Section 2.2
It instead treats fitness evaluations as a scarce resource and efficiently allocates them to better programs. This is done as an approximation to reducing time with the expectation that lesser fitness evaluations equal lesser time. The analogy fails in the case of bloat which stretches the time taken to evaluate a single candidate. To tackle this problem, we need to develop a new design in which the time, not evaluations is taken as a scarce resource.

In Chapter 3, we present the design of a system christened Evolutionary Programming System or EPSys. In this system, we have tackled the problems mentioned above. We also list other possible advantages of the system.

In chapter 4, we explain a feature reduced implementation of the design that was done. This is followed, in chapter 5, by the results of running the program and the conclusions that can be drawn from the results.

In the last chapter, as future work, we present mutation control and community optimization as a combination of techniques that could lead to significant speedup of the system. We also present “change optimizations”, a training strategy that could improve the performance of a full implementation of the design even further. We also suggest how to convert EPSys into a distributed system that can give almost linear speedup.
Chapter 2

Literature Survey

Let us now look at some of the existing literature on the area of evolutionary programming and Genetic Algorithms.

2.1 Evolution of Evolutionary Algorithms

A genetic algorithm run has many parameters that affect its speed and quality of result. In most cases the scientist hand optimizes them using previous experiences. In [1], we see other automatic ways for doing the same.

This field which focuses on the evolution of specific parameters of EC algorithms is called Adaptive Evolutionary Computation (AEC). This includes algorithms which permit variation in the gene size, representation of genotype etc.

In this paper, we study the evolution of the genetic operators involved in GP. There are two approaches. In the first, we evaluate the fitness of each evolutionary operator using a sub GP which will be run every time one is to be evaluated. This is quite inefficient considering the time requirements of a GP run.

In the second, we combine the genome of the operator set and the genome of the individual. For evaluating, we decode (convert to phenotype) the operators first and then apply the operators on the phenotype of the individual.

This is simple if the operators work in a single individual. Operators on more than one individuals (sexual recombination) pose problems. For sexual recombination, we can take operator from parent1 and apply on both parents, take the operator from parent2 and apply on both parents leading to creation of two children.

2.1.1 Experimental Results

In experimental results, it was found that this technique of evolved operators pushed up the peak fitness of the population over time far above the GA with default operators. It comes quite close to the fitness provided by the GA with optimal operators.

On the other hand, the average fitness of the GA remains consistently far below both optional and default GAs. This says that the improvement in performance is due to the maintenance of phenotypic diversity leading prevention of stagnation.
2.2 Estimation of Distribution Programming using Bayesian Network

Let us now look at an alternative strategy of basing evolutionary algorithms on a probability model. These are called Estimation of Distribution Algorithms (EDA) [5]. In this paper [10], EDA is extended to solve problem handles by conventional GP. This is then called Estimation of Distribution Programming (EDP). The probability model used here is the bayesian network.

2.2.1 The basic EDP Algorithm

The basic EDP algorithm can be given as follows.

1. Initialize a population.
2. Evaluate individuals
3. Estimate the distribution
4. If a termination criterion is met, then go to Step 7.
5. Generate a new population
6. Replace the population and go to Step 2.
7. Report the best individual.

2.2.2 Probability Distribution Model

Here, each node in the program tree represents a probabilistic variable (random variable). This random variable is assumed to be dependent on other random variables (other nodes). The process captures the dependency using a bayesian network (see paper [10] for details). Also, it is assumed that there will be no loops.

2.2.3 Program Generation

For program generation, elitism is followed. So, some of the best individuals from the parent population is carried over to the child population. The remaining population has to be generated. They are generated one by one.

In the generated bayesian network, some nodes will be independent (this because of the assumption that there will be no loops). So, to generate a child, first, these nodes are given values based on their probability. Then, all the nodes that depend on these generated values alone are also generated and so on till all the nodes have values. This leads to a single child (note that each time a child is created, it might be different from the previous because the bayesian network gives just the probability.).

This way children are created till the required population size is achieved.
2.2.4 Experimental Results

Some experiments were performed using EDP to compare with GP. The conclusion suggests that some problems favour the EDP approach while some do not. For e.g., the maximization problem quickly reached optimum for EDP compared to GP. On the other hand the boolean function problem showed no preference for either of the techniques.

2.2.5 Analysis

It is an interesting approach. But, its success is clearly too elusive for usefulness. However, its partial success emphasizes the importance of the decision to leave the choice of choosing mutation in the evolutionary system to the programs.

2.3 Estimation of Distribution Programming + GP

In [11], an effort is made to combine EDP and GP. This is tried as an improvement over [10]. This paper recommends that GP and EDP be combined together to get a single algorithm so that, some of the disadvantages of EDP could be eliminated.

The combined algorithms go as follows.

1. Initialize a population
2. Evaluate individuals and assign fitness values.
3. If a termination criterion is satisfied, then go to step 9.
4. Estimate the probability distribution.
5. Generate new $rM - Es$ individuals with GP operator.
6. Generate new $(1 - r)M$ individuals with EDP operator.
7. Replace population and go to step 2;
8. Report the best individual.

As it can be seen, the population is generated partly with GP operators and partly with EDP operators. This way, if EDP generates good children, it has an effect on the next generation. Other wise, only GP children make it through.

2.3.1 Experimental Results

It is found that the hybrid system $(r = 0.5)$ is provably (Welsh’s test) superior to pure GP $(r = 1.0)$. But it was also noted that the EDP rarely produced children that were good solutions. It appears that EDP’s role is just to enhance the role of GP.

Also, some ways of changing $r$ was tried out to see if an optimum could be achieved. It was found that increasing $r$ linearly as the generations progressed provided the best performance compared to other schemes. This suggests that EDP functions well in early generations.
2.3.2 Analysis

It might be a good idea to provide EDP based child generation algorithm in the evolutionary system so that, if found better than mutation in some cases, it could be used to speedup.

2.4 Techniques for Handling GP issues

Let us now look at ways to handle common Genetic Programming Issues.

2.4.1 Dynamic Maximum Tree Depth

Introns and the Problem of Bloat

Genetic Programming (GP) solves complex problems by evolving populations of computer programs, using Darwinian evolution and Mendelevian genetics as inspiration. The search space is infinite and the programs may grow in size during the evolutionary process. The code growth is a healthy result of genetic operators in search of better results.

But this also permits the appearance of pieces of redundant code called introns which increases the size of the programs without improving their fitness. These introns use up computational resources too and if unchecked, could develop to a degree serious enough to stagnate the evolution, preventing the algorithm from finding better solutions.

Dynamic Maximum Tree Depth

Koza [4] introduced strict limits on tree depth achieving a moderate level of success in tackling bloat. Dynamic Maximum Tree Depth (DMTD) [8] is similar to strict limits except in two aspects: it is initially set with a low value; it is increased when needed to accommodate an individual which is deeper than the limit but is better than any other individual found during the run.

Practically, strict Koza limit and the dynamic limit go hand in hand. Let us see the technique in detail.

- If the new individual is deeper than the strict Koza depth limit, reject it and consider one of its parents for the new generation instead.

- if the new individual is no deeper than the dynamic depth limit, consider it acceptable to participate in the new generation

- if the new individual is deeper than the dynamic limit (but no deeper than the strict Koza limit) measure its fitness and

  - if the individual has better fitness than the current best individual of the run, increase the dynamic limit to match the depth of the individual and consider it acceptable to the new generation

  - if the individual is not better than the current best of run, leave the dynamic level unchanged and reject the individual, considering one of its parents for the new generation instead.
Note that once increased, the dynamic limit will not be lowered again.

Experiments were conducted for symbolic regression problem and even-3 parity problem. In most experiments, the diversity of the population was seen to be going down. However, this did not seem to affect the quality of the result. For symbolic regression, the DMTD proved clearly better than lexicographic parsimony pressure. For even-3, a combination of DMTD and lexicographic parsimony pressure proved better than any of them individually.

2.4.2 Variations on Size and Depth

[9] presents two variations on the Dynamic Maximum Tree Depth technique explained in the previous section.

1. Heavy Tree depth

In this technique, the DMTD is augmented. While originally, the limit was never permitted to decrease, now the depth can decrease if the best fit individual falls below the limit.

Experimental results show that this technique does result in speed up. This is partly because of loss of diversity in the population. However, it did not lead to loss of fitness in the final solution.

2. Dynamic Maximum Tree Size

This technique is a simple variation of DMTD. Instead of limiting by depth, trees are now limited by node count. i.e., size.

This technique on the other hand had poor experimental results. Though it was able to successfully control bloat, the resultant solution was not as good as what was required. This means that search space has been limited in a way that it prevents sufficiently thorough search.

2.4.3 Analysis

It is interesting to note that again, loss of diversity in a population does not necessarily cause inability to converge to a solution of quality comparable to what a search with no loss of diversity throws up.

Also, the strict restriction of search space leading to poor results is also interesting. It means that though one may remove portions of an infinite search space, the resultant search space should still be infinite. One should not evolve in a finite search space. This could suggest that for practical problems, it would be difficult finding an upper bound for search time since quality solutions can come only by searching infinite spaces.

2.4.4 Alternative bloat control strategies

In [7], the authors tackle the problem of bloat. The paper first provides a review of the existing bloat control measures.
Existing Bloat Control Techniques

The most common approach to bloat control is establishing a hard limit on size or depth. This method has been criticized in the past for its effect on breeding \[3\]. But has proved to be a surprisingly successful method \[6\].

Outside of GP, a common alternative is \textit{Parsimony Pressure}, which includes the size of an individual as a factor in the selection procedure. There are two variations.

If it is \textit{parametric}, then the size and fitness is combines to create a selection criteria based on which the selected step is performed. Here, the programmer has to decide that so much size can be traded against so much fitness.

Recent focus has been on non-parametric parsimony techniques. One technique is \textit{lexicographic parsimony pressure}. Here, the programs are ordered by fitness. If there is a tie, they are ordered by increasing size.

If the grain of the fitness value is fine and chance of two individuals having the same fitness is very small, then the \textit{bucketing} technique is used. Here, the fitness range is split up into buckets. If two individuals fall into the same fitness bucket, they are ordered by size.

In \textit{double tournament}, the programs are first selected based on small size. Then from this set, they are selected for best fitness.

Some literature handle size as an alternative objective in a \textit{pareto-optimization} scheme. In a pareto scheme, individual $A$ is said to \textit{dominate} individual $B$ if $A$ is as good as $B$ in all objectives (fitness, size) and is better than $B$ in at least one objective. This family of methods use one of several multi-objective optimization algorithms to discover the “front” of solutions which are dominated by no one else. The literature has had mixed results, because non-dominated individuals tend to cluster near the front extremes of all-fitness or all-size both of which is undesirable.

Three new Bloat control methods

This paper \[7\] then introduces three bloat control methods and presents the experimental results. The experiments were conducted in 4 domains. Artificial ant, symbolic regression, 11-bit boolean multiplexer and even 5-parity.

1. \textbf{Biased Multi-objective Parsimony Pressure (BMOPP)}

This technique is a variation of the technique stated above. One of the problems with pareto-based multi-objective methods is that they consider any point along the front to be a valid candidate. This enables the individuals in the (low size, low fitness) to survive. Generally, this is not an interesting extreme. In BMOPP, the search in the front is biased towards the fitness end.

In this technique, the programs are arranged in to \textit{pareto layers}. First the non dominated front of the existing programs is calculated, removed and placed in layer 1. The next front is calculated from the remaining programs and placed in layer 2 and so on.

Once all layers are calculated, individuals are selected using a form of tournament selection which, with probability $P$, compares individuals solely based on their fit-
ness, and with probability $1 - P$ compares them based on their respective pareto layers (lower layers being preferred). Ties are broken with alternative comparison. If tie remains, one is chosen at random.

If $P = 1$, BMOPP is lexicographic tournament. If $P = 0$, BMOPP is entirely pareto-based.

In the experiments conducted, it was found that for $P = 1.9$, the bloat was significantly reduced without affecting fitness in all problem domains.

2. The Waiting Room

The idea of waiting room is that smaller individuals get to compete in the evolutionary process sooner.

In the waiting room, newly created individuals are punished for being large by being placed in a queue prior to entry into the population. Let the population size be $N$. At each generation, some $RN$ newly-created individuals are evaluated, then added to the waiting room. $R > 1$, and so, the waiting room will be larger than the final population size. Each individual in the waiting room is assigned a queue value equal to the individual’s size. Next, $N$ children with the smallest queue values are removed from the waiting room and form the next generation. The remaining individuals have their queue values multiplied by a cut-down value $A$ between 0 and 1. $A$ prevents queue stagnation: eventually even giant individuals may have a chance to be introduced to the population.

In the experiments conducted, it appears that a small pressure for parsimony significantly reduces the mean tree size of individuals. The setting $R = 1.125$, $A = 3$ appears to perform well across all four problems.

3. Death by Size

This scheme is intended for methods such as steady-state evolution which require a procedure for marking individuals for death. Death by size is very simple: use fitness to select individuals for breeding ($S$ is the size of the tournament), prefer to kill larger individuals ($R$ is the size of the tournament).

The experiments suggest that except for multiplexer domain this technique showed had many combination of $S$ and $R$ that provided similar fitness with smaller average tree size.

2.4.5 Plagues

In [2], the author tests the abilities of a new heuristic/operator in handling bloat. While previous techniques concentrated on multi-objective evolution for reducing the size of the individuals, this system concentrates on reducing the size of the population as a whole while leaving the individual alone.

This is done by reducing the population size as the generations are computed. For this some individuals are cyclically removed. The individuals being removed are being selected either by nature of their fitness or size.
This way, though the programs individually does not improve, overall time taken per generation remains constant. This leads to the speed up.

The author proceeds to present experimental results of using this genetic operator. The conclusion is that the plague operator leads to overall reduction in computation time. In addition, the author agrees that this claim is valid if quality of results obtained is similar to that of constant-population-size GP as it was in the experiments performed.

As a disadvantage, the author notes that the phenotypic and genotypic diversity is reduced as the population size is reduced. This makes the constant-population-size GP superior in quality for constant generation.

Analysis

We accept the fact that this is a workable heuristic. In some type of problems, while evolving, maintaining large populations is found favorable, while for some others, maintaining small populations is found to lead to a quick solution. Here, we start with a large population and as the search progresses, reduce the population size. It is possible that for some problems, this strategy might be better than either of the former.

But an interesting observation is that this heuristic does not solve the problem of bloat. It just compensates for it by reducing genetic diversity. The disadvantages of bloat i.e., survival of programs that do calculations whose results are not subsequently used, still remains.
Chapter 3

Design

Let us now see the design of the Evolutionary Programming System. While the design is being presented in detail, very little reason is given for the current form of the design. But the conclusions in chapter 6 will help you get some idea about the same.

3.1 Formal Design

The system EPSys can be denoted by the tuple \((P, T, E)\). Here, \(P\) denotes the set of members of the population that are being evolved. \(L\) denotes the information regarding the search space of the solution. \(E\) consists of a set of functions that represent the real world problem we are solving.

Since this system is used to do Evolutionary Programming, \(L\) must define a programming language. Thus, \(L\) consists of \(L_S\), the set of symbols, \(L_R\) the set of syntax rules and \(L_E\) the simulation/execution logic for the language which is a closed function that can take a program satisfying \(L_S\) and \(L_R\) are output a result in the same space. In the ideal case, it is required that \(L\) must define a Turing complete programming language. However, depending upon the requirements of the problem\(^1\) whose solution is being attempted, any programming language may be used in implementation.

Members of \(E\) (say, \(E_j\)) are analogous to the fitness function in GP. In GP, the loop executes the candidate program with the appropriate inputs (if any) and gives the result to the fitness function. The fitness function then analyses the result and supplies a score. Here, the job of \(E_j\) is to supply information about the real world problem to \(P_i\) and evaluate the ability of \(P_j\) to solve it. \(E_j\) may be defined as a function that performs the various functions necessary for it to do depending upon the input. For example, if the problem is simple and does not require any input or additional information to be supplied to \(P_i\), then it can be implemented just as a fitness function.

A value \(E_{Rj}\) is associated with \(E_j\). It denotes the relative importance of \(E_j\) among all members of \(E\). So, \(\sum_{E_j \in E} E_{Rj} = 1\). \(E_j\) is also associated with a distribution function \(E_{Dj}\) which, given all \(S_{ij}\), returns \(D_{ij}\) such that \(\sum_{P_i \in P} D_{ij} = 1\). \(D_{ij}\) denotes the relative score \(^2\)

\(^1\)Turing completeness is required in the search space to prove that all possible optimizations could be emergent in the system. In cases where all emergent optimizations are of no concern, such as for the goals of the implementation explained in the next chapter, this requirement could be relaxed.

\(^2\)Described later.
of $P_i$ among all $P$ as evaluated by $E_j$.

$P_i$ are the programs that are being evolved in this system. These are analogous to the programs evolved in GP. A value $c_i$ is associated with every $P_i$. This is called the credit possessed by $P_i$ and determines the ability of $P_i$ to survive. It is the amount of unused units of CPU time that $P_i$ has remaining. Also $W_i$, a member of $L$ is also associated with $P_i$. It stores the code denoted by $P_i$. It may also store any additional meta information needed. $W_i$ is executed (in a predefined environment if necessary) as a separate process when $P_i$ is created unlike in GP where programs are executed one after the other.

A value $S_{ij}$ is also defined which contains the score achieved by $P_i$ on being evaluated by $E_j$.

As part of the system, a handwritten $W_S$ is defined that denotes the seed for the system. This is the code corresponding to the initial population of the system but has to be inserted into a fixed wrapper code.

In addition to this, apart from the suggested design, a mutator $M$ might be necessary to get EPSys started. While ideally, $P_i$ should be able to create new children all by itself, initially, it might be preferable to give the access to a simple mutator. This mutator $M$ is a closed function in $L$. It is also closed in the meta information (if any) associated with $W_i$.

### 3.2 Working

The work is basically done in the child programs executing in parallel and the controller. Let us see the working of the controller.

#### 3.2.1 The Controller

The controller runs a credit adjustment function during periodic intervals. The algorithm for the same is as given below.

1. Initialize the system.

2. Repeat the statements give below till exit command is given or exit condition is satisfied.

   (a) Make any requested and valid transfer of credit from one program to another. This involves reduction of $c_m$ and increasing of $c_n$ by the amount requested by $P_m$ after taking any tax levied. (Optional Step)

   (b) If $T_{it}$ is the cpu time spend by $P_i$ till the $t^{th}$ iteration of these steps, and $\Delta T_{it}$ the time spent by $P_i$ between $t^{th}$ and $(t-1)^{th}$ iteration, then $\Delta T_{it} = T_{it} - T_{i(t-1)}$. So, find out $T_{it}$, fetch $T_{i(t-1)}$ and calculate $\Delta T_{it}$.

   (c) Set $S_i = S_i - \Delta T_{it}$;

   (d) Let $X_{it}$ be the sum of all the taxed levied on $P_i$ between iteration $(t - 1)$ and $t$. Levying tax involves reducing $c_i$ and increasing $X_{it}$ by the same amount for various reasons.

   (e) Let $C_t = \sum_{P_i \in P} (\Delta T_{it} + X_{it})$. 

13
(f) Adjust $C_t$ for population control and any other implementation specific needs.

(g) Calculate all $D_{ij}$ (using all $E_{ij}$).

(h) Let $\forall P_i \in P \Delta S_i = \sum_{E_j \in E} C_t \cdot E_{ij} \cdot D_{ij}$

(i) Set $\forall P_i \in P S_i = S_i + \Delta S_i$.

(j) For all $P_i$ in $P$, if $S_i < 0$ terminate $P_i$;

(k) Create seed children as necessary. The actual number of children created is not being specified in design and may be implementation specific. Recommended number of seed children is $POPULATION_{TARGET} - POPULATION_{CURRENT}$

Understand that in the above step, the credits extracted from the programs in the interval as taxes and for the utilization of resources is given back to the programs based on their scores after adjusting it for population control etc.

In addition to the credit adjustment step, The controller maintains all the necessary information about the child programs running in the system and much more. The controller contains facilities for communicating with child programs and receiving the messages and executing them in a thread safe manner so that they may reflect on the information database. These messages include those about the score suggested by the evaluator in the child program.

These messages may also include messages from the child requesting the controller to create a new child and may pass necessary information required (such as possibly the code of the child) by the controller to do so. To process this message, the controller uses the information to create a $W_i$ and spawns a new $P_i$ that has $W_i$ associated with it. In case $M$ is implemented, the messages may not contain information necessary to create a new child. Instead it may contain some information which $M$ can and will use to mutate the $W_p$ where $P_p$ is the parent making the request.

Let us now see the working of the child program.

### 3.2.2 Child Program

The main purpose of the child program $P_i$ is to execute $W_i$ associated with it. For this purpose, it contains fixed code that wraps around $W_i$. This code contacts the controller on startup, sets up communication facilities, defines the required support functions and executes $W_i$. Since $W_i$ keeps on changing, due to mutation, it is impossible to predict what it is expected to perform when it is run. So, we shall explain the recommended functionality of $W_S$.

When $W_S$ is run after the wrapper code has been executed, it is expected to first get itself evaluated by a simple evaluator to get a score to survive and then in a loop, create children in periodic intervals.

When $W_S$ requests evaluation and communicates with the evaluator, which must be part of the process $W_S$ is running on, in an implementation specific way, the evaluator evaluates $W_S$ based on the contents of the communication. It then sends the information regarding the score to the controller using the communication facilities set up by the wrapper code.
Note that in this chapter, the design has been left purposefully vague about certain aspects of EPSys. This was only done regarding features where exact specification was unnecessary and multiple choices could be made regarding that feature depending upon usage requirements and still satisfy the minimum requirements for the system to function. The reason for this is to provide freedom of choice in implementation. Let us now proceed to the same.
Chapter 4
Implementation and Experimental Setup

The potential advantages that were aimed for while EPSys was designed were many. However, due to time restrictions, implementation was limited and most of the advantages could not be implemented. The only significant advantage that this implementation has over GP is its immunity to bloat. And the implementation was modified in places to aim for this advantage only. This focus is also reflected in the interpretation of the result.

4.1 Platform Details

Implementation of EPSys was done in scheme programming language using scsh on a GNU/Linux system (Fedora Core 3 distribution). Version of scsh used was 0.6.6.

The choice of scheme as a programming language was motivated primarily by the convenience of generating and modifying domain/problem independent scheme code. A programming language such as java has complicated syntax rules, consists predominantly of user defined data types and has a large amount of statement level interdependence leading to difficulty in creating meaningful program without significant effort.

On the other hand scheme, can be stripped further down to something close to lambda calculus with data types thus greatly simplifying syntax. This makes it possible to give much lower interdependence between expressions making the job of mutating code far more easier and effective are generating meaningful code.

Experiments were run on the following configuration

- Intel(R) Pentium(R) 4 CPU 3.00GHz with Hyperthreading (2 logical cores)
- 512MB RAM
- 80GB HDD
4.2 Controller

The controller is implemented as a separate process. It contains two message queues, two separate TCP/IP servers that are listening on different ports.

4.2.1 Shell Listener

This TCP/IP server listener listens on a port which is written down in a predetermined file. Clients for this server are consoles started by the user. The clients read the file and connect to the server and drops the user to a shell. From here, the users types scheme code which is sent to the server and executed thus affecting the function of the server.

4.2.2 Child Listener

This TCP/IP server is identical to the previous. Here, the file is different and the clients are \( P_i \). Like above, messages send by the \( P_i \) is executed in the controller. Thus it is important that in the wrapper of \( P_i \), there is no possibility of \( W_i \) sending arbitrary messages to the controller since it will make it possible to sabotage the controller’s restrictions.

4.2.3 Main Message Queue

This message queue is used to execute the messages received from all the child programs in a serial fashion. Since the messages from child are executed by the child listener, the messages are such that their execution will lead to the actual messages being inserted into the main message queue.

It may be argued that this is not completely necessary and CMQ will do the job all alone. It is a relic of an earlier version of the implementation and it makes debugging less tough.

4.2.4 Controller Message Queue

This works similar to the main message queue. However, the aim is to make sure the messages manipulating the information stored in the controller is executed in a serial fashion avoiding synchronization issues.

They key difference between CMQ and MMQ is that CMQ’s loop has a modification which makes it check before processing a message. It checks whether a fixed amount of time has passed since the credit adjustment function was called the last time and if the time has passed, calls it.

4.2.5 Credit Adjustment Function

The implementation of the Credit Adjustment function is according to design. The population control was done using this formula.

\[
\text{If } POPULATION_{\text{CURRENT}} > POPULATION_{\text{TARGET}} \\
\text{Then } C_t = C_t \times \left( \frac{POPULATION_{\text{TARGET}}}{POPULATION_{\text{CURRENT}}} \right) \\
\text{Else } C_t = C_t \times (2 - \frac{POPULATION_{\text{CURRENT}}}{POPULATION_{\text{TARGET}}}).
\]
In addition, the seed children introduction policy is as recommended in design. The Controller also contains the optional mutator $M$ which is used in the implementation.

### 4.2.6 Global Settings

There are many other settings that govern the functioning of the system. As this experiment went through its many trials, some settings were changed while others were removed entirely or new ones added. Given below is the final global settings used to run the experiment.

**Mutation related**

**available-thunk-use-probability (.5)** During mutation, after a sub-tree has been chosen for mutation, the probability with which that sub-tree will be used as the thunk needed.

**environment-selection-probability (.5)** In the situation described above, having decided not to use or can not use the existing sub-tree, the probability with which an environmental variable is chosen against other choice(s) (viz., among the expansion rules).

**thunk-depth-limit (10)** This value limits the maximum depth of the random code created.

**Controller related**

**population-size-target (7)** The population at which the credits debited from the children equal the credits credited to them during a credit adjustment step.

**poll-interval (500)** Time in milliseconds between successive credit adjustment steps.

**new-child-per-poll (0)** The minimum number of children created during a credit adjustment step. A value of 0 means seed children will be created only if population falls below population-size-target.

**Credit related**

**new-child-starting-energy (1)** Default extra starting credits for new seed children. In case of mutated children, this value is determined by the parent. Seed children, when being a parent, uses the same value for its children. Minimum extra starting credits is 0.

**new-child-minimum-starting-credit (150)** Default base starting credits for all children being created. This is a system wide fixed value and cannot be changed.

**child-create-tax (10)** Additional tax for creating children. This is debited when the child is created and goes to the credit pool of the controller to be distributed during the next credit adjustment step.
The three values given above determine the speed at which the system performs search. A child can be created only if the parent has this much credits to spend. So, a parent has to wait and the waiting time is directly proportional to the sum of the above values. Thus these values are a limiting factor which reduce the speed as they increase. On the other hand, if the first two values were reduced, the children will have low starting energies. Irrespective of their merits, they would be so much in debt that they would not be able to survive the next credit adjustment step. Thus it is important that the values are not too low either.

**default-survival-tax (50)** This tax is debited from all members of the population during each credit adjustment step.

**anti-flatline-bonus-multiplier (1.3)** This parameter affects measures taken to escape out of flatlining especially among non seed children. See section 5.1.5.

**anti-flatline-bonus (1)** Same as above.

**minimum-savings (50)** This is the minimum expected remaining credits of a parent. if a parent gives a mutation command, it will be processes only if after processing it, the parent has at least this much credit remaining.

**maximum-savings (800)** This is the maximum amount of credits a child is supposed to have. If a child is found to have credits exceeding this during a credit adjustment step, it is removed from the system.

### 4.3 Search space and Mutator

The search space for the solution is designed to be similar in syntax to $\lambda$-calculus. However, since the requirements from the mutator is very limited, given the sample problem, no effort is made to prove this is computationally same as $\lambda$-calculus. Some embellishments to $\lambda$-calculus was found to be necessary for speed. This includes numbers and list along with scheme functions to manipulate them.

#### 4.3.1 Symbol Space

The symbol space $L_S$ is as follows.

\[
\{ \#t, \#f, '() 0, 1, and, or, not, cons, cdr, car, if, gt, eq, add, sub, mul, quo, div, \Lambda, \text{Applyg}, \text{Apply-lazy}, \text{gensym}_{[0-9]+}, \text{get-eval}, \text{mutate1}, \text{sleep1}, \text{get-data} \}
\]

Most of the functions and symbols are obvious in function. Those requiring special mention are

**Lambda** This is the function that can generate new functions. Takes the parameter name \[^1\] and function body which must be a function call or an application. See example below.

\[
(Lambda \ (gensym_0) \ (+ \ 1 \ (gensym_0))) \Rightarrow \ (thunk)
\]

---

[^1]: See $\text{gensym}_{[0-9]+}$
Applyg  This function works exactly like scheme’s apply function. See example below.

```
(Applyg (Lambda (gensym_0) (+ 1 (gensym_0))) 1) => (integer)
```

Apply-lazy  This function works exactly like Applyg except that the parameter is evaluated only when it is first used. In the example below, gensym_0 is not evaluated.

```
(Applyg (Lambda (gensym_0) (+ 1 0)) 1) => (integer)
```

gensym_[0-9]+  This consist of the dynamically generated symbols generated as parameters in Lambda. They start from gensym_0 and continues to gensym_1 and so on.

g-get-eval, mutate1, sleep1, get-data  These functions are introduced by the wrapper code for $W_1$. get-eval fetches the appropriate evaluator. mutate1 calls the mutator in the controller. sleep1 sleeps for a the given number of milliseconds. get-data fetches the current information regarding the child from the controller.

### 4.3.2 Syntax/Expansion Rules

Due to the introduction of the above functions to the language, type safety becomes an issue. So, for $L_R$, type rules were written in addition to $\lambda$-calculus reduction rules to specify how to change or create code in $L$ maintaining type safety. The mutator uses these type rules for changing code. Refer to the source code\(^2\) for all the type rules. An Example is given below.

```
(number? (if (boolean? number? number?)))
```

This indicates that if a number is required, if can be used with three parameter of appropriate types.

As for $L_E$, it is same as the scheme interpreter.

### 4.3.3 Mutator

The mutator accepts a type enhanced version of the code to be mutated. See code in typed-node.scm and typed-program for exact data structures. The algorithm is as in Figure 4.1.

The function called by CMQ requesting for mutation, calls the mutator with the original code. It then wraps the mutated code in the wrapper code and spawns it using the fork system call in scheme. It also makes the necessary adjustments in the information storage in the controller regarding the newly spawned child program.

### 4.4 Child Program and Evaluator (Child portion)

The child program, as in the design, consists of mutated code with wrapper code around it. Since mutated code is unpredictable and seed code is explained in section 4.7, let us look into the wrapper code.

\(^2\)See file mut/types.scm
Procedure mutate (Typed-program tp)

1. Make a copy of tp called ctp.
2. Get the size of tp. Pick a random number posn between 0 and size.
3. Find the sub-tree stp in tp corresponding to posn.
4. Generate random code ntp giving (type of stp, false, stp).
5. Put ntp in stp’s place in ctp.
6. return ctp;

Procedure random-code (Type t, bool use, Typed-program tp)

1. if use and type of tp is same as t return tp.
2. if randomly choosing to use a terminal and if a terminal term of type t could be found, make a typed program of term and return it.
3. Randomly choose (based on probabilities mentioned before) a rule r from group of t.
4. Generate sub-trees for parameters by recursing (t, true, tp). Use the results and the rule to make a new typed-program. Return it.

Figure 4.1: The mutation Algorithm
The wrapper code as soon as it starts, connects to the controller’s listener and informs the controller that it has started. It then initializes the environment for \( W_i \). It then runs \( W_i \) in the initialized environment. It also informs the controller when and if the code returns along with the return value of the code that was run (for debugging purposes). It also includes traps for system signals such as TERM and KILL.

It also includes the evaluating portion of the evaluator. Here, the evaluator is implemented simply as a fitness function which is described in section 4.6. The fitness function can be called from \( W_i \) through \texttt{get-eval}. The parameters passed during the call determine the score that will be assigned to \( P_i \). The score is sent to the controller using the TCP/IP connection established by the wrapper code when this child program was started.

### 4.5 Evaluator (Controller portion)

The portions of the evaluator that is part of the controller remembers the scores that was sent to it by the child. The controller contains \( Ed_j \) corresponding to every \( E_j \). It also contains \( Er_j \) values. It uses the score, \( Er_j \) and \( Ed_j \) during the credit adjustment step as described in section 4.2.5 to calculate the energies to be distributed to the child programs.

### 4.6 Sample Problem

To demonstrate the ability of the current implementation to eliminate bloat, it is sufficient to introduce the simplest evaluator possible. The fitness function that is called by \( W_i \) as mentioned before accepts a list that contains a number as a parameter. It then uses the table in Figure 4.2 to map the number to the score \( P_i \) has gotten. This is then sent to the controller.

Note the pyramidal arrangement of the fitness function with peak at 10. The seed programs start at 1. Thus, if the input value increases even by 1, the score changes and it has a better chance of survival. Thus this gradient makes it possible to do the stochastic hill climbing that is characteristic of GP.

### 4.7 Seed Program

The seed program or \( W_s \) is given in Figure 4.3. The seed code, when ran, evaluates itself with evaluator 1 which is the sample problem mentioned before. It then goes into a loop calling \texttt{mutate1} and then sleeping for around half a second. The sleep time varies slightly from seed to seed.
<table>
<thead>
<tr>
<th>INPUT</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 and lower</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>90</td>
</tr>
<tr>
<td>12</td>
<td>80</td>
</tr>
<tr>
<td>13</td>
<td>70</td>
</tr>
<tr>
<td>14</td>
<td>60</td>
</tr>
<tr>
<td>15</td>
<td>50</td>
</tr>
<tr>
<td>16</td>
<td>40</td>
</tr>
<tr>
<td>17</td>
<td>30</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>20 and higher</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.2: Input to score mapping for sample problem (eid 1)
Figure 4.3: Seed code
Chapter 5

Issues, Result and its Interpretation

5.1 Issues faced

During the experiments, various issues were faced that prevented the system from giving due preference of survival to the deserving members of the population. They are listed below along with the measures taken to ensure they are eliminated.

5.1.1 Seed-Flatlining

Explanation: All children are seeds and have the same low score because of which they cannot make children. See figure 5.1.

Reason: This occurred because of compulsory seed introduction. It pushed up the population and reduces the energy distribution making the programs have to live on meager earnings and cannot reproduce. Also, all being seed programs, no one had any edge leading to a practically stagnant system.

Solution: Removed the compulsory seed introduction. Now seed is introduced only if the population is below the target. This happens frequently enough to supply a sufficient number of seed programs into the system.

5.1.2 Long life

Explanation: Children live for a long time doing no productive work.

Reason: The starting credits given by the parent/system is actually sufficient to survive for a long time if most of the time is spent sleeping. This leads a useless entity taking up memory space.

Solution: Introduced survival tax. It a debit charged during every credit adjustment in which the same quantity of credit is reduced from every program in the system. However, this quantity of credit may vary from one credit adjustment to another.

5.1.3 Zero credit children

Explanation: Children of 0 credit being created. See figure 5.2.

Reason: A mutation led to putting the credit of created children to 0. This led to indiscriminate spawning of children of energy 0 which were though of high scores, never
(1151269388509 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 82.7183) (1 133.095)
(2 123.251) (3 44.5892) (4 133.493) (5 93.1348) (6 98.6905)))
(1151269389073 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 101.148) (1 151.597)
(2 141.781) (3 63.3719) (4 152.069) (5 111.643) (6 117.362)))
(1151269389623 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 101.035) (1 151.54)
(2 141.748) (3 63.5414) (4 152.072) (5 111.593) (6 117.442)))
(1151269390218 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 100.91) (1 151.478)
(2 141.712) (3 63.7279) (4 152.076) (5 111.537) (6 117.53)))
(1151269390808 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 100.773) (1 151.41)
(2 141.672) (3 63.9329) (4 152.081) (5 111.476) (6 117.627)))
(1151269391439 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 100.622) (1 141.335)
(2 141.628) (3 64.1585) (4 152.086) (5 111.409) (6 117.733)))
(1151269391989 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 101.881) (1 142.679)
(2 143.008) (3 65.8396) (4 153.52) (5 112.763) (6 119.281)))
(1151269392544 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 101.699) (1 142.589)
(2 142.955) (3 66.1126) (4 153.525) (5 112.681) (6 119.41)))
(1151269393157 LOG_ANALYSER_INPUT CREDIT_SAVINGS ((0 101.498) (1 142.489)
(2 142.896) (3 66.4128) (4 153.532) (5 112.592) (6 119.552)))

Figure 5.1: Successive credit adjustment steps with no child being created.

(1151227086998 LOG_ANALYSER_INPUT CREATE_CHILD 22 21 31501 0)
(1151227088610 LOG_ANALYSER_INPUT CREATE_CHILD 23 21 31540 0)
(1151227090300 LOG_ANALYSER_INPUT CREATE_CHILD 24 21 31578 0)
(1151227092009 LOG_ANALYSER_INPUT CREATE_CHILD 25 21 31615 0)
(1151227093540 LOG_ANALYSER_INPUT CREATE_CHILD 26 21 31646 0)
(1151227095220 LOG_ANALYSER_INPUT CREATE_CHILD 28 21 31685 0)
(1151227096742 LOG_ANALYSER_INPUT CREATE_CHILD 29 21 31722 0)
(1151227098384 LOG_ANALYSER_INPUT CREATE_CHILD 30 21 31759 0)

Figure 5.2: Portions of Log showing zero credit children.
got a chance to get the credits to live. However, the parent survived very well since it is spending absolutely no credits in the creation process.

**Solution**: Introduced mutation tax. Every parent is charged the time taken in spawning a child or a fixed amount for every child spawned.

### 5.1.4 Premature parent death

**Explanation**: Parent die immediately after creating child. See figure 5.3.

**Reason**: This leads to loss of good optimizations. This occurs because making a child drains all of the parents credits leaving it vulnerable in the next credit update.

**Solution**: Solution is to set base credit beyond mutation credits before spawning children.

### 5.1.5 Non-seed flatlining

**Explanation**: The rate of change of energy exists but is minor leading to long durations in which no child creation a.k.a search is done. See figure 5.4 to get an idea.

**Reason**: This occurs due to all programs being caught with no energy to make children.

**Solution**: Many solutions were imagined. One was to increase survival-tax till imbalance increases and new children are created. But this did not work always. Another is to knock off the child with the lowest score. However this had obvious disadvantages. Currently, this is being solved using an anti-flatline-bonus (a multiplier) which increases credits distributed between flatlining children. This value itself also increases exponentially (but with a low rate of increase) every time the flatlining issue remains unresolved.
Figure 5.4: Successive credit adjustment steps with no child being created.

Figure 5.5: Section of the log showing misers in the populations

5.1.6 Miser

Explanation: A program that makes a good score but does not spend it and accumulates the credit leading to long term survival but no usefulness. See 5.5 to observe the presence of misers.

Reason: This is a mutation and an important phenomenon. Basically misers occur because the mutation portion of the code has been overwritten. A miser affects the entire world since credit given to the miser is never circulated back.

Solution: The miser problem cannot be solved properly in the current implementation of EPSys due to its reduction in scope. This is because no issue is resolved unless a solution is encouraged and a solution to the miser issue can not be encouraged in the current system. Of course, a manual hack can be introduced to contain the issue but it goes against the hack free approach of EPSys.

The hack is that an upper limit was placed on the energy that can be maintained by any child. Beyond this, it will be terminated immediately. This appears to solve the
problem of miser effectively since it reduces the medium to long term survivability of miser to zero. Also, since misers do not reproduce, the problem does not spread. But note that we are solving the symptom not the problem. The problem being that there is no encouragement to save time in the current implementation of EPSys.

5.2 Results

After the above issues were solved, EPSys ran without any such issues. However, continuous running seemed to lead to hanging of the controller due to buffer overflow errors in the TCP/IP library that was used in the implementation. This we have as of now attributed to bugs in the scheme library. However, the system ran for a respectable amount of time making it possible to note gradual and significant increase in the maximum score registered with evaluator 1. The result for a particularly good run can be summarized as follows. All runs exhibited similar tendencies but most did not run for this long and improve this much.

- Initial score registration of 10 (corresponding to seed input 1).
- Score 20 achieved at 18 seconds.
- Score 30 achieved at 132 seconds.
- Score 40 achieved at 153 seconds.
- Score 50 achieved at 244 seconds.
- Score 60 achieved at 225 seconds.
- Score 70 achieved at 251 seconds.
- Score 80 achieved at 394 seconds.
- Gradual score increase from 10 to 80 over time.
- Code of the first child to reach score of 80 is given in figure 5.6. It is clear that the only difference from the seed code is in the expression that makes up the input value.
- CPU usage around 70%. This depends on the population.
- Memory usage less than 170MB resident.
Figure 5.6: Code of child with score 80

```
(applyg get-eval 1)
(list (applyg sleep 570))
```

```
(lambda (gensym-x)
  (applyg sleep 100))
```

```
(lambda (gensym-y)
  (applyg sleep 570))
```

```
(lambda (gensym-z)
  (applyg get-eval 1)
  (list (add (sub (add (if #t 1 0) 1) (add 1 1)))
        (lambda (gensym-f)
          (applyg mutate 100))))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```

```
(lambda (gensym-x)
  (applyg mutate 100))
```

```
(lambda (gensym-y)
  (applyg mutate 100))
```

```
(lambda (gensym-f)
  (applyg mutate 100))
```

```
(lambda (gensym-n)
  (applyg mutate 100))
```
5.3 Interpretation of Result

As mentioned in the beginning of Chapter 4, the first aim of this implementation and experiment is to prove is that like GP, given a mutator that can mutate to the solution and given a fitness function that can assign the correct fitness value to it, the current implementation of EPSys can optimize a population of computer programs according to the fitness landscape determined by the fitness function. And as we can see from the result of the experiment, EPSys has gradually increased the maximum score attained by the members of the population. Thus it has been experimentally proven that EPSys is equal to GP in its ability to evolve.

But we can also prove that quite unlike GP, EPSys does not suffer from the problem of bloat that has plagued all variations of GP since its inception. As mentioned in chapter 2, many ways to control and contain bloat has been suggested. However, no one has suggested a way to eliminate bloat.

Bloat exists in GP due to two reasons acting together. One is the occurrence of Introns which are portions of code that take up CPU while performing very little if any useful work. Ocurrence of introns which are sometimes extremely time consuming is inherent in the random nature of code mutation and crossover. The second is the ability of a program that has been allocated the CPU for execution to hold on to the CPU indefinitely.

In EPSys, we remove the second requirement for bloat, thus eliminating it. The second reason is eliminated due to the microsecond level of control of the CPU time spend by a child program using the new credit system and due to the enforcement of the CPU time restrictions using OS level commands which cannot be overridden by the program. Note that the child program that made score 80 is not suffering from bloat.

Consider a program with the bloat problem. It will continuously use CPU. Since a bloat free EPSys uses CPU at 70%, this program will get all the CPU it asks for. Since starting energies are low, it will run out of credits very fast and will be removed from the population by the OS. Note that this way, EPSys discourages bloat and encourages computational efficiency which makes it possible to claim that bloat is eliminated.

In a comparison with other bloat control strategies for GP, it must be noted that all the one’s that were surveyed operated by restricting the size of the code. Size of the code is used as an approximation to bloat. The usefulness of this approximation is clearly limited as small code can have an infinite loop. Unlike them, EPSys operates using CPU time which is the real metric of concern. These strategies try to limit bloat rather than discouraging it, the latter being far more effective.

It is also noticed that this implementation of EPSys is significantly slower that GP due to massive overhead.
Chapter 6

Conclusion and Future Work

In the introduction, we explained in short, how the conventional evolutionary techniques were plagued by problems affecting their scalability. We decided to build a platform for evolution which avoided the many disadvantages of these techniques. Thus, we began developing EPSys, an evolutionary platform for evolving programs with many potential advantages in mind.

We then go into a detailed design of EPSys and the working of the design. The design specifically leaves certain areas vague to give freedom to the implementer.

This is followed by a description of an implementation of EPSys which has been tuned down due to development time restrictions. This description fills in the implementation specific vague areas in the design thus completing the overall description of the system. The setup used for running experiments is also described.

Then, we present the issues faced in running the system and measures taken to tackle them. This is followed by the result of running the system. As is explained in the previous chapter, the inferences that can be drawn from the result is that EPSys is functionally, equivalent to GP although slower. It can also be concluded that EPSys is free from bloat, being able to discourage it rather than having to limit it.

We now present some improvements to the current implementation and design which could be expected to increase speed and scale far better than existing variations of GP.

6.1 Future Work

It must be mentioned here that a full implementation of EPSys was designed to speed up based on emergent optimizations. These are techniques for speeding up the process of searching for better child programs. They were not present in the seed code but are present in the current population having randomly come into existence over the duration of the run and having survived due to the advantage it offers. These optimizations could be a matter of new code or just adjustment of parameters.

Let us now look into some interesting features that could help speed up EPSys. There are many more possible but these are enough for now.
6.1.1 Mutation Control

It is necessary that the implementation encourage emergent optimizations. The current implementation does not do this and so we suggest an improvement to the implementation. Mutation control is the most basic emergent optimization possible in EPSys. Thus, it is also the most critical. Mutation control is the ability of the child program to control the code of children that is created from it. The design mentions that ideally, the entire code should be decided by the parent. In the previous implementation there was absolutely no control by the parent. In mutation control, we introduce possibilities in between these two states. The mutation function will be implemented in such a way that the child process is able to control the mutation function’s working.

For example, one way this could be done is by having mutatable meta information in the child code. This meta information could affect mutation in specific ways. Also a child would inherit most of the meta information from the parent. Since some values in the meta information could be more beneficial for survival that others, over time the meta information would be optimized to high survivability. As long as survivability depends on speed of evolution\(^1\), this feature would ensure that evolution speed increases.

6.1.2 Community Evolution

This feature ensures that over medium term, survivability depends on evolution speed with no exceptions. For this, we change the design a little. We split the population into a fixed number of communities. These communities are designed to be mutually exclusive i.e., the proceedings in one community does not affect the credits or proceeding of any other community.

Then, the speed of improvement of the quality of population of each community is measured using an appropriate metric. We analyze this quality over time to isolate communities that are not increasing in quality as fast as others using another appropriate metric. Next, we remove the under-performing communities from the system and create duplicates of the better performing communities to make up for the lost communities.

This leads to a situation very similar to an athletic race. Due to mutual exclusion, the only way the athletes can win is by running faster. Same applies for the communities. Any action by a member that is detrimental to evolution speed could lead to elimination of the whole community.

Note that this feature ensures the requirement of the previous feature making this a very potent combination. This also catches misers very easily.

6.1.3 Change Optimizations

Change optimizations is a difficult but very important optimization. It can be thought of as an advanced version of mutation control. It involves the emergence of an important ability among the members of the population - the ability to potentially change key aspects during mutation and do it in an efficient way without having to start from square one which is the seed or something close.

\(^1\)It does in most cases, miser issue being a notable exception.
Let us consider two scenarios where change optimizations are useful so we can understand it better.

In the first, imagine an ordinary run with EPSys. An optimization/technique $o_1$ emerges which provides a significant improvement in score. Due to natural selection, many programs adopt it and soon almost every program is having this optimization. Now assume there exists $o_2$, another optimization that can act as replacement for $o_1$ and is must faster. The strategies that give the population the ability to change from $o_1$ to $o_2$ are called change optimizations. This change is made difficult normally by the way optimization is stored in code and possibly by the community situation too. In other words, while it is quite possible for a population to go from no optimization to $o_1$ or $o_2$, it is difficult for a population to go from one optimization to another.

In the second scenario, we consider a situation where the evaluators keep on changing over time. In this case, even though a child managed to get a good score and survive well at one time, it is no guarantee that it will be able to live successfully for ever. When the evaluator changes, its score will go down. Without change optimizations, what we would see is an almost constant state of change in the population. The best soon dies off leading to a different best. In particular, the new best is most likely to have descended from a seed just like the previous best was. This scenario presents very good encouragement for change optimization for obvious reasons. The child which is able to adapt quickly will have it or its children maintaining a high score. Here, a program with best score will be child of another program with best score. Thus evaluator changing scenarios could prove to be very useful for encouraging change optimizations.

6.1.4 Other performance improvements

The current design for EPSys is non-distributed. But this puts a low limit on the size/difficulty of problems that can be solved using EPSys. Thus converting EPSys to a distributed system could prove to be a very useful step where large scale evolution is to be done. It is possible to convert the design to distributed system without much difficulty. The communication between children and the controller is done over TCP/IP which means that child and controller need not be in the same computer. However, kill signals that need to be sent to the child cannot be sent over TCP/IP and can only be done by a process running in the same machine. So, every machine in the distributed system will need to have a proxy controller which will give information regarding the children to the main controller and accept commands such as kill or spawn from it and execute it in the machine. See §6.1 for a possible design for implementation.
Figure 6.1: Future design diagram for a distributed system.
Bibliography


