

ATHABASCA UNIVERSITY

APPLICATION OF EVOLUTIONARY ALGORITHM STRATEGIES TO ENTITY  
RELATIONSHIP DIAGRAMS

BY

GLENN HEINZE

A thesis project submitted in partial fulfillment

Of the requirements for the degree of

MASTER OF SCIENCE in INFORMATION SYSTEMS


Athabasca, Alberta

May, 2004

© Glenn Heinze, 2004

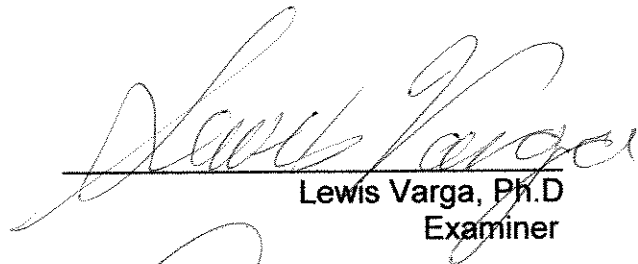
# ATHABASCA UNIVERSITY

The undersigned certify that they have read and recommend for acceptance the thesis project "APPLICATION OF EVOLUTIONARY ALGORITHM STRATEGIES TO ENTITY RELATIONSHIP DIAGRAMS" submitted by "GLENN HEINZE" in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in INFORMATION SYSTEMS.



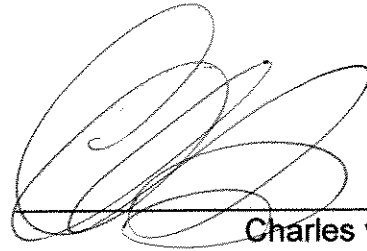
---

Xiaokun Zhang, Ph. D  
Supervisor



---

Lewis Varga, Ph.D  
Examiner



---

Charles van Duren  
Chair

Date: October 2004

## ABSTRACT

The purpose of this study is to examine programming techniques incorporating evolutionary strategies. It has been proposed that evolutionary strategies may provide alternative programming paradigms for finding solutions to problems of polynomial or NP-complete complexity. Disciplines such as Emergent Behaviour, Cognitive Science, Artificial Life, Complexity, and Bio-Informatics are working out the mechanics of how these systems operate. An attempt is made to demonstrate some of these evolutionary strategies through the creation of a program incorporating a genetic algorithm. The goal of the program is to generate aesthetically pleasing entity relationship diagrams. The crux of any genetic algorithm is to develop an appropriate fitness function. This task proves to be surprisingly difficult. Ironically, the task of finding the appropriate balance in the weighting of different fitness factors is in itself a problem that may be best addressed by an evolutionary algorithm. As a practical tool for the generation of aesthetically pleasing entity relationship diagrams, the ERD Viewer application was not overly successful. However, the implementation did provide a great deal of insight into the difficulties of specifying the rules when attempting to guide evolution. There are many suggestions for further development including a potential method to implement learning. Evolutionary strategies hold promise for exploring problem spaces where the number of potential solutions grows exponentially. The proof that these de-centralized, non-global, emergent processes do actually work surrounds us in the natural world. The prime examples are the two stochastic processes: evolution and learning. We are still in the early stages of determining how these techniques can be practically utilized in a programming environment.

# TABLE OF CONTENTS

CHAPTER I - INTRODUCTION .....	1
Statement of Purpose.....	1
Research Problem .....	1
Significance of Problem .....	2
CHAPTER II – REVIEW OF RELATED LITERATURE .....	5
Overview .....	5
Previous Work .....	7
Graph Layouts.....	8
Genetic Algorithms .....	10
CHAPTER III - METHODOLOGY .....	13
Overview .....	13
Principles for an Aesthetically Pleasing ERD .....	14
Principles of Evolutionary Strategies.....	15
Genetic Algorithm Implementation .....	18
CHAPTER IV - RESULTS.....	20
Removing Relations from the Layout Chromosome .....	20
Orthogonality.....	22
The Fitness Function .....	24
Performance.....	26
CHAPTER V – CONCLUSIONS AND RECOMMENDATIONS .....	28
REFERENCES .....	31

APPENDIX A – ERD VIEWER USER’S GUIDE .....	33
Launching the Application .....	33
Manipulating the Schema .....	33
Configuring the GA Engine.....	34
Adjusting Fitness Values .....	34
Generating Layouts .....	36
APPENDIX B – ERD VIEWER DESIGN OVERVIEW.....	37
GA Engine .....	37
ERD ADT .....	38
GUI Components .....	39
Graph ADT .....	40

## LIST OF TABLES

Table 1. The ERD ADT .....	38
Table 2. The ERD gui components.....	39

## LIST OF FIGURES

Figure 1. The relationship of a Layout to a Schema.....	19
Figure 2. Annealing for preferred edge layouts.....	21
Figure 3. Determining an ideal edge layout between entities with an intersecting horizontal plane.....	21
Figure 4. Fitness weighting relative to the slope of an edge .....	23
Figure 5. The ERD Viewer in operation .....	25
Figure 6. Fitness factor configuration options.....	35
Figure 7. The ERD ADT class diagram. ....	39
Figure 8. The ERD gui class diagram.....	40
Figure 9. The graph ADT class diagram.....	41

# CHAPTER I

## INTRODUCTION

### Statement of Purpose

The purpose of this study is to examine programming techniques incorporating evolutionary strategies. It has been proposed that evolutionary strategies may provide alternative programming paradigms for finding solutions to problems of polynomial or NP-complete complexity. As a demonstration of how some of these techniques may be implemented, an application incorporating a genetic algorithm is developed with the goal of finding an aesthetically pleasing layout for an entity relationship diagram.

### Research Problem

Entity Relationship Diagrams (ERDs) are graphical representations of data items and their inter-relationships. ERDs are typically used by database designers to document and communicate the relationships between tables within a database application. Developers use ERDs as a tool in order to quickly determine which tables contain the information they are looking for. They can follow the linkage from known item to desired item through a series of relationships between tables in order to determine a retrieval path. As a tool for understanding how a datastore is pieced together, an ERD is indispensable.

Most database vendors provide tools to view and manipulate the graphical representations of schema objects. For example, Oracle Corporation provides a tool called “Designer” which incorporates this ability. However, these layout tools typically do not attempt to apply any intelligence to the layout of the schema objects beyond the avoidance of entity overlap. It is up to the user of the tool to re-position the locations of



the graphical components into an arrangement that makes sense. That is, into an aesthetically pleasing layout.

There are different factors which may contribute to a layout being considered aesthetically pleasing. One factor is the degree to which a layout conforms to conventions. Although the notations used by each vendor may vary, conventions have developed for the expected layout. For example, it is generally accepted that, if possible, the relationships between entities are drawn as orthogonal lines with a minimal number of intersections. Another consideration which could contribute to an aesthetically pleasing layout would be the grouping of related objects closer together, where the grouping may be defined by user preferences. Both these examples can be surprisingly difficult to accomplish, especially as the number of entities and relations in the schema start to grow. This probably accounts for why the auto-layout feature on ERD tools do not even attempt anything beyond ensuring entities do not overlap each other.

One of the motivations for examining evolutionary strategies in the development of an ERD layout program is to learn whether these types of problems can be addressed in an elegant manner. Is it possible to create a tool to sample the problem space and provide suggestions on layouts which may be considered aesthetically pleasing?

### Significance of Problem

Attempting to develop a method for deriving ERD layouts serves an obvious practical and utilitarian purpose. However, the use of evolutionary strategies incorporating techniques such as evolution have a much broader importance. Evolution and learning have been referred to as the great stochastic processes (Bateson, 1979, p. 141). These two naturally occurring systems merit examination from a computational

perspective because they have demonstrated the ability to address computationally difficult problems. These stochastic systems are often tasked to solve problems of polynomial or NP-complete complexity, and yet they propose results in a reasonable amount of time. The success of these systems is not necessarily in obtaining the best solution to a problem, but in obtaining a sufficient result in a restricted amount of time. Any known brute force algorithms attempting to solve these types of problems would take longer to complete than is of practical use. In the natural world, this may equate to making a “fight or flight” decision before being eaten. In a computational sense, this can be translated into performing a limited search of a problem space in order to locate a local optima (if not quite the ideal global optima) in a limited amount of time.

Disciplines such as Emergent Behaviour, Cognitive Science, Artificial Life, Complexity, and Bio-Informatics are discovering the mechanics of how these systems work. A common theme from these studies suggests complex behaviour can be obtained via simple rules without the need for some sort of centralized orchestration. This suggests that the techniques developed in these fields may become significant in the development of non-deterministic algorithms for solving difficult problems. These techniques include the use of populations, evolution, selection, fuzzy logic, feedback, adaptation, and learning.

The study of evolutionary algorithms is of interest for many reasons. From a strictly aesthetic point of view, it is amazing that events such as random mutation and natural selection can lead to points of stability without the need of a guide and without any “knowledge” of the problem being solved. We tend to view our world from a hierarchical perspective. We have leaders of countries whom we view as our benefactors

or the cause of our woes. We write computer programs with controllers, dispatchers, and daemons. So when we are confronted with systems that operate in a truly distributed manner, without a hierarchy, it is almost shocking to believe it is possible that they can function at all. That it is so un-intuitive indicates that there is gap in our understanding. And yet we know that many natural systems which appear to be directed, such as ant colonies and molds, truly do act in a non-hierarchical manner (Johnson, 2001, p. 14-17). Learning to understand the principles of how these systems operate may provide a greater insight into how the natural world works. We may also be able to apply the knowledge to our computing paradigms leading to systems of equal components without critical points of failure.

The two great stochastic processes, evolution and learning, are directly related to the fields of genetics and cognitive science. The investigation of evolutionary algorithms can also be considered as an exercise of modelling and simulation within these fields. Discoveries in genetic processes may provide ideas for developing computational processes. Similarly, the discovery of strategies which “work” in the computing environment may provide clues to the mechanics of cognitive processes. Integrated studies start becoming studies of pattern abstracted from the hardware of implementation whether it be genes, neurons, or resistors.

Besides aesthetic or biological purposes, evolutionary algorithms have shown themselves as practical alternative methodologies for searching large problem spaces where only polynomial search algorithms are known, or even worse, where no known algorithm exists at all. Further research into evolutionary strategies in algorithms may suggest approaches on how to deal with these difficult problems.

## CHAPTER II

### REVIEW OF RELATED LITERATURE

#### Overview

Edward O. Wilson is well known for his study of insect societies. He has laid out the groundwork showing how sophisticated processes, as witnessed in ant or termite colonies, can emerge by having individual members of the population following simple rules. Building on these rules, ant simulation programs have become a favourite model in the artificial life community. In *The Mind's I*, Daniel Dennett and Douglas Hofstadter muse how such emergent behaviour may have applicability to the mind and discuss the concepts of reductionism and holism. The importance of pattern and the study of linkages is stressed by Gregory Bateson in *Mind and Nature, A Necessary Unity*. Authors such as Ray Kurzweil argue that mind itself is simply pattern. Christopher Langton, whose important contributions helped bring together an “Artificial Life” community, similarly expresses “life as a property of the *organization* of matter, rather than a property of the matter which is so organized.” (Langton, 1989/1992, p. 2). Concepts in this field include “themes such as *bottom-up* rather than *top-down* modeling, *local* rather than *global* control, *simple* rather than *complex* specifications, *emergent* rather than *prespecified* behavior, *population* rather than *individual* simulation, and so forth.” (Langton, p. xvi).

Bateson also discusses the importance of the random in order to provide the potential for something new. For example, in the basic version of Conway’s “Game of Life,” the random element does not exist. The “game” is a two dimensional version of a cellular automata where the next state of each cell is determined by evaluating the state of

neighbouring cells. Depending on the initial state of the board and the rules for how the next state of a cell is calculated, patterns may emerge which will eventually achieve a steady state after a sufficient number of generations. In order to escape the steady state, something must enter the picture to disturb the steady state. Typically this is done by adding an “obstacle” in the environment –the apparently random element. Many have made the observation that the complexity of systems are in reality reflections of the complexity of the environment. Richard Belew states that “ecologists have long recognized that the complexity of an organism’s behavior is related to the environment it must ‘solve’.” (Belew, 1991, p. 12)

Works by Richard Dawkins, such as *The Selfish Gene* and *The Blind WatchMaker*, have made significant contributions in introducing and promoting the concept of evolution to a broad audience. He manages to instil another level of humility in mankind: not only does the sun not revolve around the earth, but the world does not revolve about man either –we are simply gene machines, the products of a great stochastic process.

The combination of these diverse fields of thought comes together in evolutionary algorithms: cellular automata, evolutionary concepts, stochastic process, etc. James Kennedy and Russell C. Eberhart have put together these pieces in a beautiful book: *Swarm Intelligence*. The work emphasizes the importance of populations. Generate a population of potential solutions to a problem and let the population evolve in a “problematic” environment. After numerous generations, it will be those members who have adapted to the environment that will survive. It is not an overly technical work, and yet it provides sufficient examples to make it practical.

John Holland is often considered the father of the genetic programming field due to his great influence. Many of his students are well known for their classic works in the area. For example, David E. Goldberg's *Genetic Algorithms in Search of Optimization & Machine Learning* provides a step by step process for creating generic genetic algorithms. This work derived from his experiences in trying to solve a traveling salesman class of problem in pipeline networks. Another of Holland's students, John R. Koza, has authored a technical tome entitled *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. While genetic algorithms evolve populations where the individual member's fitness represents a solution to a problem, and particle swarms concentrate on interactions of individuals, genetic programs concentrate in evolving the actual *programs*.

#### Previous Work

Using genetic algorithms to generate aesthetically pleasing graph layouts is not an unusual exercise. This is probably true for a number of reasons. For one, graph problems are often difficult problems to solve, involving questions on existence, construction, enumeration, and optimization (shortest path, scheduling, minimum connector, traveling salesman) (Wilson & Watkins, 1990, p. 293-299). There is an ongoing search to find more practical methods to solve these types of problems. It is also true that these types of problems are very common. Nodes represent things like people, cities, or hardware while the edges represented paths and restrictions between them. The following sections review some of the previous work in graph layouts as well as the more general usage of genetic algorithms.

Graph Layouts. Toshiyuki Masui's paper *Evolutionary Learning of Graph Layout Constraints from Examples* provides a good starting point for defining what constraints might define 'good' graphs. Masui starts by listing constraints which make the layout look nice (Masui, 1994, p. 104):

1. There should be enough space between nodes.
2. The head of an arc should be below the tail.
3. Arc crossings should be avoided.
4. There should be as much symmetry as possible.
5. The angle between two arcs should not be too small.
6. Nodes should be placed uniformly in the region.

From these rules, Masui developed the following formula to calculate the fitness value for his generated graphs:

$$\begin{aligned} & 3000 * (\text{the number of arrows going upward}) + \\ & 400 * (\text{the number of arcs shorter than a constant } C1) + \\ & 300 * (\text{the number of arc crossings}) + \\ & 400 * (\text{the number of angles between arcs which are smaller than a constant } C2) \end{aligned}$$

In this case, the smaller the fitness value calculated for a graph, the greater the fitness of the graph. This means the fitness function actually acts as a penalty function. At a glance it seems apparent that most constraints are relatively evenly weighted -except for upward pointing arcs which seem to be more severely penalized. The formula for this fitness function seems simple enough. "However, getting an appropriate evaluation function is not an easy task," Masui warns. "This formula contains many magic numbers. A number of trial and error loops have been performed before getting these values.

Changing the values and functions slightly will produce totally different results that are quite unpredictable.” (Masui, 1994, p.104-5).

M.H.W Hobbs and P.J. Rogers paper *Representing Space: A Hybrid Genetic Algorithm for Aesthetic Graph Layout* provides an example very similar in nature to Masui’s work. While using a different method to perform crossovers (involving subgraphs rather than locations), Hobbs and Rogers also started their work by defining criteria for aesthetically pleasing graphs. Their list considered (Hobbs & Rogers, 1998, p. 416):

1. Total edge length. This is simply the sum of the lengths of each edge.
2. Graph area. This is a calculation of the area of the bounding rectangle of the nodes in the graph.
3. Node overlap. A weighted penalty is given to nodes that are too close or overlap each other.
4. Angular resolution. This is a calculation of the angle between edges connected to the same node and penalizes connections that form acute angles.
5. Edge crossing. Applies a penalty function based on the number of edge crossings in the graph.

Hobbs and Rogers optimistically conclude that their genetic algorithm provides “an excellent pre-processor to these methods where larger graph sizes make an exhaustive, deterministic approach slow or prone to trapping in local optima” (p. 417).

Masui and Hobbs & Rogers concerned themselves with generating aesthetically pleasing graphs in general. They were not considering ERD graphs in particular. In the paper *Drawing Database Schemas*, Di Battista et al. focus exclusively on ERDs (although



with standard deterministic algorithms). In their introduction, they comment that “to our knowledge, no drawing technique is available to automatically produce high quality diagrams representing database schemas” (Di Battista, Didimo, Maurizio, & Pizzonia, 2002, p. 1065). There is no standard for creating ERDs; however, there are certainly generally accepted conventions. For example, ERDs typically try to use orthogonal lines. Di Battista et al. propose their considerations for generating aesthetically pleasing ERDs which tend to follow these conventions by minimizing the following (p. 1072):

1. number of crossings
2. number of bends
3. area of the bounding box
4. total edge length

It is desirable to minimize all these criteria. The comment that “it is impossible in general to minimize at the same time all the above measures. Only tradeoffs among them can be pursued” echoes Masui’s comments on the difficulty of obtaining an appropriate fitness function (Di Battista et al., 2002, p. 1073). While the work of Di Battista et al. approaches the problem from a deterministic perspective, the criteria they use and the insights they provide are still very useful in the consideration of other implementation techniques such as genetic algorithms.

Genetic Algorithms. Goldberg’s authoritative work *Genetic Algorithms* provides the mechanics for the implementation of a genetic algorithm. The beauty of the genetic algorithm is that the engine is blind to the meaning of the problem. Once an engine is built, it can be used to evolve populations for any kind of problem, provided the problem conforms to a specific interface. A minimal interface includes the following criteria:

1. a population member can represent itself as an encoded string
2. a fitness value can be calculated from the encoded string
3. a new population member can be derived from an encoded string

What the meaning of the encoded string is or how the fitness value is calculated is irrelevant to the engine and becomes part of the definition of the problem space. The genetic algorithm simply selects members of a population based on their fitness, and then randomly breaks and recombines the string encodings of the selected members in order to create a new population. The alphabet of the encodings relate to genes, and the string encodings to chromosomes. Over generations, characteristics that lead to higher fitness values should start dominating the population as members with these characteristics will be the ones able to reproduce for subsequent generations. The members “evolve” to a higher fitness level through natural selection, hopefully producing future generations with members having a high enough fitness value to represent a sufficient solution to a problem.

Kennedy and Eberhart’s *Swarm Intelligence* extends the idea of genetic algorithms to another level. For them, a problem (or the intelligence of a solution) is represented by the population rather than an individual member. They have considered Bateson’s two great stochastic systems, evolution and mind, and have realized that “the great stochastic systems are different from one another: one uses selection, removing less fit members from the population, and the other adapts by changing the states of individuals who persist over time” (Kennedy & Eberhart, 2001, p. 15). A genetic algorithm uses a purely genotype centric approach. It corresponds to only one stochastic system: evolution. But what about the other stochastic model: the mind? Kennedy and

Eberhart argue that the mind does not function in isolation but requires the interaction of a population. There needs to be a force that can change the states of individuals within a generation. Changes do not just occur over generations. This may be equivalent to what we consider culture and learning. During our lives, we encounter bodies of knowledge, either abstract or within other people. These encounters may or may not change our internal states. Oliver Selfridge makes the point succinctly: “Can you think of some chore or duty that a person does that she doesn’t do better the second time? Or can you think of some chore or duty that a computer does that it does do better the second time?” (Hearst & Hirsh, 2000, p. 11)

The present study was designed to investigate evolutionary programming strategies and attempt to apply the lessons learned in a practical example. The hope was to develop an emergent piece of software where the result could be guided by a basic set of rules rather than with a global, deterministic approach. The genetic algorithm provided the infrastructure for the program, while the observations of what constitutes an aesthetically pleasing graph layout provided the guiding rules.

# CHAPTER III

## METHODOLOGY

### Overview

The purpose of this work is to explore how evolutionary strategy techniques can be used as a programming paradigm. In order to obtain a deeper understanding of some of these principles, a practical example has been undertaken. This example attempts to evolve an aesthetically pleasing ERD layout for a database schema using a genetic algorithm.

Scope and time constraints have limited the number of evolutionary techniques that could be explored. However, the results of the ERD Viewer program provide a good indication of how the program could be developed further. Due to the visual nature of this problem, a graphical interface was developed in order to view the results. The graphical objects in the layout could be easily enhanced if desired, but this was not the focus of this work.

The implementation of this program followed the following steps:

1. Derive a list of criteria which would lead to an aesthetically pleasing ERD layout.
2. Derive a list of principles which reflect evolutionary bottom up strategies rather than traditional, hierarchical programming principles.
3. Implement a generic genetic algorithm (GA) engine.
4. Develop a population of members conforming to the genetic algorithm interface where the fitness function takes into consideration the principles from points 1 and 2.
5. Experiment with the fitness function.

## Principles for an Aesthetically Pleasing ERD

By convention, an ERD diagram typically has vertical and horizontal edges to connect nodes. The nodes represent entities (tables) within a schema, and the directed edges represent foreign key relations between tables. In a complex ERD, it is usually not possible to directly connect entities with single orthogonal line segments. In this case, edges still run orthogonally, but will abruptly changing direction 90 degrees (perhaps more than once) in order to complete the run from one end point to the other. For the purposes of this program, it is assumed that relatively simple ERDs are to be plotted. This program does not attempt to break lines. Considering the principles proposed in previous works and by examining typical ERDs generated in the work environment, a number of principles for creating aesthetically pleasing ERD layouts are proposed. The following principles are used in the implementation of the ERD Viewer program:

- An entity has a preferred height based on the number of columns in the table and a preferred width based on the size of the name of longest column.
- A tendency to the vertical or horizontal is preferred for edges.
- Entity overlaps are highly undesirable. A minimum distance between entities should be maintained.
- Edge intersections should be minimized.
- Edge – entity overlaps should be minimized; however it is typically preferable to have such an overlap (entity on top) than an edge intersection.
- The entire area occupied by the ERD should be minimized.

The following list of principles would also add to an aesthetically pleasing layout for an ERD, but have not been implemented in this program:

- An entity can be stretched a limited amount from its preferred size provided it improves the overall fitness.
- Edge end-points should be as close to the middle of the attached entity edge as possible.
- It is preferable to have horizontally and vertically aligned entities.
- Entities related to each other should generally be plotted closer together (maintain grouping).
- In a master-detail relationship, it is preferable to be consistent in having the master table in the same relative position to the detail table (e.g. always on top or always to the left).

If we consider these principles to be forces tugging the layout of the graph in different directions, it becomes apparent that there are many conflicting forces. For example, attempting to minimize the area will increase the likelihood for intersections. The difficulty in developing the fitness function for the ERD Viewer is to find an appropriate set of relative weights for each principle. The ERD Viewer allows the weights to be user configurable in order to experiment with different weightings. The difficulty of determining the weighting is a serious issue. Masui actually discusses “developing the layout evaluation function through genetic programming” (Masui, 1994, p. 105).

#### Principles of Evolutionary Strategies

Some principles to consider when employing evolutionary strategies include:

- using local information rather than global information
- employing feedback
- learning

- annealing results

In general, evolutionary strategies applied to algorithms evolve a set of possible results through generations. Each generation is derived by the manipulation of members from the previous generation. Perhaps the single most important concept when considering evolutionary strategies and how the transformation between generations is to occur is to think locally. Assume no global knowledge. We can see why this is so important when we consider the types of problems we are trying to deal with. The traveling salesman problem grows exponentially with each addition of a city. It is because of this exponential explosion that brute force algorithms become unusable. They are simply unable to traverse the entire problem space to find the optimal solution in a reasonable amount of time. Traversing the entire problem space is an attempt to gather complete (or global) knowledge which is simply not practical. To circumvent this, decisions need to depend solely on the knowledge of the local area (nearest neighbours). Every one of the millions of decisions we make each day is based on an assessment of the limited local knowledge surrounding us, not complete information. It is often true that an optimal solution cannot be determined strictly by local knowledge, but it is hoped that that over the long run, the proper local decisions will lead to a good global solution. This is what we expect natural selection to evolve towards over a number of generations: a sufficiently optimal solution.

Another important strategy is to employ feedback, where the results of one iteration directly impact the results of the next. This is something built directly into genetic algorithms with the propagation of strings in the genotype over the generations.

A strategy which is missing from genetic algorithms, however, is learning. The phenotype should be able to learn either from other members of the population or through a common body of knowledge which is made available in the environment of the population members. Everything should not be strictly dependent on the genotype. Kennedy and Eberhart describe how it may be possible to implement this: when an entity encounters a neighbour, give it a chance to copy a characteristic of the neighbour (Kennedy & Eberhart, 2001, p. 265). If we allow the fittest members of the population the ability to survive into subsequent generations before dying of old age, they will have a greater likelihood of sharing their knowledge with younger members, and hopefully that includes their survival characteristics.

Similarly we could create an ever-present library in the environment which may swap characteristics with those members who encounter it (Kennedy & Eberhart, 2000, p. 254). If the library is visited by numerous members with a particular characteristic, that characteristic might be added to the library (part of the culture). A member encountering the library may be inclined to adapt a characteristic from the library. In this way, a practical implementation of learning is possible, without any knowledge of what is being learned nor of the correctness of what is being learned. Learning becomes a matter of imitating others.

Stochastic processes do not have to dictate all events. Sometimes there is a low cost, deterministic method to improve or “add value” to a solution. This is a process often referred to as annealing. It may be very useful for members of a population to undergo an annealing phase in order to nudge characteristics in a direction known to improve fitness. Annealing is one of the evolutionary strategies the ERD Viewer program attempts to

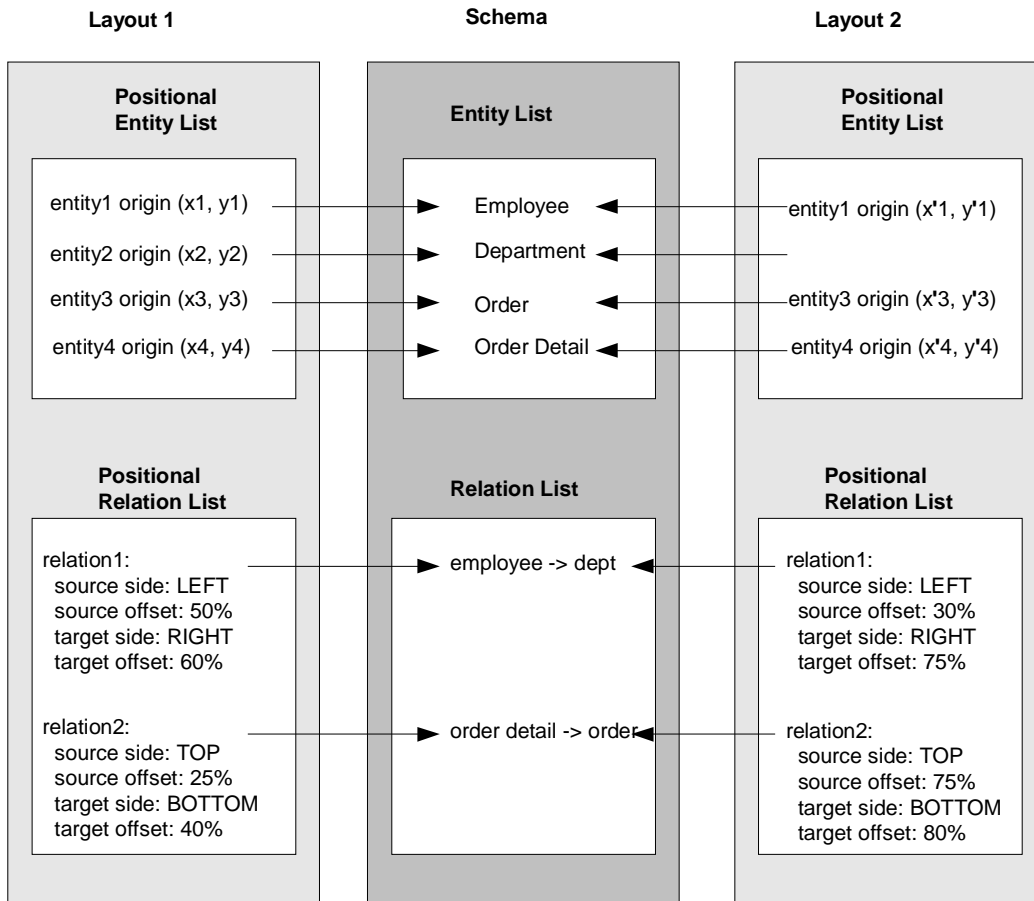


employ: once a layout is determined, the annealing stage attempts to slide the edge into a more ideal position with respect to the entity.

### Genetic Algorithm Implementation

The ERD Viewer's genetic algorithm is based on the manipulation of positional co-ordinates for entities and relations. A schema contains a list of entities and a list of relations between entities. A layout is applied to a schema and it maintains a positional mapping for each entity and a positional mapping for each relation within that schema. A layout has sufficient information to plot entities by maintaining the x and y co-ordinate of the origin for each entity. The length and width of the entity can be determined from the entity's preferred size. To plot a relation, the layout maintains the relative offset of the end point of each relation from the side of the entity to which the end point is attached. For example, a relation may start at 30% down the right side of the source entity and run to 50% down the left side of the target entity. Once a schema is loaded or created, a population of layouts is randomly generated for the schema. Figure 1 illustrates the relationship between layout and schema. It shows two different layouts for the same schema and each layout would result in a different graphical representation of the schema.

**Figure 1. The relationship of a Layout to a Schema**



A binary string representing a layout is formed by concatenating the positional co-ordinates (in binary format) of the layout. The co-ordinates are also used to determine the fitness of the layout while the genetic algorithm engine operates on a population of layout strings. To plot a schema, the entities and relations of the schema are positioned according to the co-ordinates in a layout.

In the early stages of the design concept, there was consideration of using XML elements in place of the traditional binary strings for building the chromosome. This idea was quickly discarded when it became apparent that the layout for an ERD could be easily represented with purely numerical values (positional co-ordinates), a far more convenient mechanism.

## CHAPTER IV

### RESULTS

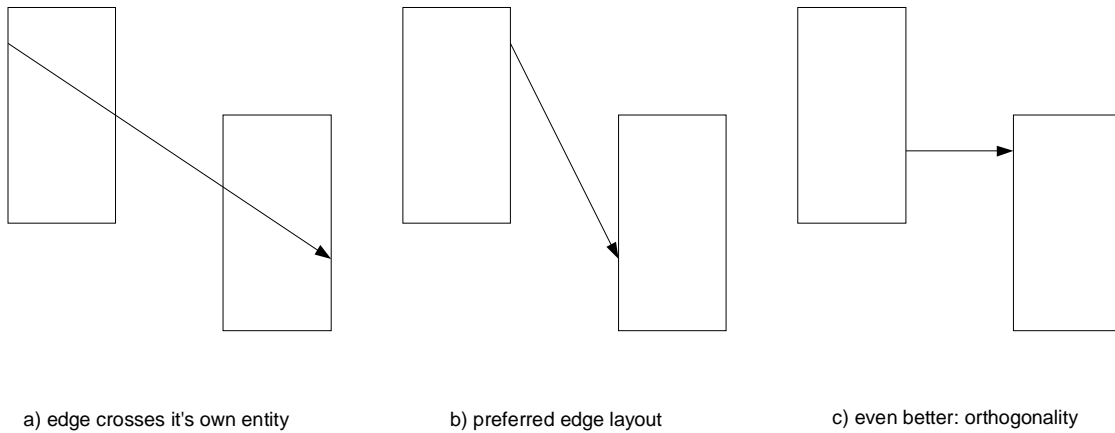
Building a generic genetic algorithm engine, creating a data structure to maintain entity and relation information, and carefully planning the fitness function for the layout did not magically result in producing the desired aesthetically pleasing ERDs. Initial runs did not seem to differ much from random plots. The only truly noticeable result was that the entities were grouped in such a way as to minimize the area the layout occupied. The vast majority of work up to this point was on infrastructure. A surprisingly large amount of effort then involved the manipulation of the fitness function.

#### Removing Relations from the Layout Chromosome

It was observed that a relation would frequently cross the entities at its own end points. This never needs to happen in an ERD, and the frequent occurrence was quite surprising. However, a re-examination of the principles set out for an aesthetically pleasing layout pointed to the reason: a relation intersecting an entity was penalized less severely than an entity – entity intersection or an edge – edge intersection.

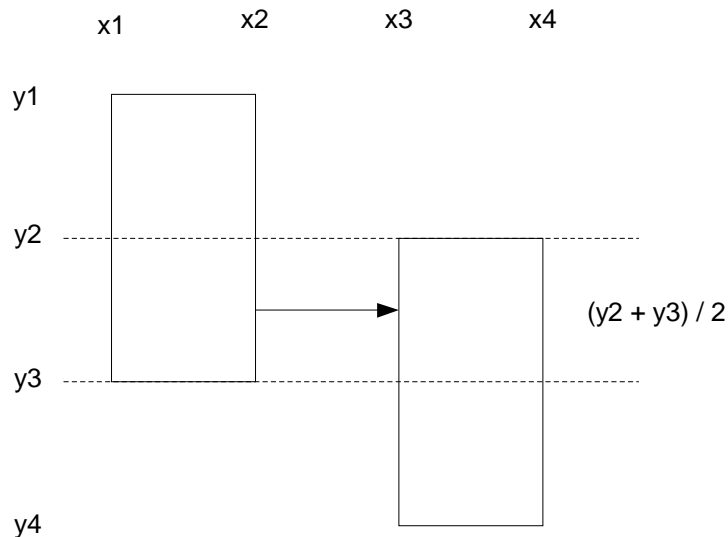
Along with this observation came the realization that the side of a relation to which an end point should be attached could be determined in a constant amount of time: there were only four edges per entity, of which only two could be possible candidates. One way to determine the best side of the entity to which a relation should attach is to cycle through all the entity's sides and choose that which results in the shortest segment length for the relation. Figure 2 shows how edges were often displayed (a), a layout to which edges could be relatively easily re-positioned with some annealing (b), and a suggestion for an even better layout with a horizontal line (c).

**Figure 2. Annealing for preferred edge layouts**



It also became apparent that the relative position on the side of the entity from which the relation should emanate was also easily calculable since the goal was to plot relations as vertical or horizontal lines. If the entities shared a vertical or horizontal plane, the midpoint of the plane could be used, otherwise the edge would emanate from a corner. Figure 3 shows two entities sharing a horizontal plane between  $y_2$  and  $y_3$ .

**Figure 3. Determining an ideal edge layout between entities with an intersecting horizontal plane**



In this case, the co-ordinates of a horizontal line connecting the two entities could easily be calculated as:

$$[x_2, (y_2+y_3)/2] \text{ to } [x_3, (y_2+y_3)/2]$$

This observation led to the removal of relation co-ordinates from the chromosome representing the layout: it was no longer required. Appropriate relation co-ordinates could be quickly determined in an annealing stage. The size of the resultant chromosome representing the layout was drastically reduced. Originally, the size of the chromosome was proportional to the number of entities and the number of relations:

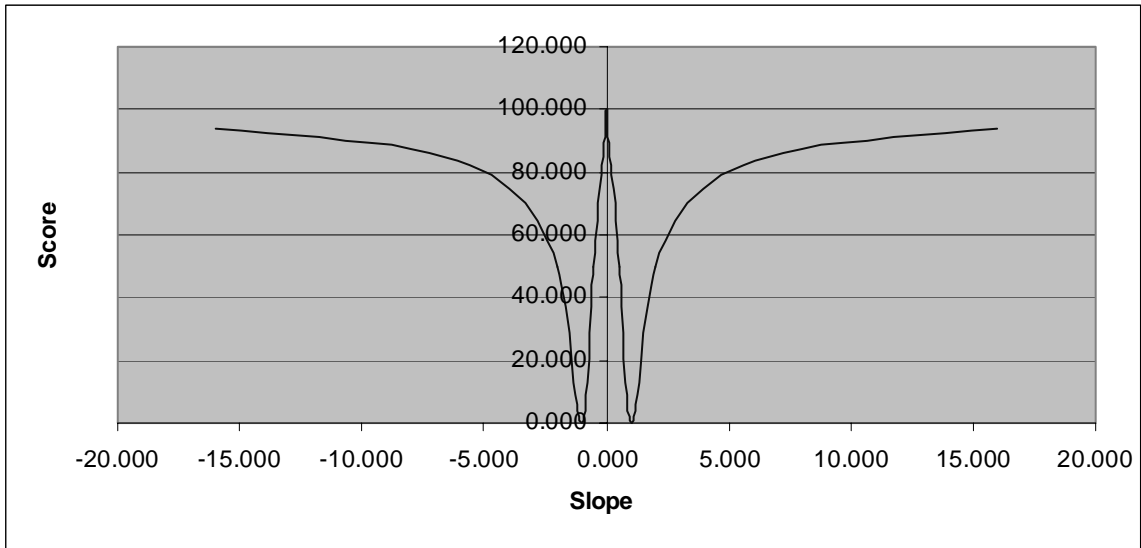
$$\begin{aligned} & \# \text{ of entities} * (10\text{bits for } x \text{ co-ordinate of origin}) + \\ & \# \text{ of entities} * (10 \text{ bits for } y \text{ co-ordinate of origin}) + \\ & \# \text{ of relations} * (2 \text{ bits for source entity side} + 7 \text{ bits for source side offset}) + \\ & \# \text{ of relations} * (2 \text{ bits for target entity side} + 7 \text{ bits for target side offset}) \\ & = \# \text{ of entities} * 20\text{bits} + \# \text{ of relations} * 18\text{bits} \end{aligned}$$

If we assume a typical ERD has at least as many relations as entities, the removal of relations from the representative layout chromosome will effectively halve the chromosome length. This means the problem space searched by the GA is similarly reduced. Implementation of this annealing step did improve the quality of generated ERDs. Edge intersections with the entities at their end points were eliminated.

### Orthogonality

One of the big disappointments with the ERDs generated was the difficulty in obtaining orthogonal lines for the relations. The original fitness function was designed to promote orthogonal edges. The best score was allocated for those edges which had a slope approaching 0 or infinity. A slope of 1 (45°) was the worst case and allocated no bonus. The graph in Figure 4 illustrates how horizontal and vertical lines were promoted in the fitness function.

**Figure 4. Fitness weighting relative to the slope of an edge**



Any edges that were exactly horizontal or vertical were given an additional bonus. However, this did not turn out to be overly successful. Rarely were edges truly orthogonal. When the fitness function was weighted more heavily in favour of orthogonality, it came at the expense of other characteristics such as entity intersections.

Another problem encountered when the weight of the orthogonality factor was increased in the fitness function was the tendency for the entire population of a run to lock onto a particular layout. For example, if a vertical line was found between two entities, the same vertical line would probably be found among many ERDs of the population generated for the particular run. This is an example of locking onto a local optima. The GA engine employs random mutation to try to prevent this, but finding the appropriate mutation rate is another fine balance: too high a mutation rate makes it difficult to evolve any kind of solution and ultimately becomes a random search.

The best results for orthogonality appeared with annealing as described in the previous section. Better results are probably attainable if the relation offsets would be

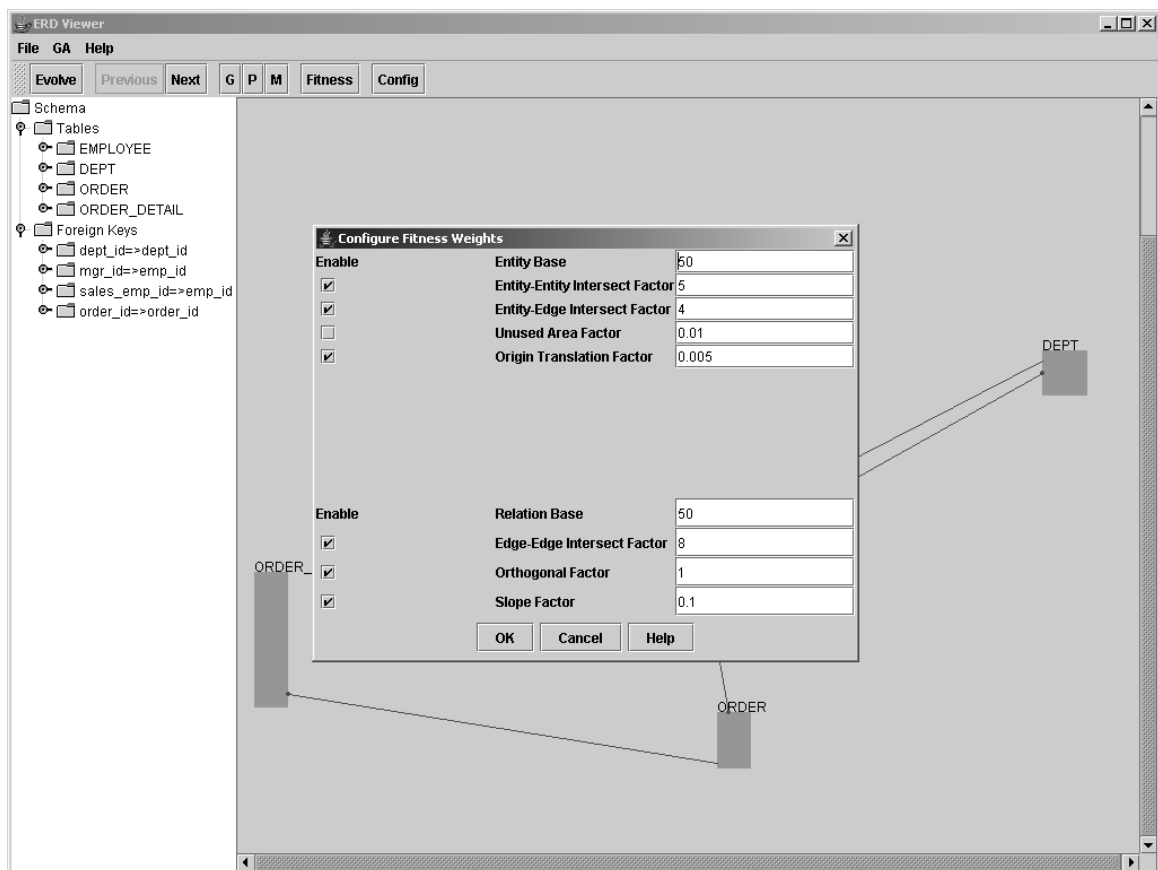
described by an absolute co-ordinate rather than the current percentage offset representation so that the discovery of horizontal or vertical lines are not lost due to rounding errors in the offset representation.

### The Fitness Function

Masui's comments on the difficulty of finding the appropriate fitness function ended up becoming quite prophetic for the experiments of this project. The significance of his statements did not become apparent until a re-reading of his work after the ERD Viewer implementation. In project management we learn of the triple constraint between time, cost, and effort and how the changing of one necessarily impacts the others. The fitness function is a multi-dimensional constraint.

In order to find an appropriate balance between the constraints, the ERD Viewer application allows the weightings of each fitness factor to be adjusted or disabled.

**Figure 5. The ERD Viewer in operation**



Manually tweaking configuration values, however, becomes a complex task in itself. The proper balance of weights is perhaps a task for yet another program. Once a proper balance of factors is derived, the hope would be that the same factors would remain constant for the application and be applicable regardless of the scale of the schema.

Originally the application had absolute values for bonuses and penalties, but it quickly became apparent that this did not scale properly when the number of entities and relations changed. For example, a single edge intersection among two edges is a far more severe issue than a single edge intersection in a diagram with 100 edges. For this reason, the computation of fitness was modified to be computed as a factor of the number of



entities or the number of relations. The base values allow proportioning the weights of entity factors and relation factors. These changes helped avoid negative fitness values when graph sizes increased. Negative fitness values are not handled very well by the GA. If a negative fitness value occurs, the GA engine simply increases the fitness of all members by the magnitude of the minimum value. This could improperly skew the relative fitness values of population members.

### Performance

Performance is proportional to the number of entities, the number of relations, the size of the population, and the number of generations to evolve. While performance was never a driving factor for this project, it obviously has practical repercussions. The original implementation of the ERD Viewer used a directed graph abstract data type (ADT) where each node housed an entity and each directed edge housed a relation. This was considered overly complex for the given application and discarded for a simple list of entities and relations. However, it now appears that it may make sense to return back to the graph ADT in order to avoid issues such as scanning the entity list in order to find the end point of an edge. It would also promote the evolutionary strategy concept of knowing only local information (edge end points directly accessible) rather than global information (full list scans).

The ERD Viewer was exercised with configurations of between 20 and 1000 generations. No appreciable improvement in the layouts of the ERDs seemed to occur when using more than 100 generations. Similarly, no substantial difference was observed between using a population of 50 or 500 members. It seems the fitness function weights exerted a far more significant influence on the outcome of the layouts than the population

size or the number of generations of evolution. The conclusion from this observation is that it is more productive to experiment with smaller population and generation sizes and tweak the fitness function. Only when a successful formula for the fitness function is found might it be useful to re-examine the settings for the population size and the number of generations to evolve for further tuning.

## CHAPTER V

### CONCLUSIONS AND RECOMMENDATIONS

As a practical tool for the generation of aesthetically pleasing entity relationship diagrams, this application was not overly successful. The tool can display potential layouts with minimal intersections. These layouts could then be used as a basis for manually generated diagrams, but they are not complete by themselves. The greatest failure in the tool was in failing to generate orthogonal placements of edges.

Subsequent versions of this program can implement features that may help improve the output. Some of these changes might include:

- Changing the relation offset from a percentage offset of an entity side to an absolute offset. Percentages may not be providing sufficient accuracy in locations needed to provide the full bonus for vertical or horizontal lines.
- Modifying the GA engine to allow for the survival of the fittest members through multiple generations before aging and dying. In this way “fit” members have a better opportunity to share their attributes.
- Implementing learning by allowing population members to interact with neighbours to share attributes.
- Implementing learning by introducing a library in the environment where characteristics can be stored and retrieved.
- Changing the implementation data structure to a graph ADT to more easily determine “neighbours.”

- Implementing a meta-GA to determine the ideal weightings of the fitness function.
- Implementing functionality to promote grouping so that group members that are located together score a bonus.

Implementing the ERD Viewer application provided a great deal of insight into the operation of genetic algorithms. While the GA engine is mechanical in nature, making it independent of the problem space it is evolving through, the fitness function of the population members forms the core of the process. It was quickly discovered that the greatest challenge in building a successful GA is building the fitness function. Even assuming knowledge of the factors that affect the fitness function, balancing these factors against each other can prove to be the most challenging task.

Ironically, it may require a GA to search the potentially infinite problem space of fitness factor weightings in order to find the ideal weightings for a given problem type. This could also be built into the GA engine without there being any need to understand the problem being addressed. It would require an extension to the GA interface so that a population member not only needs to be able to get a chromosome, set a chromosome, and return a fitness value, but also needs to be able to get and set population fitness factor weights.

Evolutionary strategies do hold promise for problems where the number of potential solutions grows exponentially and there is no known way to traverse the problem space efficiently. They offer a more disciplined approach than random searches. The proof that these de-centralized, non-global, emergent processes do actually work surrounds us in the natural world. The prime examples are the two great stochastic

processes: evolution and learning. The difficulty is in determining the base rules in order to guide the emergent result. Perhaps the answer is another layer of abstraction: an algorithm to explore and discover the appropriate set of base rules.

## REFERENCES

- Bateson, Gregory. (1979/2002). *Mind and Nature: A Necessary Unity*. Cresskill, NJ: Hampton Press, Inc.
- Belew, Richard K. (1991). Artificial Life: A Constructive Lower Bound for Artificial Intelligence. *IEEE Expert*, 6, 1, (February 1991) pp.8-15.
- Dawkins, Richard. (1986). *The Blind WatchMaker*. London: Penguin Books.
- Dawkins, Richard. (1989). *The Selfish Gene*. Oxford: Oxford University Press. (Original work published 1976)
- Di Battista, Giuseppe, Didimo, Walter, Patrignani, Maurizio, Pizzonia, Maurizio. (2002). Drawing Database Schemas. *Journal of Software – Practice and Experience*, 32, 11, (2002) pp.1065-1098.
- Goldberg, David E. (1989). *Genetic Algorithms: In Search, Optimization & Machine Learning*. Reading, MA: Addison-Wesley Publishing Company, Inc.
- Goodrich, Michael T., Tamassia, Roberto. (2001). *Data Structures and Algorithms in Java, Second Edition*. New York: John Wiley & Sons Inc.
- Hearst, Marti, Hirsh, Haym. (2000). AI's Greatest Trends and Controversies. *IEEE Intelligent Systems*, 15, 1, (January/February 2000) pp.8-17.
- Hobbs, M.H.W, Rodgers, P.J. (1998). Representing Space: A Hybrid Genetic Algorithm for Aesthetic Graph Layout. *FEA'98 Frontiers in Evolutionary Algorithms in Proceedings of JCIS'98 The Fourth Joint Conference on Information Sciences*, 2, (October 1998) pp.415-418.
- Hofstadter, Douglas R., Dennett, Daniel C, eds. (1981). *The Mind's I: Fantasies and Reflections on Self and Soul*. New York: Basic Books.
- Johnson, Steven. (2001). *Emergence: The Connected Lives of Ants, Brains, Cities, and Software*. New York: Scribner.
- Kennedy, James, Eberhart, Russel C. (2001). *Swarm Intelligence*. San Diego: Morgan Kaufman Publishers.
- Koza, John R. (1992/1994). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.
- Kurzweil, Ray. (1999). *The Age of Spiritual Machines: When Computers Exceed Human Intelligence*. New York: Penguin Books.

- Langton, Christopher, ed. (1992). *Artificial Life VI: A Proceedings Volume in the Santa Fe Institute Studies in the Sciences of Complexity*. Don Mills, Ontario: Addison-Wesley Publishing Company, Inc. (Original work published 1989)
- Masui, Toshiyuki. (1994). Evolutionary Learning of Graph Layout Constraints from Examples. *Proceedings of the 7<sup>th</sup> annual ACM Symposium on User Interface Software and Technology*, (November 1994) pp.103-108.
- Wilson, Robin J., Watkins, John J. (1990). *Graphs: An Introductory Approach*. New York: John Wiley & Son Inc.

## APPENDIX A

### ERD VIEWER USER'S GUIDE

#### Launching the Application

ERD Viewer is a java swing application compiled under JDK1.4. The application entry point is through the class:

```
com.myidb.ERDLayout.ERDViewer
```

The application has been packaged into an executable jar file named ERDViewer.jar. The application can be launched by double clicking the jar file in a Windows environment or by executing the following command at the command prompt of any other system supporting JDK1.4:

```
java -jar ERDViewer.jar
```

#### Manipulating the Schema

The ERD Viewer consists of four main visual components:

- a menu bar at the top for selecting commands
- a toolbar used for configuration changes and located below the menu bar
- an expandable tree on the left side of the window to view and manipulate the schema
- a plot area on the right side of the screen to view the layouts

By default, the application is loaded with a test schema which is visible in the schema tree and consisting of four entities and four relations. Components may be added or removed from the tree by selecting an item in the tree and right-clicking to bring up a context sensitive pop-up menu. For example, selecting the "Tables" node found beneath



“Schema” and right-clicking will allow the addition of a new table to the schema.

Selecting a table, such as the “Employee” table will allow the options to either add a column to the table or to drop the table entirely.

Going to the menu bar and selecting “File – Load Schema from File” will allow loading a new schema based on comma separated text files for the tables and relations. This process could be easily extended to load the schema directly from the system catalogue of a database server.

### Configuring the GA Engine

Three buttons are available on the toolbar for configuring the GA Engine:

- **G** Use this option to set the number of generations the GA Engine should evolve.
- **P** Use this option to set the population size.
- **M** Use this option to set the mutation rate. The number should be between 0 and 1 and represents a percentage chance of a gene mutating

### Adjusting Fitness Values

The “Config” button on the toolbar allows the manipulation of fitness weights. A sample configuration can be viewed in Figure 6.

**Figure 6. Fitness factor configuration options**

Section	Option	Value
Entity Base	Enable	<input checked="" type="checkbox"/>
	Entity Base	50
	Entity-Entity Intersect Factor	5
	Entity-Edge Intersect Factor	4
	Unused Area Factor	0.01
Relation Base	Enable	<input checked="" type="checkbox"/>
	Relation Base	50
	Edge-Edge Intersect Factor	8
	Orthogonal Factor	1
	Slope Factor	0.1

All weights are multiples of the base value. For example, in the above configuration, the fitness will commence with 50 points for each entity. Each entity-entity intersection will result in a penalty of 5 times the base value, or 250 points. Each entity-edge intersection will result in a penalty of 4 times the base value, or 200 points. For each percentage difference between the total area available and the total area occupied, a bonus of 0.01 times the base value will be awarded. A layout occupying 30% of the area would score  $(100-30) * 0.01 * 50 = 35$  points. The Origin Translation factor works in a similar fashion, except it calculates occupied area from the origin to the  $\max(x)$  and  $\max(y)$  values of the layout rather than between  $\min(x)$ ,  $\min(y)$  and  $\max(x)$ ,  $\max(y)$ . The purpose of this is to encourage translation towards the origin.

The relational weightings work in a similar manner. For each relation, an initial bonus of 50 points is awarded. Each edge-edge intersection costs a penalty of 8 times the relational base value, or 400 points. Each orthogonal line is awarded 1 times the base, or 50 points. Each time the slope doubles in magnitude (when above 1), or halves in magnitude (when below 1), an additional 0.1 times 50 points is scored. For example, an edge with a slope of 32 would score as follows: 0.1 times 50 = 5 points for reaching a slope of 2, another 5 points for reaching a slope of 4, another 5 points for 8, another for 16, and yet another for 32. The total score for a slope of 32 ( $2^6$ ) would be  $6 * 5 = 30$  points. This type of iteration is capped at 8 levels.

Any of these factors can be included or excluded from the fitness calculation by selecting or de-selecting the “enabled” checkboxes.

### Generating Layouts

With a schema loaded, the GA Engine configured, and the fitness weights adjusted, layouts can be evolved. This is done by clicking on the “Evolve” button on the toolbar. On completion, the “fittest” layout is plotted on the display. The “Previous” and “Next” buttons can be used to step through the entire population. To view how the running total fitness of any particular member was calculated, click on the “Fitness” button. An entirely new population can be evolved by clicking on the “Evolve” button again.

## APPENDIX B

### ERD VIEWER DESIGN OVERVIEW

#### GA Engine

The GA Engine is based on a generic genetic algorithm as described in Goldberg's *Genetic Algorithms*. The implementation consists of the class GAEngine and the interface GAMember found in the com.myidb.GA package.

The GAEngine generates a population of GAMembers, and then repetitively iterates through the following core evolutionary tasks:

1. **getChromosomes**: get the binary strings representing each population member
2. **calculateFitness**: determine the fitness of each member
3. **calculateChance**: determine likelihood of a member to breed based on fitness
4. **getBreeders**: select reproducing members based on calculated chances
5. **crossOver**: mate the selected breeders
6. **mutate**: let the new population undergo random mutation
7. **setChromosomes**: replace the population with the new generation

GAMember describes the interface that anyone wishing to use the GAEngine must implement. Any population member must be able to provide it's own fitness value. Any member must be able to present a chromosome string representing its structure, and be able to accept a chromosome to change it's structure. The member also has the option to perform fine tuning (annealing) after it's structure has been changed. By implementing this interface, the GAEngine has enough information to build and evolve populations.

```
public interface GAMember {
```

```

public int getFitness();

public String getChromosome();

public void setChromosome( String chromosome );

public void anneal();

}

```

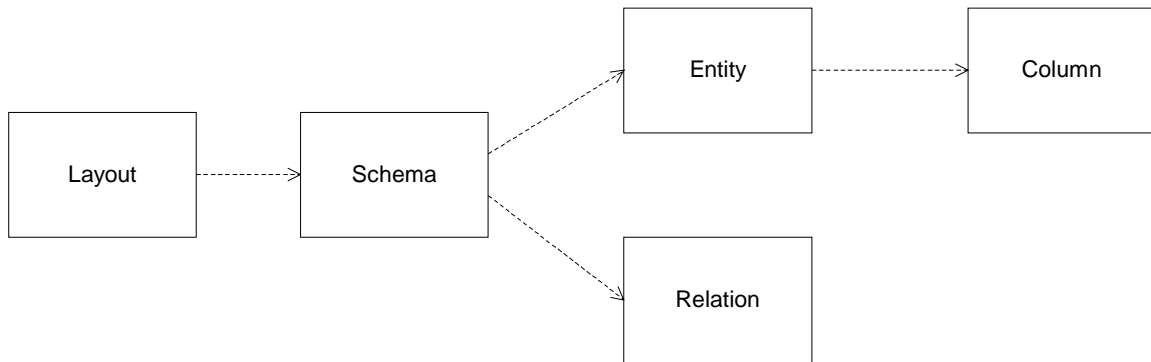
## ERD ADT

Data structures to store the ERD information are located in the `com.myidb.ERDLayout` package. Table 1 summaries the data structures and Figure 7 shows the relationships between the classes.

**Table 1. The ERD ADT**

<b>Class Name</b>	<b>Description</b>
Column	A Column represents a column in a database table and as such acts as a composite member of the Entity class. A Column may have many attributes such as name, size, data-type, isMandatory, etc. Some of these attributes may be desirable to display on an ERD. This implementation only collects the name of the column.
Entity	An Entity maintains an ordered list of Columns for a database table. The list is defined by the order of insertion with no duplicate values allowed.
Relation	A Relation represents a foreign key relationship between two tables. It has a source table, a destination table and a name. Logically, the relation is an edge on a graph. Physically, it is a line on a plot. Relation can be expanded in the future to include "from" and "to" column names.
Schema	A Schema maintains a set of "Entity"s and a set of Relations representing the tables and relations of a database schema.
Layout	A Layout maintains the graphical co-ordinates for plotting a schema. The entity co-ordinates consist of (x,y) locations of the top left corner of each entity. The relation co-ordinates consist of an edge indicator (top, right, bottom, left) and an offset. Two such pairs are required for a relation: one for positioning on the origin entity, and the other for positioning on the destination entity.  A Layout is also the implementor of the GAMember interface.

**Figure 7. The ERD ADT class diagram.**



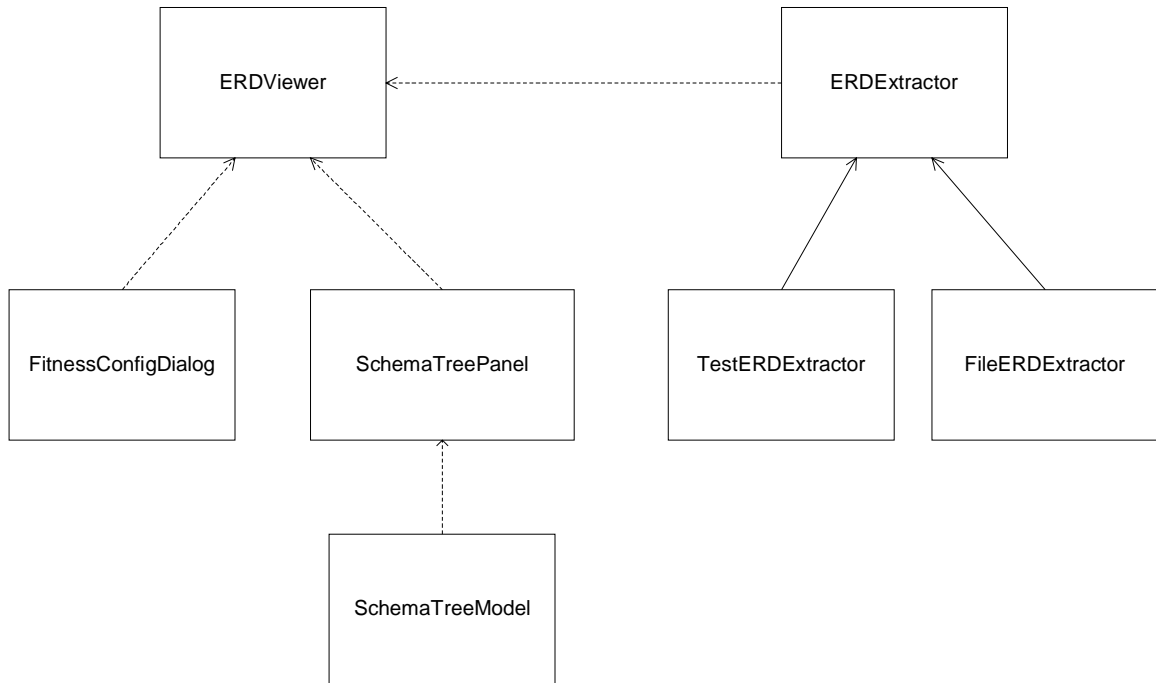
### GUI Components

ERD Viewer is a java swing application compiled under JDK1.4. Many of the forms were generated using the NetBeans IDE with the intention of auto-generating as much gui code as possible. The classes listed in Table 2 form the core gui components for the ERD Viewer, while Figure 8 shows the relationships between these classes.

**Table 2. The ERD gui components**

<b>Class Name</b>	<b>Description</b>
ERDExtractor	An interface to obtain ERD information from external sources. Implementations may include comma separated files or data dictionaries. Two known implementations: FileERDExtractor and TestERDExtractor.
FileERDExtractor	An implementation of the ERDExtractor interface, this dialog is used to load schema information from a text file.
TestERDExtractor	An implementation of the ERDExtractor interface, this class is used to load a default sample test schema.
ERDViewer	The application entry point, ERDViewer is a gui framework for loading, displaying, and evolving layouts for a schema.
FitnessConfigDialog	The FitnessConfigDialog allows the user to modify the weights associated with calculating the fitness of a Layout. The weights are associated with either Entities or with Relations.
SchemaTreeModel	A java TreeModel ADT implementation based upon a Schema object and used to support a SchemaTree.
SchemaTreePanel	The SchemaTreePanel contains a JTree for a schema. It provides functionality for modifying the schema by the use of pop-up menus when right-clicking on elements in the graphical tree.

**Figure 8. The ERD gui class diagram**



### Graph ADT

The implementation of the Graph ADT is based on Goodrich and Tamassia's *Data Structures and Algorithms in Java*. The ERDGraph was originally proposed to support what has become the Schema class. The graph ADT is not in use in the current version of the program, but is suggested to be re-implemented in a future version. Figure 9 shows the relationship between the graph ADT classes.

**Figure 9. The graph ADT class diagram**

