To the Graduate Council:

I am submitting herewith a thesis written by Liliya Goldshmid entitled "Extinction, speciation and morphological evolution on a holey hypercube." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mathematics.

Sergey Gavrilets, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor and
Dean of The Graduate School

EXTINCTION, SPECIATION AND MORPHOLOGICAL EVOLUTION ON A HOLEY HYPERCUBE

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Liliya Goldshmid

August 1998

ACKNOWLEDGEMENTS

I owe my thanks to many people who contributed in various ways toward completion of this thesis.

First of all, I would like to thank you my committee members, Dr. Gavrilets, Dr. Rosinski, and Dr. Vose for their patience, guidance and help in the development and preparation of this thesis.

I would like to thank all of my friends for being there for me during my years in graduate school.

Last but not least, I would like to thank all of my family members for their encouragement and their faith in me. Without their support this thesis would never be possible.

ABSTRACT

Scientists agree that many animal groups exhibit rapid initial morphological diversification, and then the process slows down. In his work Mike Foote presented a model and simulations, which show the same behavior. However, in order to achieve these results, he had to adjust morphological steps to slow diversification process down. This thesis presents an alternative model, which does not require changing the size of morphological steps. We develop analytical results for the simpler models, which deal only with mutation, and numerical results for more complex models involving mutation, speciation, and extinction. This thesis also contains a program, which simulates mutation, speciation, and extinction on a hypercube with holes, and algorithms for implementation of the program. Using this program one can easily observe how the parameters, like percentage of holes of a hypercube, speciation rate, and extinction rate, affect the dynamics of the population size and morphological diversification. The results from the simulations and analysis are also presented in this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE **PAGE**

CHAPTER 1

Introduction

1.1 Background

Studying the paleontological data, scientists concluded that the history of species is not random. For several major animal groups morphological diversification was the most rapid early in the history, and then the process tended to slow down. Examples include bryozoans [1], gastropods [14], crinoids [7], blastozoans [8], and marine arthropods [2]. It also has been shown that initially morphological diversity among the species increases with the larger rate than taxonomic diversity. In other words, the differences between the species increase faster than their number. These tendencies are clearly shown in the study of Blastozoa by Mike Foote.

In Foote's study of Blastozoa sixty-five characters describing a species were defined. Most of them were binary. The difference between two traits in species is zero for matching pair of characters and one for the mismatching pair. Morphological distance (or dissimilarity) between two species is measured as a sum of differences over all traits, and disparity of the clade is measured as the mean pairwise dissimilarity among all of the species in each time period. For this study 147 species were used, and the time interval covered was about 330 million years [6, 8].

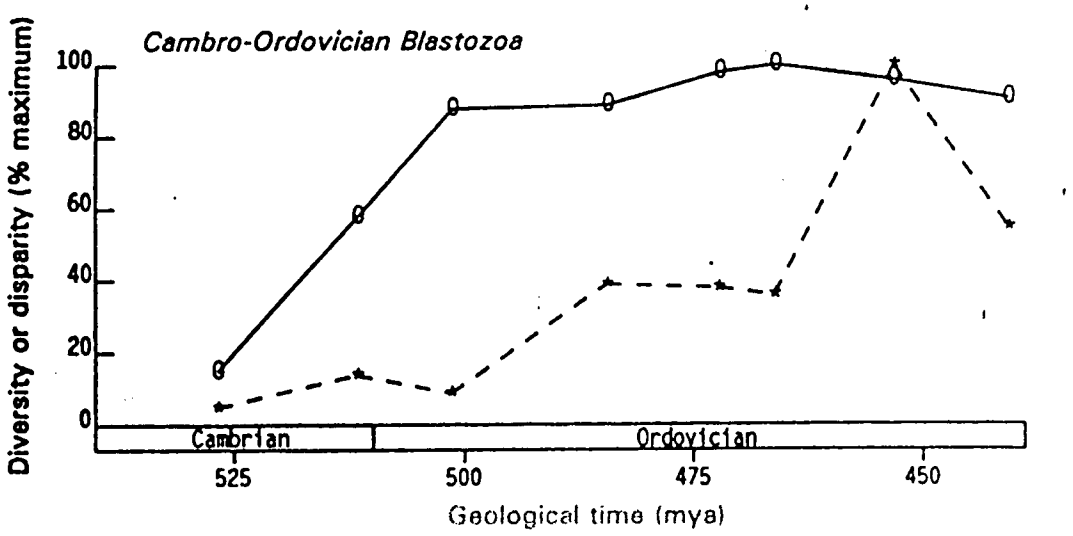


Figure 1.1 Diversification of Blastozoan Echinoderms.

From "Models of Morphological Diversification" by Mike Foote [2]

Solid line shows morphological distance between genera and dashed line indicates the number of genera.

The results of the study are shown in the Figure 1.1. It can be easily seen that morphological distance between the species, represented by solid line, grows initially faster than the number of genera, shown by dashed line.

There are two leading explanations to the phenomenon described above. First one claims that at first small number of species occupied large range of morphological space, and rapid colonization of empty morphological space took place; hence, causing initial rapid morphological diversification. Low levels of competition and selection at initial stages, when the number of species was relatively small, allowed wide ranges of forms

[5, 13]. The other states that increased complexity of genetic and developmental system slows down morphological diversification [8].

In his paper, *Models of Morphological Diversification*, Mike Foote describes several different simulations of morphological and taxonomic diversification, which were designed to produce results similar to those seen in nature. Each simulation began with a single lineage with arbitrary assigned morphology. In the first set of simulations, the speciation and extinction rates were constant over time, and morphological steps stayed the same through the simulations. The results showed almost liner increase in morphological diversification. In order to slow down the process of morphological diversification, the second set of simulations was adjusted in such a way that the first 50 million years the morphological steps are five times as large as morphological steps in the last 50 million years. This way the graphs showed rapid initial diversification and a reduced rate later on [6].

The goal of this thesis is to develop a model, which would describe this phenomenon of the rapid initial morphological diversification, and which wouldn't require "artificially" adjusting the morphological steps.

1.2 Thesis overview

Random diffusion on a hypercube with holes represents a model for macroevolutionary changes [4]. We would like to develop some intuition about evolution, speciation, and extinction modeled this way, and to answer the following

questions: how the percentage of holes, birth, death, and mutation rates affect the dynamics of the population size and morphological differences of the population.

Two different approaches are used in order to reach this goal: analytical one for the simple models and computational one for more complex models. In chapter 2 we will analyze simple models, and compare the analytical results with numerical simulations. In chapter 3 we will discuss the theoretical ways to speed up the simulations. Chapter 4 will deal with implementation of the program, which simulates the random diffusion on hypercube with holes. Finally, in chapter 5 we will see the results of simulations and the answers to the questions posted above as well as compare our results with paleontological data.

1.3 General description of the model

Morphological space can be mathematically considered to be equivalent to hypercube. Hence, we model evolution, extinction, and speciation as a random diffusion on a hypercube with holes. Borrowing the idea of binary characters for describing the traits, we will represent each phenotype that has L traits by a binary string of the length L: $r_j = (r_{j,0}, r_{j,1}, ..., r_{j,L-1})$, where $r_{j,k}$ is 0 or 1.

Thus, each species can be viewed as a particle located on a vertex of a hypercube. When a species mutates, i.e. one of the entries of the string changes to the opposite one, a particle moves to an adjacent vertex of a hypercube. Holes on the hypercube represent combination of traits with low fitness. If a given vertex is a "hole", the combination of

the traits, which that vertex represents, has fitness that is not viable. If a vertex is not a "hole", the corresponding combination of traits is viable. We simulate speciation by adding an exact copy of a parent string to the population and letting new species to mutate independently from the parent. Extinction is simulated by removing strings from the population. Hence we have four parameters: percentage of the holes on a hypercube, p, mutation rate, μ, speciation rate, σ, and extinction rate, δ. And we would like to find out how they affect morphological diversity, D, population size, N, and the average distance between the species and the founder of the clade, d.

Translating with model into mathematical language we get the following:

1. Morphological difference between two species is measured by the number of mismatched traits and can be calculated using the formula

$$d(r_i, r_j) = \sum_{k=0}^{L-1} r_{i,k} \oplus r_{j,k}$$

where $\oplus$ denotes exclusive-or (i.e., addition modulo 2).

Hence, morphological diversity of a population of a size N is given by

$$\mathrm{D} = \frac{2}{N(N-1)} \sum_{i>j} d(r_i, r_j)$$

2. Our population is growing exponentially; thus at time t the size of the population size is $N = N_0 e^{(\sigma - \delta)t}$

where N_0 is the initial size of the population.

3. The average distance of the clade from the founder, d can be calculated as

$$d = \frac{1}{N-1} \sum_{i=1}^{N-1} d(r_0, r_i).$$

Note that the main difference between our approach and the simulation described in the Foote's article is that he ran simulations for a single trait, while we have L binary traits. Now we are ready to consider the simple models.

CHAPTER 2

Analyzing Simple Models

2.1 Overview

The evolution of artificial populations, where individual species may be represented as binary strings, may be modeled by three operations. First is *mutation*, which changes one or more entries in a string. Second is *speciation*, which adds a new string to the population. And the last operation is *extinction*, which removes a string from the population. Working with a model of this type, one may be interested in the mean and variance of the distance between the species as a function of time.

In this chapter we will consider a simple model, which deals only with mutation. We start out with two species. At time t = 0, these two species are exactly alike, i.e. the morphological difference between the species *i* and *j*, D, is 0. Then at each time step we randomly select one of the species and try to mutate one randomly selected trait. If a new specie has the fitness allowing it to survive, then mutation is successful, and we measure the morphological difference them. Otherwise, mutation doesn't take place, and morphological difference doesn't change.

Hence, we model a random walk on a hypercube with holes. Hence, for this problem we have mutation rate $\mu = 1$, speciation rate $\sigma = 0$, and extinction rate, $\delta = 0$,

and $N_0 = N = 2$ (since we don't have speciation and mutation, number of species stays constant).

Let p be percentage of holes. We assume that they are uniformly distributed. Although, in our simulations holes are actually randomly distributed, for this model we assume certain kind of symmetry: each vertex has the same percentage of holes around it. At each time step we randomly pick one of these two particles and try to move it to a randomly chosen adjacent vertex. If that vertex is not a hole, we complete the move. However, if the vertex where tried to move is a hole, a particle does not move, and the distance does not change at this time step.

2.2 Finding expected distance at time t

We would like to be able to predict the morphological difference between these two species at time t. In other words, we would like to know the expected distance between the particles at time t. Let D_t be the distance at time t. We also assume that the probability of the distance at a time t + 1 being j, Pr $\{D_{t+1} = j\}$, depends only on the distance at time t, i.e. D_t. Thus, the conditional probability of D_{t+1} being j, given that D_t was k, is

$$\Pr\{D_{t+1} = j \mid D_t = k\} = \begin{cases} p, & \text{if } j = k \\ (1-p)\dfrac{k}{L}, & \text{if } j = k-1 \\ (1-p)\left(1-\dfrac{k}{L}\right), & \text{if } j = k+1 \end{cases}$$

Then the expected distance at time t +1 is given by the following:

$$E[D_{t+1} \mid D_t = k] = k\,p + (k-1)(1-p)\frac{k}{L} + (k+1)(1-p)\left(1-\frac{k}{L}\right)$$

$$= k\,p + \frac{k^2}{L}(1-p) - \frac{k}{L}(1-p) + k(1-p)\left(1-\frac{k}{L}\right) + (1-p)\left(1-\frac{k}{L}\right)$$

$$= kp - 2(1-p)\frac{k}{L} + k(1-p) + (1-p)$$

$$= k\left[p - 2(1-p)\frac{1}{L} + 1 - p\right] + (1-p)$$

$$= k\left[1 - \frac{2(1-p)}{L}\right] + (1-p)$$

Hence, $E[D_{t+1}] = E[D_t]\left[1-\frac{2(1-p)}{L}\right] + (1-p)$. Now let q be equal to $1-p$ and $E[D_t] = M_t$. Then we have the following:

$$\begin{cases} M_{t+1} = M_t\left(1-\frac{2q}{L}\right) + q \\ M_0 = 0 \end{cases}$$

Solving the equation above for M_t, we get

$$M_t = q\left[\left(1-\frac{2q}{L}\right)^{t-1} + \left(1-\frac{2q}{L}\right)^{t-2} + \ldots + \left(1-\frac{2q}{L}\right) + 1\right] = q\sum_{i=0}^{t}\left(1-\frac{2q}{L}\right)^i$$

Simplifying the sum above, we get

$$M_t = \frac{L}{2}\left[1-\left(1-\frac{2q}{L}\right)^t\right]$$

Note that as $t \to \infty$, $M_t \to \frac{L}{2}$.

2.3 Finding the variance

Using the formula Var $(D_t) = E[D_t^2] - (E[D_t])^2 = E[D_t^2] - M_t^2$, we only need to find $E[D_t^2]$ since we already know M_t.

$$E[D_{t+1}^2] = k^2 p + (k-1)^2 \frac{k}{L}(1-p) + (k+1)^2 (1-p)\left(1-\frac{k}{L}\right)$$

$$= k^2\left(1-\frac{4q}{L}\right) + 2qk + q$$

where $q = 1 - p$.

$$\text{Hence, } E[D_{t+1}^2] = \left(1-\frac{4q}{L}\right)E[D_t^2] + 2E[D_t]q + q = \left(1-\frac{4q}{L}\right)E[D_t^2] + 2M_t q + q .$$

Let $S_t^2 = E[D_t^2]$. Then we have

$$\begin{cases} S_{t+1}^2 = \left(1-\frac{4q}{L}\right)S_t^2 + 2M_t q + q \\ S_0^2 = 0 \end{cases}$$

Solving this for S_t^2, we get $S_t^2 = \sum_{i=1}^{t}\left(1-\frac{4q}{L}\right)^{i-1}(2M_{t-i}q + q)$ or

$$S_t^2 = q\sum_{j=0}^{t-1}\left(1-\frac{4q}{L}\right)^{j}(2M_{t-1-j} + 1)$$

$$= q\sum_{j=0}^{t-1}\left(1-\frac{4q}{L}\right)^{j}\left[(L+1) - L\left(1-\frac{2q}{L}\right)^{t-1-j}\right]$$

$$= q(L+1)\sum_{j=0}^{t-1}\left(1-\frac{4q}{L}\right)^{j} - \frac{q(L-2q)^{t-1}}{L^{t-2}}\sum_{j=0}^{t-1}\left(\frac{L-4q}{L-2q}\right)^{j}$$

Simplifying the sums above,

$$\sum_{j=0}^{t-1}\left(1-\frac{4q}{L}\right)^j = \frac{1-\left(1-\frac{4q}{L}\right)^t}{1-\left(1-\frac{4q}{L}\right)} = \frac{L}{4q}\left(1-\left(1-\frac{4q}{L}\right)^t\right) \quad \text{and}$$

$$\sum_{j=0}^{t-1}\left(\frac{L-4q}{L-2q}\right)^j = \frac{1-\left(\frac{L-4q}{L-2q}\right)^t}{1-\left(\frac{L-4q}{L-2q}\right)} = \frac{L-2q}{2q}\left(1-\left(\frac{L-4q}{L-2q}\right)^t\right)$$

substituting them back and simplifying the expression again, we get

$$S_t^2 = \frac{(L+1)L}{4} - \frac{L+1}{4L^{t-1}}(L-4q)^t - \frac{(L-2q)^t}{2L^{t-2}} + \frac{(L-4q)^t}{2L^{t-2}}$$

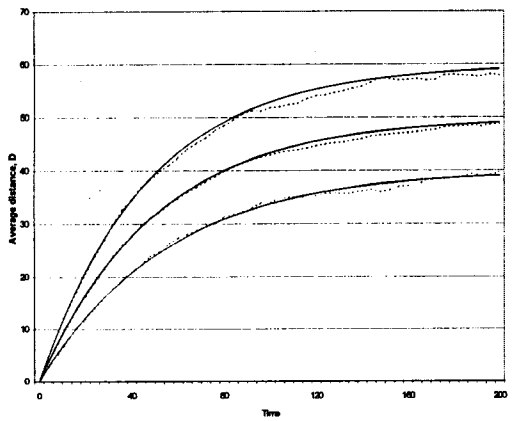
Now we can compute the variance by subtracting M_t^2 from S_t^2.

$$\mathrm{Var}(D_t) = \frac{L}{4} + \frac{(L-4q)^t(L-1)}{4L^{t-1}} - \frac{(L-2q)^{2t}}{4L^{2t-2}}$$

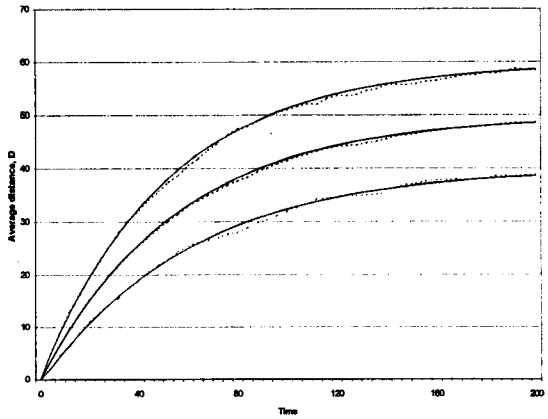
As $t \to \infty$, $\mathrm{Var}(D_t) \to \frac{L}{4}$.

The results of these formulas is compared with numerical results in the Figure 2.1. For numerical simulations we used $L = 100$, and p = 0, 0.1, 0.2, 0.3. 0.4, 0.5. 0.6 and 0.7. Each graph shows average distance and two standard deviations.

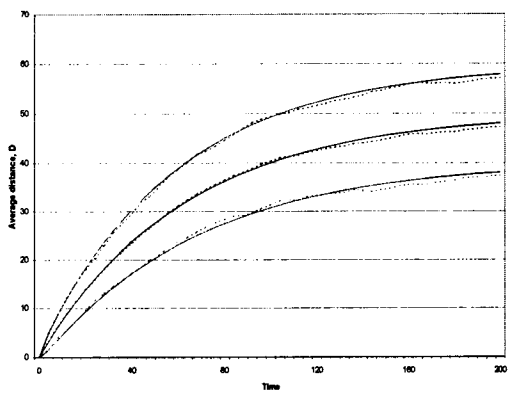
Numerical results are shown by dashed line, and solid lines represent results from the formulas. It's easy to notice that for the relatively small percentages of holes on a hypercube, i.e. for p = 0, p = 0.1, and p = 0.2 the formula describes the behavior of of the distance quite well.



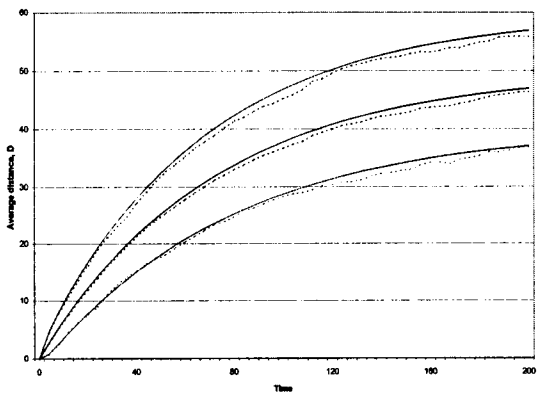
a). p = 0.0



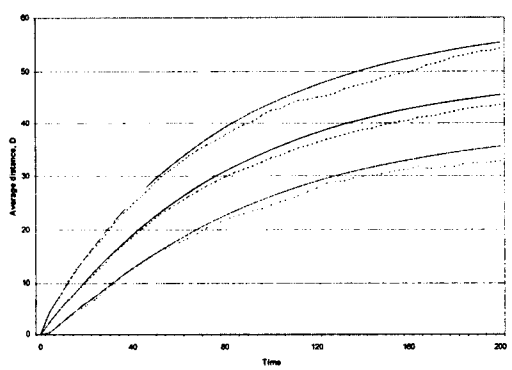
b). p = 0.1



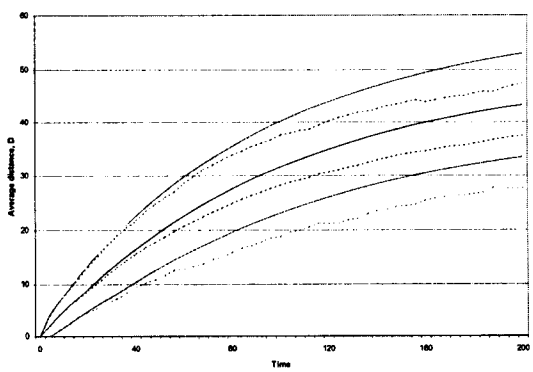
c). p = 0.2



d). p = 0.3



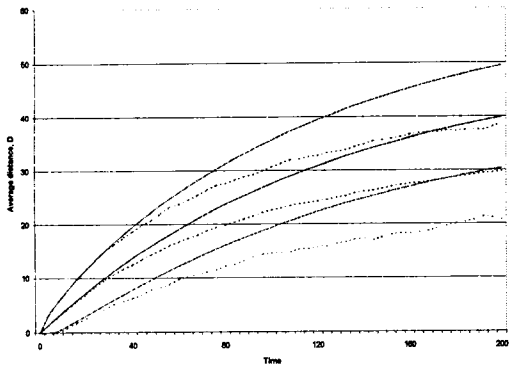
e). p = 0.4

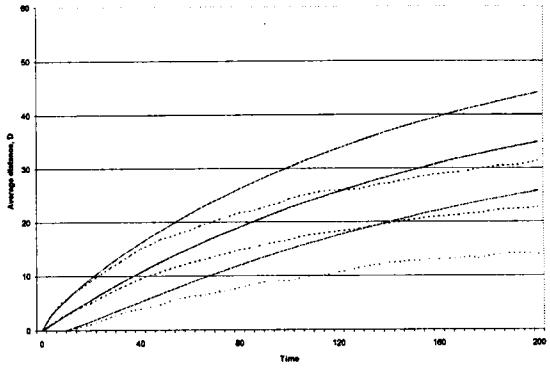


f). p = 0.5

Figure 2.1 Comparing the results from the formulas with numerical results

Numerical results are shown by dashed line, and solid lines represent result from the formulas.





g). p = 0.6 h). p = 0.7

Figure 2.1 (continued) Comparing the results from the formulas with numerical results. Numerical results are shown by dashed line, and solid lines represent result from the formulas. Center lines show average distance, while the rest of the line represent two standard deviations.

However, starting with p = 0.3 and for the larger values of p, the results from the formula differ from the numerical results. This happens because the holes are not actually uniformly distributed through the hypercube. And hence, we have to do numerical simulations.

CHAPTER 3

Numerical Algorithm for Computing the Mean and the Variance

3.1 Overview

Recomputing the average morphological diversity and variance after each operation is straightforward, but time consuming: the computational expense is $O(N^2L)$, where L is the string length and N is the number of species. Since the population size is growing exponentially, simulations take long time to complete as the number of species increases. Hence, we needed a new way for computing the average morphological diversity and variance of entire population without figuring out morphological difference for each pair of species in the population.

In this chapter an alternative way of incrementally computing mean and variance of morphological diversity of a population will be presented. The time needed to compute the new mean using this method is O(1) in the case of mutation, and is $O(N)$ in the case of speciation and extinction. For the variance, the time needed is $O(N)$ for all three operations.

Let M be a $N \times L$ matrix representing the population. The rows $r_0, \ldots, r_{N-1}$ of M represent the population members. Define the distance between r_i and r_j to be

$$d(r_i, r_j) = \sum_{i=0}^{L-1} r_{i,k} \oplus r_{j,k}$$

where $\oplus$ denotes exclusive-or (i.e., addition modulo 2). The average distance, D, between different rows of M is

$$D = \frac{2}{N(N-1)} \sum_{i>j} d(r_i, r_j)$$

We are interested not only in the average distance, but also in the variance $Var(D)$, defined by

$$Var(D) = \frac{2}{N(N-1)} \sum_{i>j} (D - d(r_i, r_j))^2$$

In a situation where p is changing at discrete times, say $t_1, t_2, t_3, \ldots$, both D and $Var(D)$ are functions of t. Although the computation of D and $Var(D)$ is straightforward, it could be more efficient to consider how they change from time t_i to time t_{i+1}, rather than to recompute them.

Changes to M can take various forms. For instance, N and L could remain the same, while the entries in M undergo change. Since any such change to M can be thought of as the result of a series of basic changes that affect only a single entry, it suffices to consider the case where a single bit of M changes. In section 3.2 change of this type is considered. Another type of change to M would be that N changes. As before, such changes involving several rows can be thought of as the result of a series of basic changes, which in this case affect N by either increasing or decreasing it by 1. In section 3.3 changes of this type are considered.

3.2 Recomputing the mean and the variance after a mutation

The average distance D between different rows of M is $D = 2\xi / (N^2 - N)$, where

$$\xi = \sum_{i>j} d(r_i, r_j) = \sum_{i>j} \sum_{k} r_{i,k} \oplus r_{j,k}$$

We consider the case where the single entry $r_{h,l}$ changes, i.e. if it was 1, it becomes 0 and vise versa. Let the original matrix be M with entries $r_{i,j}$, and let the new matrix (produced by the change) be M' with entries $r_{i,j}'$. Note that

$$r_{i,j}' = r_{i,j} \oplus [\, i = h \wedge j = l \,]$$

where square brackets as above are used to denote an indicator function: if *expr* is an expression which may be true or false, then

$$[expr] = \begin{cases} 1, \text{if } expr \text{ is true} \\ 0, \text{if } expr \text{ is false} \end{cases}$$

After the change $M \rightarrow M'$, the new mean is $D' = 2\xi' / (N^2 - N)$ where ξ' *is*

$$\begin{aligned}
&\sum_{i>j} \sum_{k} \left(r_{i,k} \oplus [i = h \wedge k = l] \right) \oplus \left(r_{j,k} \oplus [j = h \wedge k = l] \right) \\
&= \sum_{i>j} \sum_{k} r_{i,k} \oplus r_{j,k} \oplus [k = l \wedge (i = h \vee j = h)] \\
&= \xi + \sum_{i>j} r_{i,l} \oplus r_{j,l} \oplus [i = h \vee j = h] - r_{i,l} \oplus r_{j,l} \\
&= \xi + \sum_{i>j} \left(r_{i,l} \oplus r_{j,l} \oplus 1 - r_{i,l} \oplus r_{j,l} \right) [i = h \vee j = h] \\
&= \xi + \sum_{j \neq h} r_{h,l} \oplus r_{j,l} \oplus 1 - r_{h,l} \oplus r_{j,l}
\end{aligned}$$

Hence, $\xi' = \xi + \delta_\xi$, where the change δ_ξ in ξ is

$$\sum_{j\neq h} r_{h,l} \oplus r_{j,l} \oplus 1 - r_{h,l} \oplus r_{j,l}$$
$$= [r_{h,l} = 1]\sum_{j\neq h} r_{j,l} - 1 \oplus r_{j,l} + [r_{h,l} = 0]\sum_{j\neq h} r_{j,l} \oplus 1 - r_{j,l}$$
$$= (2r_{h,l} - 1)\sum_{j\neq h} r_{j,l} - 1 \oplus r_{j,l}$$

The above sum is $\sum_{j\neq h} r_{j,l} - \sum_{j\neq h} 1 \oplus r_{j,l} = a_l - b_l + (1 - 2r_{h,l})$

where a_l is the number of 1^s in column l and b_l is the number of 0^s in column l.

It follows that δ_ξ is $(2r_{h,l} - 1)(a_l - b_l + (1 - 2r_{h,l}) = (2r_{h,l} - 1)(a_l - b_l) - 1$

To speed up calculations, we store a_i for each column i. It follows that the change δ_D in D is

$$\delta_D = D' - D = 2((2r_{h,l} - 1)(2a_l - N) - 1)/(N^2 - N)$$

Therefore D' can be computed in time O (1), including the time to update a_l.

Next we consider the change in variance. Suppose M changes in position h,l to become M'. As before, $r'_{i,j} = r_{i,j} \oplus [i = h \wedge j = l]$. Let *Var(D)* and *Var(D)'* be the variances corresponding to M and M' respectively. Define η via the equality

$$Var(D)' - Var(D) = \frac{2}{(N-1)N}\sum_{i<j}\left((D' - d(r'_i, r'_j))^2 - (D - d(r_i, r_j))^2\right) = \frac{2}{(N-1)N}\eta .$$

Note that $d(r'_i, r'_j) = d(r_i, r_j)$ if neither i or j is h. Moreover,

$$d(r'_h, r_i) = \sum_{k=0} (r_{h,k} \oplus [k = l]) \oplus r_{i,k}$$
$$= r_{h,l} \oplus 1 \oplus r_{i,l} + \sum_{k\neq l} r_{h,k} \oplus r_{i,k}$$
$$= d(r_h, r_i) - r_{h,l} \oplus r_{i,l} + r_{h,l} \oplus 1 \oplus r_{i,l}$$
$$= d(r_h, r_i) + 1 - 2(r_{h,l} \oplus r_{i,l})$$

Therefore η is given by

$$\sum_{i<j}(D'^2 - D^2) + 2D\sum_{i<j} d(r_i, r_j) - 2D'\sum_{i<j} d(r_i', r_j') + \sum_{i\neq h} d^2(r_h', r_i) - d^2(r_h, r_i)$$

$$= \frac{N^2 - N}{2}(D'^2 - D^2) + (N^2 - N)D^2 - (N^2 - N)D'^2$$
$$+ \sum_{i\neq h}(d(r_h, r_i) + 1 - 2(r_{h,l} \oplus r_{i,l}))^2 - d^2(r_h, r_i)$$

$$= \frac{N^2 - N}{2}(D^2 - D'^2) + \sum_{i\neq h} 2d(r_h, r_i)(1 - 2(r_{h,l} \oplus r_{i,l})) + (1 - 2(r_{h,l} \oplus r_{i,l}))^2$$
$$= \frac{N^2 - N}{2}(D^2 - D'^2) + N - 1 + 2\sum_{i\neq h} d(r_h, r_i)(1 - 2(r_{h,l} \oplus r_{i,l}))$$

From previous considerations we know how to compute the change in D quickly. To speed up calculations of η we could save $d(r_i,r_j)$ for $i < j$, hence the sum $\sum_{i\neq h} d(r_h, r_i)(1 - 2(r_{h,l} \oplus r_{i,l}))$ can be computed in time O(N), assuming the distances were known. Note that for $i \neq j$, $d(r_i', r_j') - d(r_i, r_j) = [i = h](1 - 2(r_{h,l} \oplus r_{j,l})) + [j = h](1 - 2(r_{h,l} \oplus r_{i,l}))$. So a distance requires update only when $i = h$ and when $j = h$. To summarize, the change $\delta_{Var(D)}$ in *Var(D)* is

$$\delta_{Var(D)} = Var(D)' - Var(D) = D^2 - D'^2 + \frac{2}{N} + \frac{4}{N^2 - N}\sum_{i\neq h} d(r_h, r_i)(1 - 2(r_{h,l} \oplus r_{i,l}))$$

3.3 Recomputing the mean and the variance after speciation or extinction

In this section we consider changes that affect N. First, consider the change in D when N changes. Suppose we adjoin a row. Our new matrix M' has dimensions $N+1 \times L$. Using previous notation, the new value ξ' is given by

$$\xi + \sum_{j=0}^{N-1} \sum_{k} r_{N,k} \oplus r_{j,k}$$

$$= \xi + \sum_{k} \sum_{j=0}^{N-1} r_{N,k} \oplus r_{j,k}$$

$$= \xi + \sum_{k} [r_{N,k} = 0] a_k + [r_{N,k} = 1](N - a_k)$$

$$= \xi + \sum_{k} r_{N,k}(N - 2a_k) + a_k$$

Therefore,

$$D' = \frac{2}{(N+1)N} \xi'$$

$$= \frac{N-1}{N+1} \left(\frac{2}{N^2 - N} (\xi + \sum_{k} r_{N,k}(N - 2a_k) + a_k) \right)$$

$$= \left(1 - \frac{2}{N+1} \right) D + \frac{2}{N^2 + N} \sum_{k} r_{N,k}(N - 2a_k) + a_k$$

It follows that in the case of speciation the change in D is

$$\delta_D = D' - D = \frac{-2D}{N+1} + \frac{2}{N^2 + N} \sum_{k} r_{N,k}(N - 2a_k) + a_k$$

Now suppose we delete a row. Without loss of generality assume that we delete the last row. The new matrix M' has dimensions N-1 $\times$ L, and the new value ξ' is

$$\xi - \sum_{j=0}^{N-2}\sum_{k} r_{N-1,k} \oplus r_{j,k}$$

$$= \xi - \sum_{k}\sum_{j=0}^{N-2} r_{N-1,k} \oplus r_{j,k}$$

$$= \xi - \sum_{k} [r_{N-1,k} = 0]a_k + [r_{N-1,k} = 1](N - a_k)$$

$$= \xi - \sum_{k} r_{N-1,k}(N - 2a_k) + a_k$$

Therefore,

$$D' = \frac{2}{(N-1)(N-2)}\xi'$$

$$= \frac{N}{N-2}\left(\frac{2}{N(N-1)}(\xi - \sum_{k} r_{N-1,k}(N-2a_k) + a_k)\right)$$

$$= \frac{N}{N-2}D - \frac{2}{(N-1)(N-2)}\sum_{k} r_{N-1,k}(N-2a_k) + a_k$$

Hence, in the case of extinction the change in D is

$$\delta_D = D' - D = \frac{2}{N-2}D - \frac{2}{(N-1)(N-2)}\sum_{k} r_{N-1,k}(N-2a_k) + a_k$$

We now consider the change in *Var(D)* when a row is deleted. As above, we may assume without loss of generality that it is r_{N-1}. Note that $Var(D) = \frac{2}{N^2 - N}\sum_{i<j} d^2(r_i, r_j) - D^2$

Therefore,

$$\sum_{i<j<N-1} d^2(r_i, r_j) = \frac{N^2 - N}{2}(Var(D) + D^2) - \sum_{i<N-1} d^2(r_i, r_{N-1})$$

It follows that

$$Var(D)' = \frac{2}{(N-1)(N-2)}\left(\frac{N(N-1)}{2}(Var(D) + \mu^2) - \sum_{i<N-1} d^2(r_i, r_{N-1})\right) - D'^2$$

$$= (1 + \frac{2}{N-2})Var(D) + \frac{N}{N-2}D^2 - \frac{2}{(N-1)(N-2)}\sum_{i<N-1} d^2(r_i, r_{N-1}) - D'^2$$

The change $\delta_{Var(D)}$ in *Var(D)* is therefore

$$\delta_{Var(D)} = Var(D)' - Var(D) = \frac{2}{N-2} Var(D) + \frac{N}{N-2} D^2 - \frac{2}{(N-1)(N-2)} \sum_{i<N-1} d^2(r_i, r_{N-1}) - D'^2$$

Finally, we consider the case where a row is adjoined (corresponding speciation). Employing the same technique as above,

$$\sum_{i<j<N+1} d^2(r_i, r_j) = \frac{N^2 - N}{2} (Var(D) + D^2) + \sum_{i<N} d^2(r_i, r_N)$$

It follows that

$$Var(D)' = \frac{2}{(N+1)N} \left(\frac{N(N-1)}{2} (Var(D) + D^2) + \sum_{i<N} d^2(r_i, r_N) \right) - D'^2$$

$$= \left(1 - \frac{2}{N+1}\right) Var(D) + \frac{N-1}{N+1} D^2 + \frac{2}{(N+1)N} \sum_{i<N} d^2(r_i, r_N) - D'^2$$

The change $\delta_{Var(D)}$ in *Var(D)* is therefore

$$\delta_{Var(D)} = Var(D)' - Var(D) = \frac{N-1}{N+1} D^2 - \frac{2}{N+1} Var(D) + \frac{2}{(N+1)N} \sum_{i<N} d^2(r_i, r_N) - D'^2$$

Thus the time needed to compute the variance is O(*N*).

CHAPTER 4

The Program

4.1 Overview

I was trying to write a very generic program, which can be easily adapted for different simulation. Most of the parameters, like speciation, mutation, extinction rates, percentage of holes on a hypercube, number of time steps, and number of runs, as well as the frequency of the outputs, can be set by user at the run time or default values supplied by the program can be used. Also the program allows to chose between death-birth-mutation and birth-death-mutation orders. In this chapter we will go over the code description and the algorithms for the program and its functions.

4.2 Code description and algorithms

4.2.1 Main

Main module of the program performs initialization by calling *InitGlobals* function, after that it starts simulation runs, number of which is determined by the user. On each run program attempts to mutate traits chosen at random, add and remove new species. Results after successful runs (determined either by distance after the last

mutation, or by the final number of species) are summed up, and averages are outputted after all runs are done.

The algorithm of the main program follows.

```
Initialize global variables and coefficients
      For all runs
            Create and initialize (place on cluster) first specie
            Copy it into second specie
                  For specified number of time steps
                        For all available species
                              Try to mutate the specie
                              Change distances in the system, if
                              mutation was successful
                        Try to add/remove species
                  If run was successful, store add its results to the
                  grand totals
      Output averages
```

4.2.2 Function

This function, called *Function*, converts species' genotype into a number between 0 and 1. These numbers are uniformly distributed and consistently reproducible for the given genotype in a run. In other words, it assigns fitness to each of the genotypes. Using them, we can avoid saving the tables that tell us, which of the vertices of the hypercube are the "holes" and which are not. As the result of that, we can use this program for the hypercubes of higher dimensions. The algorithm implemented in this function was described in an article "Random walks in a closed space" by N. Lemke and I.A. Campbell [10]. Note, that the results of the function are platform-specific, as they depend on the machine's precision of "double" format. User can improve "randomness" of the results by increasing the number of times chaotic map operation is performed, but it will negatively affect performance of the program.

4.2.3 InitGlobals

The purpose of this function is to allocate memory and initialize global arrays, variables, and coefficients. At the beginning of the first run this function asks a user to enter new parameters and based on their values or default values allocates the memory for the arrays. The algorithm follows.

```
Calculate default number of trial steps for cluster test
Ask user for new values (or keep default)
Allocate and null memory for global arrays using user's defined
dimensions
Open output files
```

4.2.4. CheckSurvival

This function checks result from *Function* against user-supplied percentage of holes and decides if a given vertex is a "hole" or not. Based on that a mutation event either occurs or it does not. It uses the following algorithm:

```
Call Function to create a random value for a genotype
Return result of comparison
```

4.2.5. AddGenotype

This function adds a new specie to the system and calls the function *ChangeDistances*. Function *AddGenotype* is used to create the very first specie as well as in every speciation. Also it manages memory resources. It uses the following algorithm:

```
If speciation takes place, change distances in the system
If adding a new specie, allocate the memory
Do the speciation(copy the genotype)or null the memory for a new
genotype, which will be initialized later in InitGlobals
```

```
Increment counters
```

4.2.6. RemoveGenotype

The *RemoveGenotype* function is called when extinction takes place. It removes a species and adjusts the distance by calling *ChangeDistances*. It also helps to manage memory, by caching already allocated arrays of species' genotypes for future use. The algorithms of this function is very straightforward:

```
Change distances in the system
Move the specie in the used pool
Decrement counters
```

4.2.7. CleanAfterRun

Depending on the flag passed, this function either nulls global arrays and variables, or frees all allocated memory and closes output files at the end of all runs. It also plays important role in memory management by cleaning all memory structures between the runs. It uses the following algorithm:

```
Move all species in the used pool
Null all global structures
If final cleanup specified, free all allocated memory and close open
files
```

4.2.8. MakeMutationStep

This function attempts to make a mutation step for a species that passes to it. If mutation survives, i.e. the vertex where it tried to move is not a "hole", it stays. Otherwise, it gets reverted to the previous state. The algorithm follows:

```
If mutation probability test is satisfied
      Randomly pick a trait and mutate it
      If the mutation above does not survive, roll it back
```

Note that mutation probability test is satisfied if random number generator returns a smaller number than the mutation rate specified by the user.

4.2.9. InitializeGenotype

This function is used to initialize species' genotype and to perform cluster test. It creates a species randomly and checks to make sure that this species is not on a "hole". If it is, the function creates a new species and continues to do so until it creates one, which is not located on a "hole". Then the function checks it for being on the giant cluster. If it's on the giant cluster, then the function ends, if not, the function starts over from the beginning. The algorithm follows.

```
While not on cluster
      Keep initializing the genotype until starting configuration
      survives
      Store the starting configuration
      Conduct cluster test
Restore the starting configuration
Initialize starting arrays
```

Note, that cluster test is declared successful if, within maximal allowed time set by the user, as the result of mutations, the species' distance to the starting configuration will be greater or equal to the minimally acceptable number set by the user (both maximal steps number and minimal distance are calculated by formulas based on the genotype's dimension and number of holes).

4.2.10. ChangeDistances

This function updates the distances among all mutating genotypes and to the starting point. Based on which of the events takes place – birth, death, or mutation – it updates distance according to the formulas described in chapter 3, and then updates the global arrays. This is the only function in the program, which actually changes the distances, the rest of them call this function when distances need updating.

4.2.11 TryAddRemoveGenotype

This function goes through all available species and tries to proliferate and/or kill them in turn, updating distances as necessary by calling *ChangeDistances* by using the following algorithm:

```
For all species
      If birth/death order is chosen
            If birth probability is satisfied a copy of the species is
            added
            If death probability is satisfied, the species is removed
      Else, if death/birth order is chosen
            If death probability is satisfied, the species is removed
            If birth probability is satisfied a copy of the species is
            added
```

Note that birth (death) test is satisfied if random number generator returns a smaller number than the birth (death) rate specified by the user.

4.2.12. PrepareForNextStep

This function copies all data necessary for the next run.

4.3 Parameters

There are two sets of the parameters in this program. First set is the parameters set in the program and the second set is the parameters set by a user at the run time.

4.3.1 Parameters set in the program

While doing the simulations one may want to change the parameters related to the fitness function. It's easy to do so within a program, but the program has to be recompiled after that. Changing these values, changes the performance of the function *Function*. Since we wanted these parameters to be the same throughout all of the simulations, we are not setting them up during runtime.

4.3.2 Values that can be set by the user

CUTOFF is percentage of holes on the hypercube, p. It changes between 0 and 1.
DIMENSION is length of string, which represents a species, L, or the dimension of the hypercube. The default value is 100.
MUTATIONS is number of time steps in a run. We each time step is 0.1 million years.
STEPSOUT is indicating how often to output the results, i.e. 1 means do the output every time step, 2 means every other time step and so on.
RUNS is a number of different runs in a simulation. The grand total is averaged over all runs. The default value is 200

BIRTHCUTOFF is a speciation rate, σ. A new species is created if random number generator returns a number smaller than BIRTHCUTOFF. Values used for different simulations were 0.032 and the multiples of it [6].

DEATHCUTOFF is an extinction rate, δ. A species is removed if random number generator returns a number which is smaller than DEATHCUTOFF. Values used for different simulations were 0.025 and the multiples of it [6].

MUTATCUTOFF is a mutation rate, μ. An attempt to make a mutation is made only if random number generator returns a value smaller than MUTATCUTOFF. Values used for different simulations were (0.001 * DEFDIMENSION) and 1 [6].

TRIALDISTANCE is a trial distance. It indicates how far the specie has to get from the starting point in TRIALSTEPS number of steps for it to be on the giant cluster. The default is DIMENSION/2 - DIMENSION/10.

TRIALSTEPS is maximal allowed number of steps for cluster test. Computed as an average value of numbers for densely and scarcely populated matrices: (T + T1)/2, where T = -L/(2 * p) * ln(1 - (2 * d)/L) (formula for the dense cluster), T1 = ((2 * d) ^ 2)/p (formula for sparse cluster) and L = DIMENSION, p = 1 - CUTOFF, mu1 = mu*p = p/L.

To test out the algorithm described in the chapter 3 and this program we recreated the simulations described in the article "Random walks in a closed space" by N. Lemke and I. A. Campbell [10], and received the results described in there.

CHAPTER 5

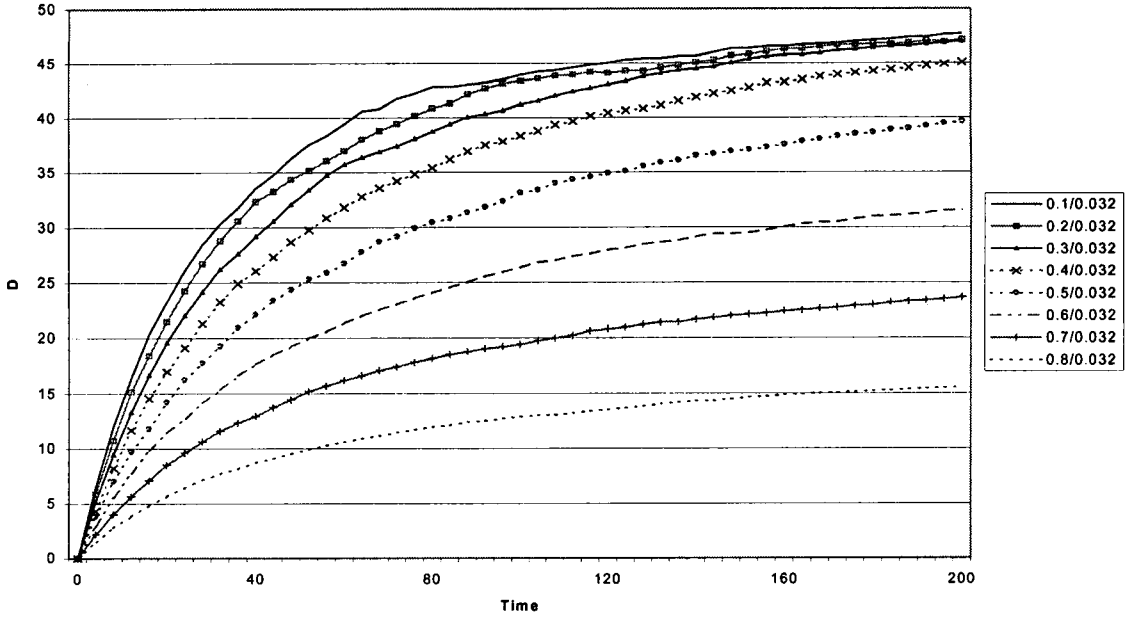
Analysis of the Simulation Results

5.1 Overview

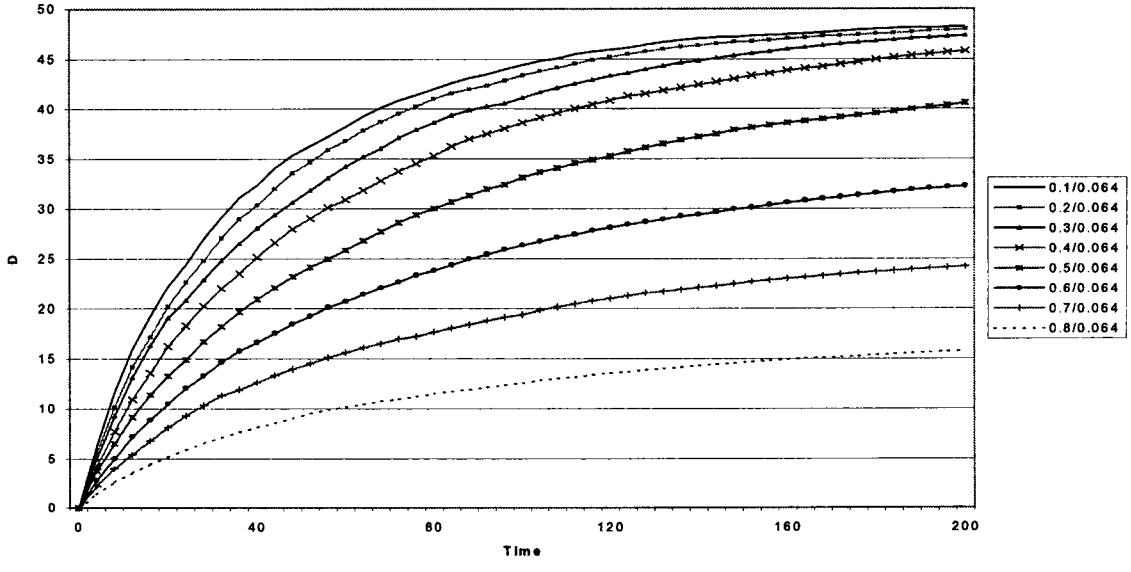
In this chapter we will show how different extinction, speciation and mutation rates affect the distance between the species, the distance from the starting point and the number of species. In all of our simulations we used L equal to 100. We start by considering the effects of different speciation rates.

5.2 Effects of different speciation rates

First we would like to see how different speciation rates affect the morphological diversity between species. So, we start by running the program for different percentages of holes on the hypercube and for different speciation rates. The extinction rate is $\delta = 0$, i.e. we don't have death events. The first choice for the birth rate is $\sigma = 0.032$ per lineage per 0.1 million years [6]. And then we try the multiples of it. As one can see in the Figures 5.1(a) and 5.1(b) the shapes of the graphs are very similar for two different birth rates. If we graph the results from different birth rates on the same graph, the behavior will become more obvious.



a). $\sigma = 0.032$



b) $\sigma = 0.064$

Figure 5.1 Effects of different birth rates $\sigma = 0.032$ and $\sigma = 0.064$

The legends for the graphs are given in p /σ , the percentage of holes on a hypercube / speciation rate, format.

As we start out, the distances between the species with a smaller birth rate are greater. However, as time goes along, the distances between the species with larger birth rate start to "catch up", and soon become greater. One can see that in the Figure 5.2, where solid lines show birth rate of 0.032 and birth rate of 0.064 is shown by dashed lines. Intuitive explanation is simple. At the beginning the distances between the particles on a hypercube are small, and adding additional particles will bring the average down. However, as time goes along and the particles get sufficiently far away from each other, adding a particle adds additional large distances. Hence, having large speciation rate slows the process initial and then speeds it up.

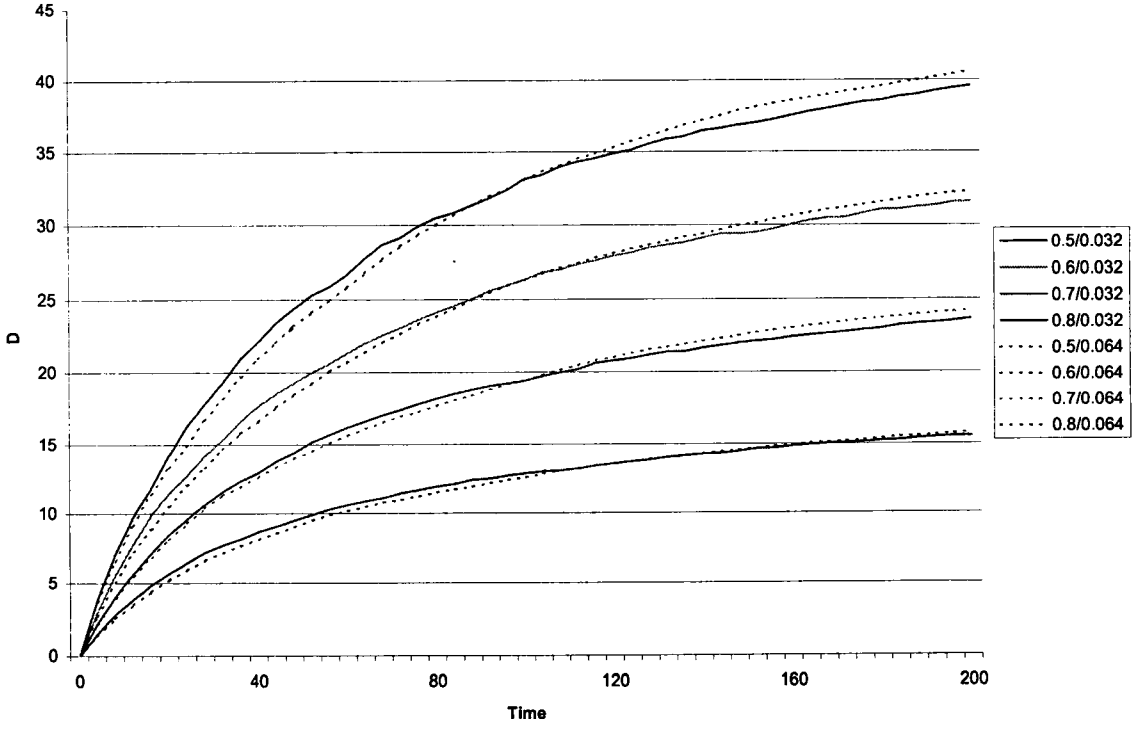


Figure 5.2 Comparing different speciation rates. Legend is in p/σ format.

Next we would like to consider the relationship between the average distances between the species and the average distances from the starting point. Logically, the distance between the species should increase faster, since at each mutation step it is possible to increase the distance by two for each pair of species. While the distance between each species and its starting point can increase only by 1. The Figure 5.3 supports this logic. Note that eventually both types of distances tend to the same values. As the percentage of holes on a hypercube increases, the average distances increase more slowly.

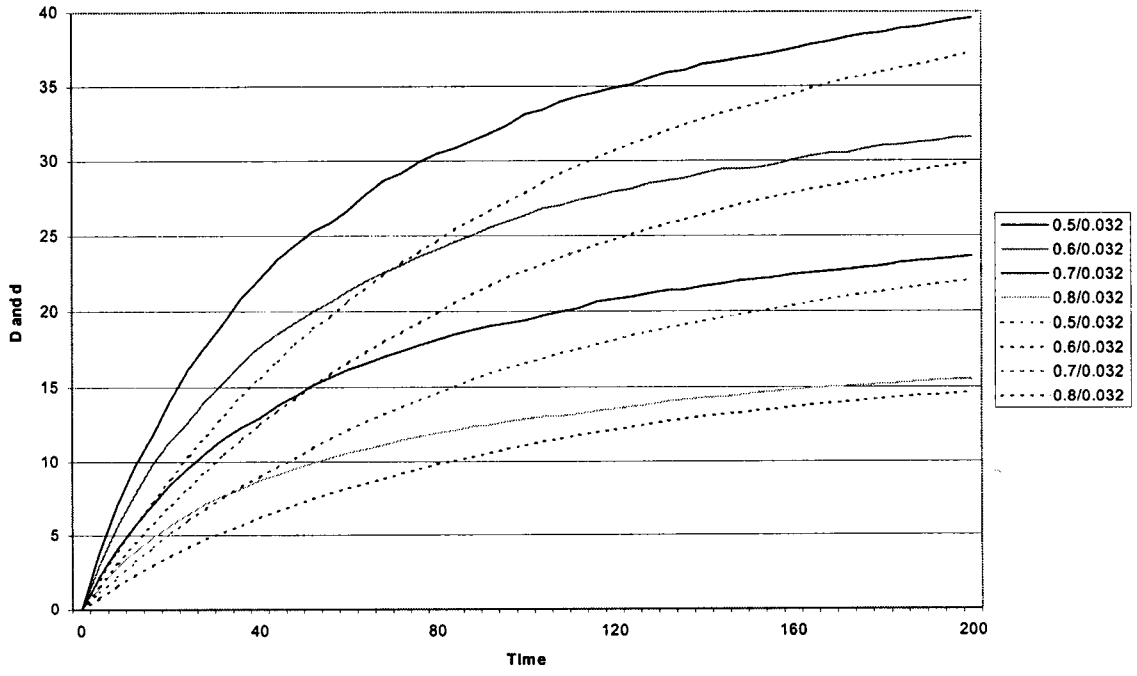


Figure 5.3 Comparing the average distances between the species with the average distances from the starting point for different percentages of holes on a hypercube. The average distances from the starting point are shown by dashed lines, and the average distances between the species show by solid lines. The legend is given in the p /σ format.

Now consider what effect the birth rate has on the number of species. The logical answer is that the higher the speciation rate the larger the number of species. Birth rate and the number of species do not depend on the number of holes on a hypercube. The population growth exponentially, according to the formula

$$N = N_0 e^{(\sigma - \delta)t}$$

The results of different birth rate shown in the Figure 5.4. The birth rates simulated are σ = 0.032, 0.064 and 0.096. Simulating higher speciation rates requires a lot of memory, and slows down the program significantly, since the number of species gets very large fast.

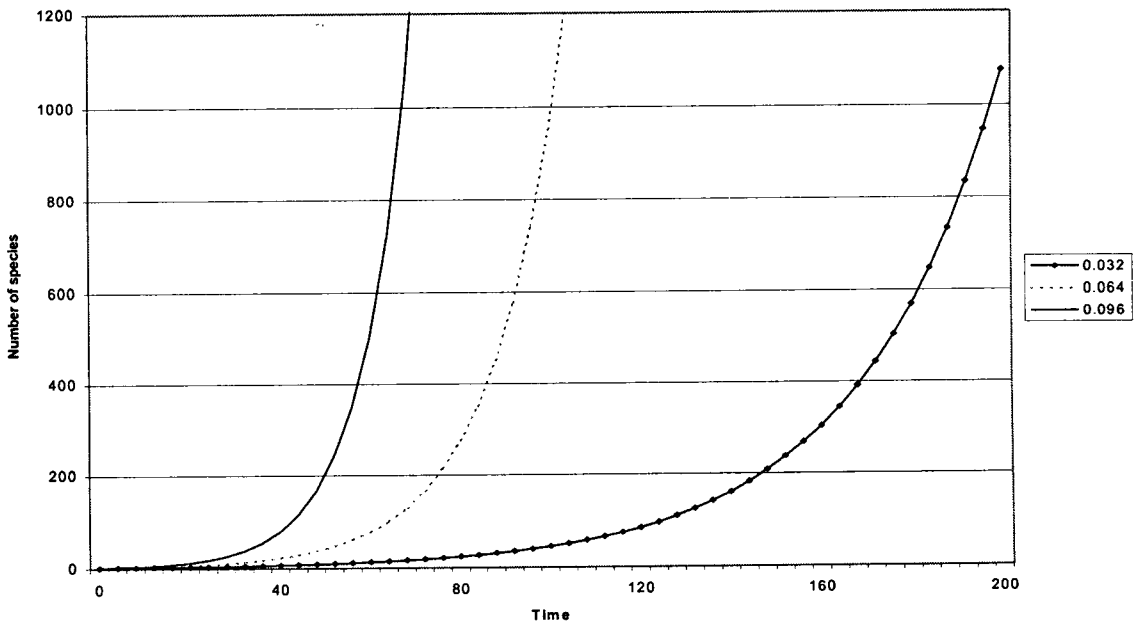


Figure 5.4 Number of species for different birth rates. Legend shows corresponding σ rates.

5.3 Effects of different speciation and extinction rates

Next we will discuss how speciation and extinction rates combined affect the average distances between the species and the average distance from the starting point. For these simulations the speciation rates used are $\sigma = 0.032$ and multiples of it, and death rates used are $\delta = 0.025$ and its multiples [6]. Throughout each simulation these rates stay the same. In all simulation speciation rates kept higher than extinction rates to keep the population from dying out.

First we consider different combinations of speciation and extinction rates for the same percentage of holes on a hypercube. The differences between the distances are not very significant, but we can clearly see that largest distances produced by the largest difference between the birth and death rates. In the cases, where the difference between the rates was close, the simulations produce very similar results, which can be seen in the Figure 5.5.

Analyzing the results of simulations we can clearly see that the percentage of holes on a hypercube plays larger role than extinction and speciation rates combination. To see this, we graph the average distance between the species for different percentages of holes on a hypercube and the same speciation and extinction rates. As percentage of holes increases, the average distances increase more slowly. This is illustrated in the Figure 5.6.

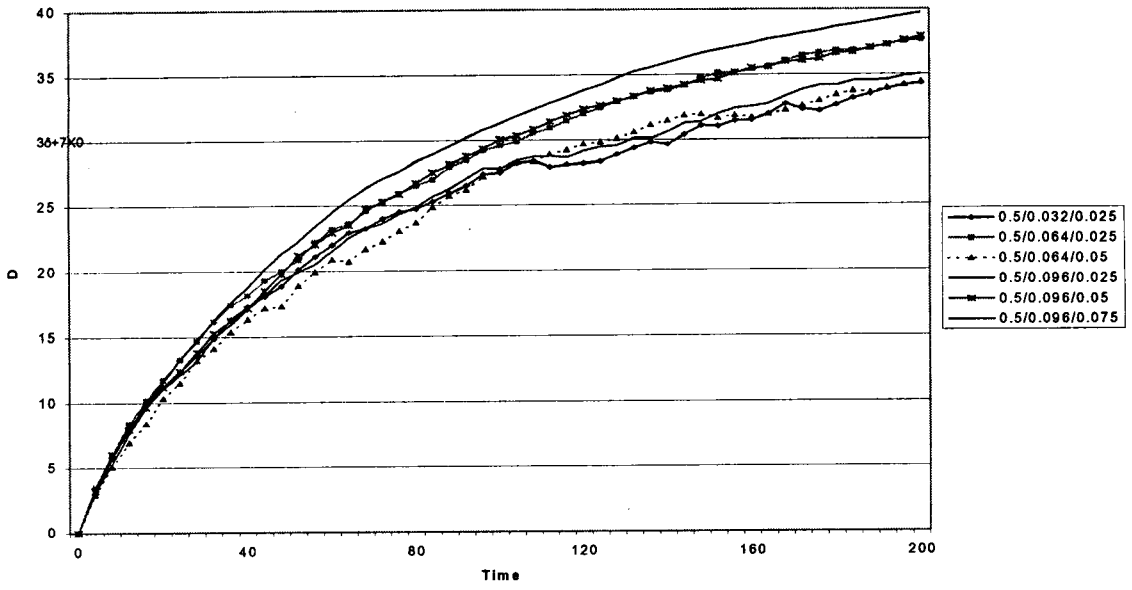


Figure 5.5 Effect of different speciation and extinction rates on D.

The legend for the graph is given in the p/σ /δ format.

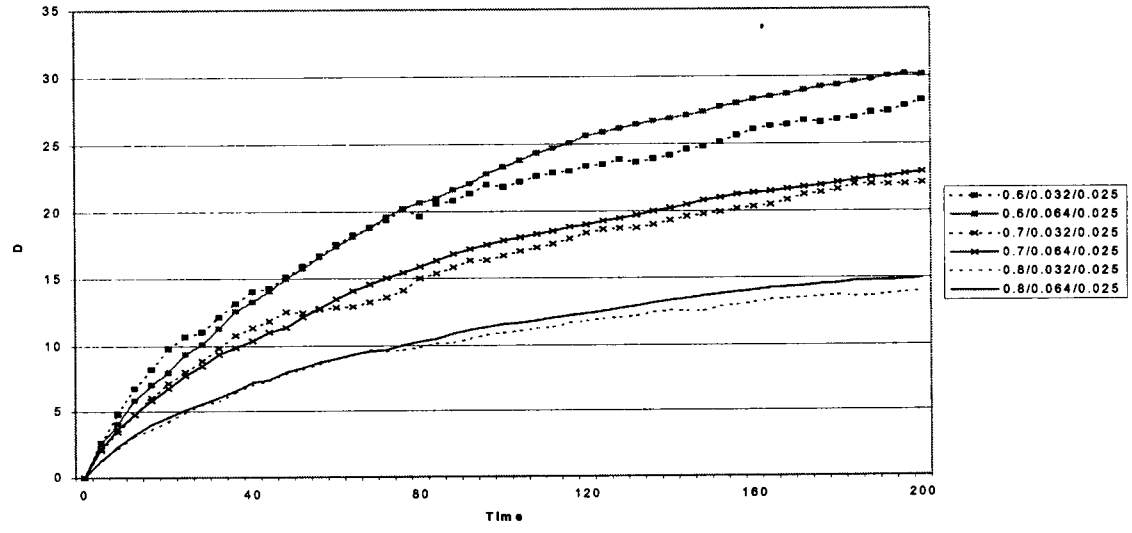


Figure 5.6 Effects of different speciation and extinction rates on D for different percentages of holes on a hypercube.

The legend for this graph is given in the p/σ /δ format.

The number of species in each case is affected by the difference between speciation and extinction rates. The largest numbers produced by the birth rate of $\sigma = 0.096$ and death rate of $\delta = 0.025$. The second largest numbers resulted from birth rate of $\sigma = 0.096$ and death rate of $\delta = 0.05$. And the smallest numbers were the result of $\sigma = 0.032$ and $\delta = 0.025$ rates. These results can be seen in the Figure 5.7.

5.4 Effects of different mutation rates

In all of the previous sections, the probability of mutation was 1, i.e. we attempted to mutate randomly chosen species at each time step. Now we would like to consider, what happens if we change the probability of mutation.

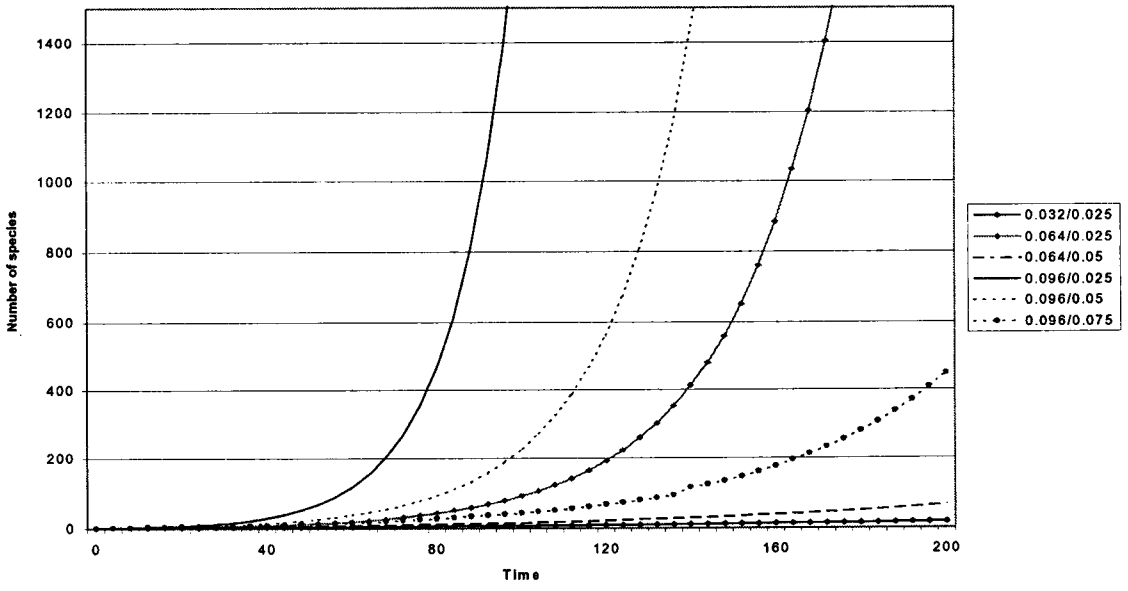


Figure 5.7 The number of species for different speciation and extinction rates. Legend is given in the format σ/δ.

First of all, since the distance changes most rapidly, when mutation takes place, we have to increase the number of time steps, as the process will slow down, when the probability of mutation decreases. We start by trying the probability of mutation 0.001 per trait [6], and for comparison we try the probability of mutation 0.005 per trait. The rest of the parameters is kept constant: speciation rate of 0.032 and extinction rate of 0.025. In the Figure 5.8 the results for p = 0.5 are shown. It is clear that the smaller the probability of mutation, the slower the distances grow.

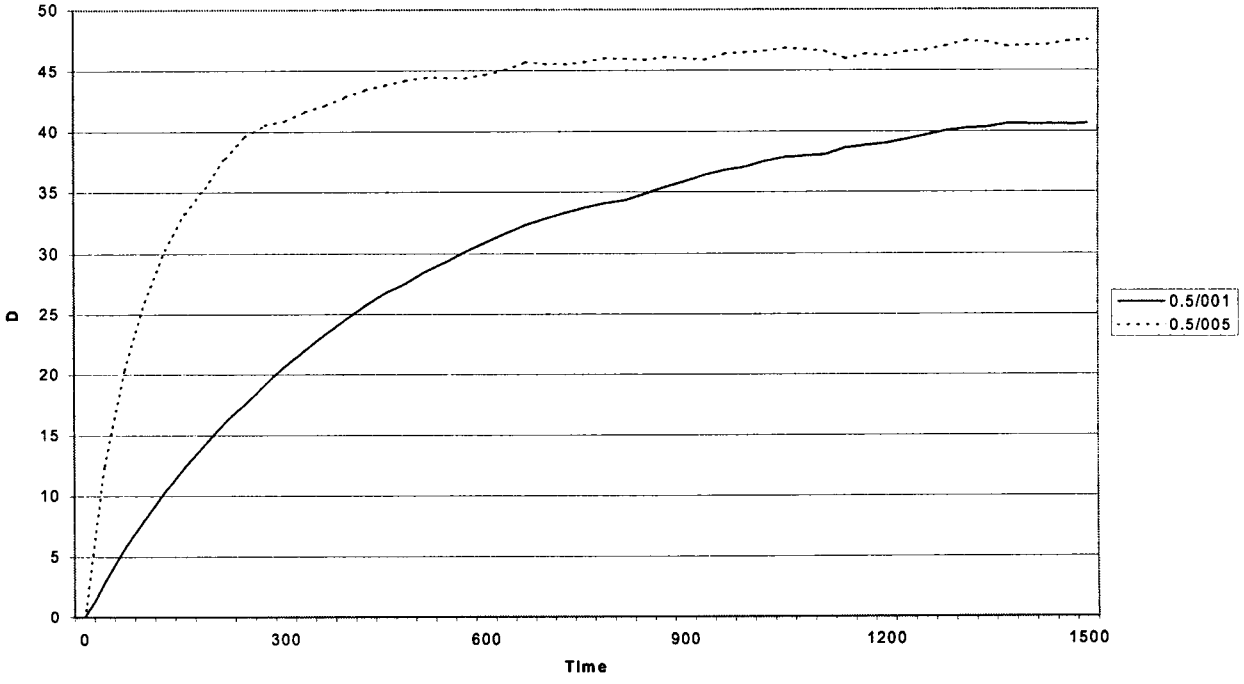


Figure 5.8 Effects of different mutation rates on D. The legend is in the p / μ format

However, as the time passes, the slower mutation rates start to "catch up" and eventually they tend to the same values. This is shown in the Figure 5.9. In that figure it also can be seen that the percentage of hole on a hypercube plays the most important role in the distance between the species. Although, the mutation does affect them, as time goes along its effect is not that significant.

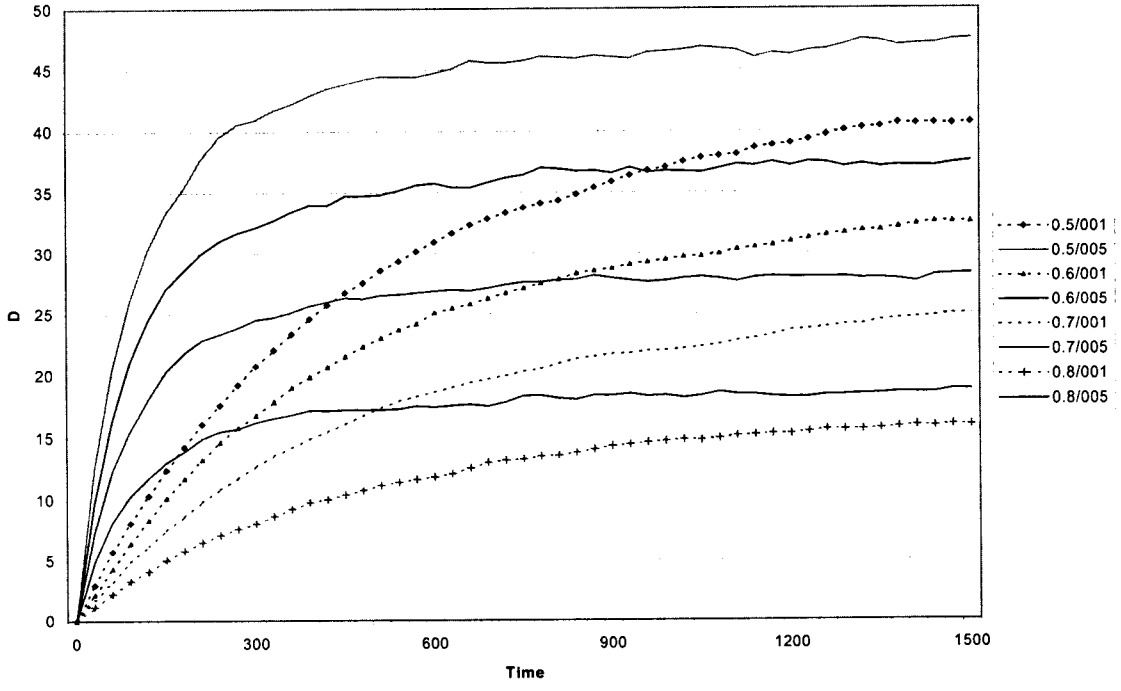


Figure 5.9 Different mutation rates for different percentages of holes. The legend is given in p / μ.

5.5 Comparing results with paleontological data

As we already noted in the introduction paleontological data suggests that morphological diversification is rapid at the beginning, and then it slows down. Our model produced exactly the same results, as can be seen by comparing the figures in this chapter to Figure 1.1. Hence, by modeling mutation, speciation, and extinction as a random diffusion of a hypercube with hole we get results similar to the exiting paleontological data. And our model does not require the adjusting of the time scale.

As we start out with 2 species that are exactly the same, every time we mutate a randomly chosen trait, the distance between the species will increase, since the probability of choosing the trait that is the same in both species and making it different is high. Hence, initially morphological difference growth rapidly. As time passes more and more traits become different as the result of mutations. Therefore, probability of choosing a trait that is the same in both species decreases, and so does the speed of morphological diversification. Hence, morphological diversification slows down.

It's worth noting that N. Lemke and I. A. Campbell got stretched exponential behavior for the graph of the overlap plotted on the log-log scale in their simulations, given by the formula $q(t) = A\exp(-(t/\tau)^{\beta})$, where t is time and A, τ, and β are the parameters [10]. Repeating their experiments we also got similar results, which can be seen in the Figure 5.10. Overlap, *q(t)*, is related to the distance from the starting point by the formula $q(t) = 0.5 - \frac{d}{L}$. However, when we graphed overlap for the data from our experiments on the log – log scale, we did not receive stretched exponential graph. The

stretched exponential graph supposed to look like a straight line on the log-log scale for large values of t, and our results do not look that way. That can be seen in the Figure 5.11. This discrepancy is left for the future studies.

5.6 Summary

In preparation for this thesis I studied already existing models of evolution. Using some ideas from Foote's articles, in this thesis we developed a model, which allowed us to describe paleontological phenomenon of rapid initial morphological diversification without "artificially" slowing down the process by adjusting the time scale.

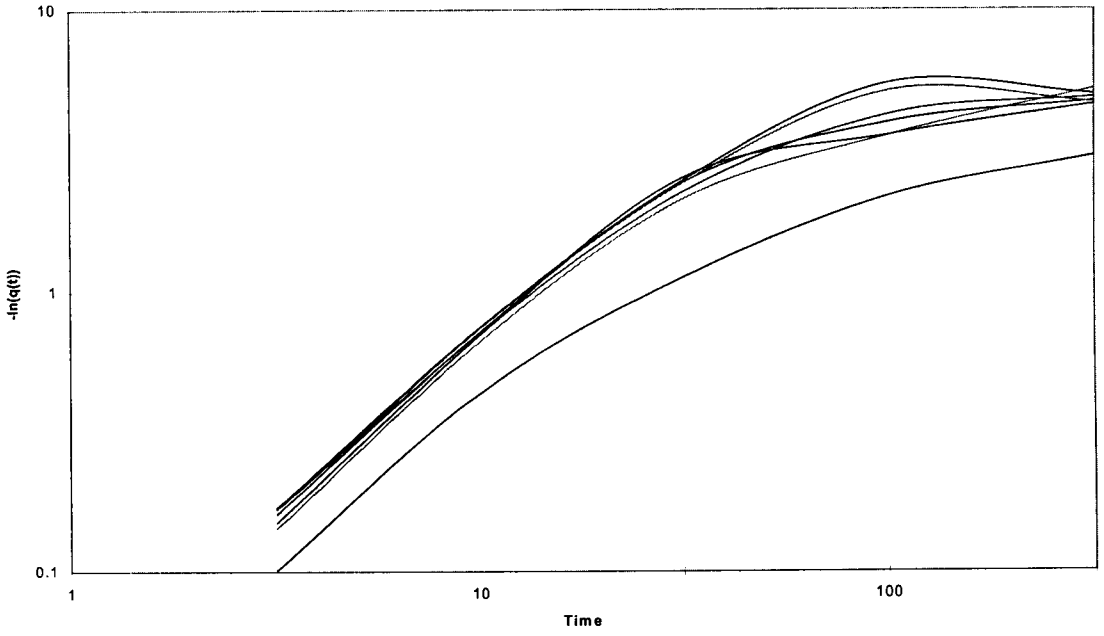


Figure 5.10 Recreating the results of N. Lemke and I.A. Campbell.

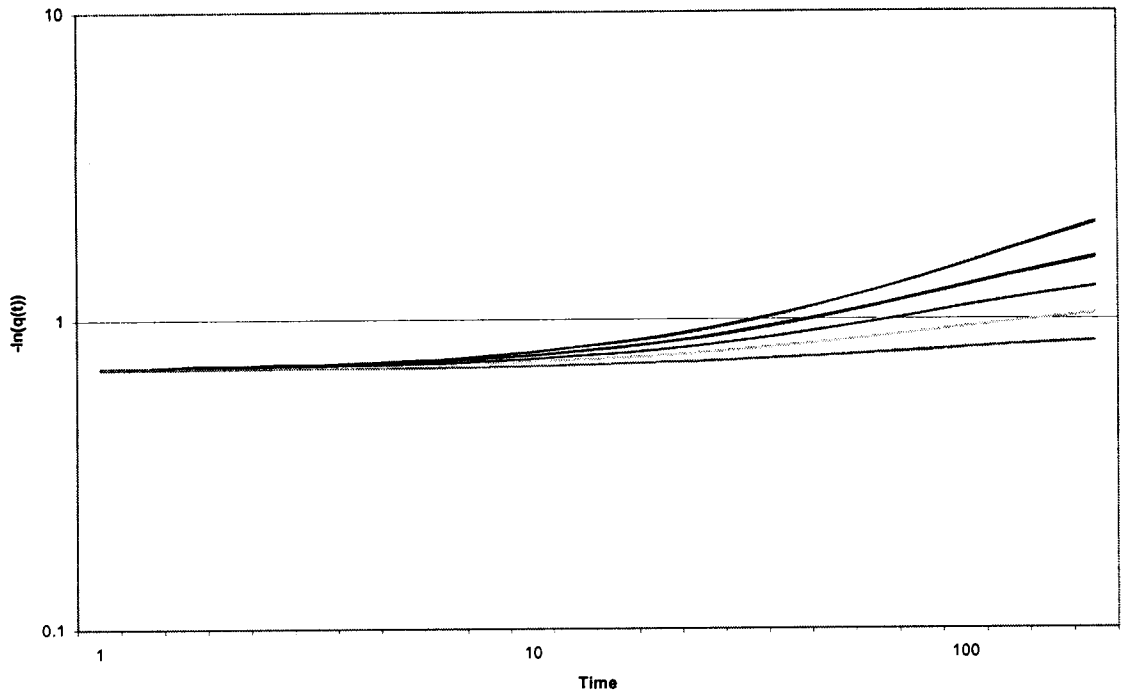


Figure 5.11 Overlap from our simulations on log-log scale

In our model, this process slows down, although the morphological steps do not change during simulations. We studied the simple models, which involved only mutation and no extinction or speciation, and developed analytical results for them. From those results one can easily see that the morphological distances between the species tend to the number equal to the half of the number of traits.

Also we developed a numerical algorithm allowing us to speed up the simulations for this model significantly. Using this algorithm we do not have to go through all the pairs of species in order to figure out morphological diversity each time mutation, speciation or extinction take place. The program, which uses this algorithm, was written and tested. The program was written in the way that allows maximum flexibility for

different simulations. User can easily set such parameters as percentage of the holes on a hypercube, mutation, speciation and extinction rates during the run time of the program. The number of time steps, the order of the events, the number of runs in the simulation is also set during run time by a user.

Using the program described above we studied the behavior of the morphological difference, population size, and the distance of the clade from the founder. It was found that percentage of holes on the hypercube plays the most important role in how fast morphological difference and the distance from the founder of the clade grow. Higher percentages of holes resulted in slower growth rate. It turned out that population size depends only on the difference between the speciation and extinction rates, and the percentage of holes on a hypercube does not have any effect on it.

This program and modeling techniques may be helpful in a future studies of random diffusion on a hypercube with holes and for developing intuition about the processes taking place and the effects of the parameters on the dynamics of morphological diversification and population sizes.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Anstey, R. L., Pashut, J.F. [1995]. "Phylogeny, diversity history, and speciation in Paleozoic bryozoans." *New approaches to speciation and in the fossil record*, Columbia University Press, 239-84

[2] Briggs, D. E. G., Fortey, R. A., Wills M. A. [1992]. "Morphological disparity in the Cambrian", *Science*, vol. 256, 1670-73

[3] Campbell, I. A., Flesselles, J. M., Jullien R., Botet R. [1987]. " Random walks on a hypercube and spin glass relaxation", Journal of Physics, vol. 20, L47-L51

[4] Gavrilets, S. [1997]. " Evolution and speciation on holey adaptive landscapes", *TREE* , vol. 12, no. 8, 307-312

[5] Gould, S. J., Raup D. M., Sepkoski, J. J., Schopf, T. J. M., Simberloff, D. S. [1977]. "The shape of evolution: A comparison of real and random clades", Paleobiology, vol. 3: 23 - 40

[6] Foote, M. [1996]. "Models of Morphological Diversification", *Evolutionary Paleontology*, University of Chicago Press, 63-82

[7] Foote, M. [1994]. "Morphological disparity in Ordovician-Devonian crinoids and the early saturation of morphological space", Paleobiology, vol. 19, 185-204

[8] Foote, M. [1992]. "Paleozoic record of morphological diversity in blastozoan echinoderms", *Proceedings of the National Academy of Science*, vol. 89, 7325-7329

[9] Jepsen, G.L., Mayr, E., Simpson, G.G. [1949]. "Genetics, Paleontology and Evolution", Princeton University Press

[10] Lemke, N., Campbell, I. A. [1996]. "Random walks in a closed space", PHYSICA, vol. A 230, 554-562

[11] Raup, D.M., [1991]. "A kill curve for Phanerozoic marine species", Paleobiology, vol.17, 37 – 48

[12] Slatkin, M. [1981]. " A diffusion model of species selection", Paleobiology, vol.7, 421 - 425

[13] Sprinkle, J. [1980]. "An overview of the fossil record", *Echinoderms: Notes for a short course*, Knoxville: University of Tennessee, 15-26

[14] Wagner, P. J. [1995]. "Testing evolutionary constraint hypotheses with early Paleozoic gastropods", Paleobiology, vol. 21, 248 - 72

[15] Wright, S.[1968]. "Genetic and Biometric Foundations", the University of Chicago Press

APPENDIX

The Program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>
#include <math.h>

/*
//////////////////////////////////////////////////
//function-specific coefficients
*/

/*round-off index*/
#define DEFZ      1000000
/*"magic" function coefficient*/
#define B   1.018556342096842
/*chaotic map index*/
#define N   10
/*percentage of holes on the hypercube*/
#define     DEFCUTOFF   (float)0.5

/*
//////////////////////////////////////////////////
//global coefficients
*/

/*array's dimention*/
#define DEFDIMENSION    100
/*number of mutations in a run*/
#define DEFMUTATIONS    200
/*output steps - every...*/
#define DEFSTEPSOUT           1
/*number of different runs*/
#define DEFRUNS               200
/*birth rate: "create new one if smaller"*/
#define     DEFBIRTHCUTOFF    (float)0.032
/*death death: " delete one if smaller"*/
#define DEFDEATHCUTOFF  (float)0.025
/*mutation rate: "try to make a mutation if smaller"*/
#define DEFMUTATCUTOFF  /*(float)(0.001 * DEFDIMENSION)*/ 1.0
/*trial distance for giant cluster test*/
#define DEFTRIALDISTANCE      (int)((DEFDIMENSION/2) -
(DEFDIMENSION/10))

/*
//////////////////////////////////////////////////
//macros
*/

#define ZeroMemory(p,b) memset(p,0,b)

#define SIZEMUTATIONS (sizeof(int) * (MUTATIONS + 1))
#define SIZEFLOATMUTATIONS (sizeof(float) * (MUTATIONS + 1))
#define SIZECHARPMUTATIONS (sizeof(char*) * (MUTATIONS + 1))
```

```
#define SIZEMUTATIONSARR (nSizeGTypes * SIZECHARPMUTATIONS)
#define SIZEINTDIMENSION (sizeof(int) * DIMENSION)
#define SIZEDBLDIMENSION (sizeof(double) * DIMENSION)
#define SIZECHARDIMENSION (sizeof(char) * DIMENSION)

#define DEFARRSIZE MUTATIONS
#ifndef RAND_MAX
/*max value for rand()*/
#define RAND_MAX 0x7fffffff
#endif
#define BOOL      int
#define TRUE      1
#define FALSE     0

/*
//////////////////////////////////////////////////
*/

typedef enum enReason
{
      REASON_MUTATION,
      REASON_ADDGTYPE,
      REASON_REMGTYPE
} enumReason;

/*
//////////////////////////////////////////////////
//global vars
*/

long DIMENSION = DEFDIMENSION, MUTATIONS = DEFMUTATIONS,
      STEPSOUT = DEFSTEPSOUT, RUNS = DEFRUNS, TRIALSTEPS,
/*minimal acceptable distance after "trial" steps*/
      TRIALDISTANCE = DEFTRIALDISTANCE,
/*for chaotic map*/
      Z;
/*    this flag specifies the order of birth/death*/
BOOL  BIRTHDEATHORDER = TRUE;
float BIRTHCUTOFF = DEFBIRTHCUTOFF, DEATHCUTOFF = DEFDEATHCUTOFF,
            CUTOFF = DEFCUTOFF, MUTATCUTOFF     = DEFMUTATCUTOFF;

/*
//////////////////////////////////////////////////
//function-specific coefficients
*/

double      fCoeff = 0.0, *afPowB = NULL;
long  nBorn = 0, nDied = 0, nMinSpeciesNum;
FILE  *fd = NULL, *fnum = NULL, *fav = NULL;
/*
this array will hold all living and mutating genotypes
arrays of previously used genotypes
preallocate enough room to keep pointer even if we added one genotype
each step
assuming that we can add only 1 genotype per mutation step
```

```
in case we can add more than one this should be changed
+1 is because we add 2 on the initial step
*/
char  **aGenotypes = NULL, **aUsedGenotypes = NULL;
/*
this array will keep distances between mutating genotypes on each step
second array keeps distances to the original genotype
*/
int         *anAvDistancesGTypes = NULL, *anAvDistancesStart = NULL,
/*
this helper array keeps current number of 1s in all genotypes
*/
            *anOnes = NULL,
/*
this array keeps number of active genotypes on each step
*/
            *anNumGTypes = NULL,
/*
this var keeps number of used genotypes on each step (ready to be
reused)
*/
            nNumUsedGTypes = 0, nSizeGTypes,
            *anTotalNumberGTypes = NULL;
/*
this is a helper array to keep the original (starting) genotype for
each run
*/
char  *acStart = NULL;
/*    these arrays will keep grand totals*/
float *afTotalDistGTypes = NULL, *afTotalDistStart = NULL;

/*
//////////////////////////////////////////////////
forward declarations
*/
void ChangeDistances(enumReason eReason, int nMutation, int
nGenotypeInd, int nChangedCell);

/*
//////////////////////////////////////////////////
the function that returns a unique code from binary array
*/

double function(const char* pnS)
{
      double fRes = 0.0;

      double fSum = 0.0;
      int i;
      double fz;

/*    apply the "magic" function*/
      for(i = 0; i < DIMENSION; i++)
      {
            fSum += pnS[i] ? afPowB[i] : 0;
```

```
        }
        fRes = fCoeff * (double)fSum;

/*      apply round-off extractor*/
        fz = fRes * Z;
        fRes = fz - (double)(int)fz;

/*      apply chaotic map*/
        for(i = 0; i < N; i++)
        {
                double fTmp = fRes > 0.5 ? fRes - 0.5 : 0.5 - fRes;
                fRes = 1 - 2 * fTmp;
        }

/*      hopefully, it's random enough by now...*/
        return fRes;
}

/*
//////////////////////////////////////////////////
globals initialization
*/

void InitGlobals()
{
/*      initialize globals*/
        double fPowerB = 1.0;
        int i;

        char buffer[100];

        double T, T1;
/*      figure out our distance for giant cluster test
        L = DIMENSION, p = 1 - CUTOFF, mul = mu*p = p/L
        for dense cluster
        T = -L/(2 * p) * ln(1 - (2 * d)/L)
        for sparse cluster
        T1 = ((2 * d) ^ 2)/p
*/
        T = -(DEFDIMENSION*log(1.0-
(2.0*DEFTRIALDISTANCE)/DEFDIMENSION))/(2.0*(1.0-DEFCUTOFF));
        T1 = (4.0*DEFTRIALDISTANCE*DEFTRIALDISTANCE)/(1.0-DEFCUTOFF);
/*      we have to make no more than TRIALSTEPS to reach TRIALDISTANCE*/
        TRIALSTEPS = (int)((T + T1)/2.0);

/*      ask the user for values other than default*/
        printf("Enter Dimension or keep default %d:",DEFDIMENSION);
        gets(buffer);
        sscanf(buffer,"%d",&DIMENSION);
        printf("Enter Mutations or keep default %d:",DEFMUTATIONS);
        gets(buffer);
        sscanf(buffer,"%d",&MUTATIONS);
        printf("Enter StepsOut or keep default %d:",DEFSTEPSOUT);
        gets(buffer);
        sscanf(buffer,"%d",&STEPSOUT);
```

```
      printf("Enter Runs or keep default %d:",DEFRUNS);
      gets(buffer);
      sscanf(buffer,"%d",&RUNS);
      printf("Enter TrialSteps or keep default %d:",TRIALSTEPS);
      gets(buffer);
      sscanf(buffer,"%d",&TRIALSTEPS);
      printf("Enter TrialDistance or keep default
%d:",DEFTRIALDISTANCE);
      gets(buffer);
      sscanf(buffer,"%d",&TRIALDISTANCE);

      printf("Enter FunctionCutOff or keep default %f:",DEFCUTOFF);
      gets(buffer);
      sscanf(buffer,"%f",&CUTOFF);
      printf("Enter BirthRate or keep default %f:",DEFBIRTHCUTOFF);
      gets(buffer);
      sscanf(buffer,"%f",&BIRTHCUTOFF);
      printf("Enter DeathRate or keep default %f:",DEFDEATHCUTOFF);
      gets(buffer);
      sscanf(buffer,"%f",&DEATHCUTOFF);
      printf("Enter MutationCutOff or keep default
%f:",DEFMUTATCUTOFF);
      gets(buffer);
      sscanf(buffer,"%f",&MUTATCUTOFF);
      printf("Enter 0 to have Death/Birth order or enter to keep
Birth/Death order:");
      gets(buffer);
      sscanf(buffer,"%d",&BIRTHDEATHORDER);

      nSizeGTypes = DEFARRSIZE;
      nMinSpeciesNum = (long)exp((BIRTHCUTOFF - DEATHCUTOFF) *
MUTATIONS);

/*    allocate memory for global arrays*/
      afPowB = (double*)malloc(SIZEDBLDIMENSION);
      aGenotypes = (char**)malloc(SIZEMUTATIONSARR);
      aUsedGenotypes = (char**)malloc(SIZEMUTATIONSARR);
      anAvDistancesGTypes = (int*)malloc(SIZEMUTATIONS);
      anAvDistancesStart = (int*)malloc(SIZEMUTATIONS);
      anOnes = (int*)malloc(SIZEINTDIMENSION);
      anNumGTypes = (int*)malloc(SIZEMUTATIONS);
      acStart = (char*)malloc(SIZECHARDIMENSION);
      afTotalDistGTypes = (float*)malloc(SIZEFLOATMUTATIONS);
      afTotalDistStart = (float*)malloc(SIZEFLOATMUTATIONS);
      anTotalNumberGTypes = (int*)malloc(SIZEMUTATIONS);

      /* Seed the random-number generator with current time so that
      * the numbers will be different every time we run.
      */
      srand( ((unsigned)time( NULL ))<<16 );

/*    initialize weights array once*/
      for (i = 0; i < DIMENSION; i++)
      {
            afPowB[i] = fPowerB;
```

```
            fPowerB *= B;
        }
/*      init the coefficient only once*/
        fCoeff = (double)(B - 1)/(double)(fPowerB * B - 1);
/*      randomly init Z for chaotic map, it varies DEF_Z +- RAND_MAX*/
        Z = DEFZ + (long)(DEFZ * (rand()/(double)RAND_MAX));
/*      make sure there's no overflow*/
        if (Z < 0)
        {
            Z &= 0x7fffff;
        }

        fd = fopen("MutOut.txt","wt");
        fnum = fopen("Number.txt","wt");
        fav = fopen("Average.txt","wt");

/*      NULL all arrays*/
        ZeroMemory(afTotalDistGTypes,SIZEFLOATMUTATIONS);
        ZeroMemory(afTotalDistStart,SIZEFLOATMUTATIONS);
        ZeroMemory(anTotalNumberGTypes,SIZEMUTATIONS);
        ZeroMemory(aGenotypes,SIZEMUTATIONSARR);
        ZeroMemory(aUsedGenotypes,SIZEMUTATIONSARR);
        ZeroMemory(anAvDistancesGTypes,SIZEMUTATIONS);
        ZeroMemory(anAvDistancesStart,SIZEMUTATIONS);
        ZeroMemory(anNumGTypes,SIZEMUTATIONS);
        ZeroMemory(acStart,SIZECHARDIMENSION);
        ZeroMemory(anOnes,SIZEINTDIMENSION);
}

/*
//////////////////////////////////////////////////////
function that checks survival of a particular state
*/

BOOL CheckSurvival(const char* pnS)
{
        double fRes = function(pnS);
        return fRes >= CUTOFF;
}

/*
//////////////////////////////////////////////////////
function that adds a genotype
*/

void AddGenotype(int nFromInd, int nMutation)
{
/*      need to make a copy of the nFromInd's element of the species'
array*/
        char* pAddGenotype = NULL;

/*      if not -1 - change the distances*/
        if (nFromInd != -1)
        {
/*              change distances and helpers*/
```

```
            ChangeDistances(REASON_ADDGTYPE,nMutation,nFromInd,-1);
      }
/*          try array of used ones first*/
      if (nNumUsedGTypes)
      {
/*                success, copy the pointer*/
            pAddGenotype = aUsedGenotypes[nNumUsedGTypes - 1];
/*                decrement the counter*/
            aUsedGenotypes[(nNumUsedGTypes--) - 1] = NULL;
      }
      else
      {
/*                no previously used elements available, create a new
one*/
            pAddGenotype = (char*) malloc(SIZECHARDIMENSION);
      }
/*    start with fresh memory*/
      ZeroMemory(pAddGenotype,SIZECHARDIMENSION);
/*    if not -1 - copy from that genotype*/
      if (nFromInd != -1)
      {
/*          copy the array's element*/

      memcpy(pAddGenotype,aGenotypes[nFromInd],SIZECHARDIMENSION);
      }
/*          add to the end*/
      if (nSizeGTypes <= anNumGTypes[nMutation])
      {
            nSizeGTypes += DEFARRSIZE;
            aGenotypes = (char**)realloc(aGenotypes,SIZEMUTATIONSARR);
            aUsedGenotypes =
(char**)realloc(aUsedGenotypes,SIZEMUTATIONSARR);
      }
      aGenotypes[anNumGTypes[nMutation]] = pAddGenotype;
/*          increment counter ONLY if we added to the end, not inserted
in just freed position*/
      anNumGTypes[nMutation]++;
      nBorn++;
}

/*
//////////////////////////////////////////////////
function that removes a genotype
*/

void RemoveGenotype(int nInd, int nMutation)
{
      char* pRemoveGenotype = aGenotypes[nInd];
      int i;

/*    need to remove the genotype and all distances for it*/
/*    change the distances and other helper arrays*/
      ChangeDistances(REASON_REMGTYPE,nMutation,nInd,-1);
/*    shift remaining gtypes,*/
/*          move it to used array, increment Used's counter*/
```

```
	aUsedGenotypes[nNumUsedGTypes++] = pRemoveGenotype;
	for(i = nInd; i < anNumGTypes[nMutation] - 1; aGenotypes[i] =
aGenotypes[i + 1], i++);
	aGenotypes[anNumGTypes[nMutation] - 1] = NULL;
	anNumGTypes[nMutation]--;
	nDied++;
}

/*
//////////////////////////////////////////////////
this function cleans up global structures at the end of a run
*/

void CleanAfterRun(BOOL bAll)
{
	int i;
/*	reset the born/died counters*/
	nBorn = nDied = 0;
/*	move all genotype arrays, present after last mutations, to Used*/
	for(i = anNumGTypes[MUTATIONS] - 1; i >= 0; i--)
	{
/*		move it to used array if not NULL, increment Used's
counter*/
		aUsedGenotypes[nNumUsedGTypes++] = aGenotypes[i];
	}

/*	NULL all arrays before the next run*/
	ZeroMemory(aGenotypes,SIZEMUTATIONSARR);
	ZeroMemory(anAvDistancesGTypes,SIZEMUTATIONS);
	ZeroMemory(anAvDistancesStart,SIZEMUTATIONS);
	ZeroMemory(anNumGTypes,SIZEMUTATIONS);
	ZeroMemory(acStart,SIZECHARDIMENSION);
	ZeroMemory(anOnes,SIZEINTDIMENSION);
/*	don't NULL number of used gtypes, as we'll use it in the
following runs*/

	if (bAll)
	{
/*		remove all entries from the Used arrays*/
		int i;
		for(i = nNumUsedGTypes - 1; i >= 0; i--)
		{
			free(aUsedGenotypes[i]);
		}
/*		free all dynamically allocated arrays*/
		free(afPowB);
		free(aGenotypes);
		free(aUsedGenotypes);
		free(anAvDistancesGTypes);
		free(anAvDistancesStart);
		free(anOnes);
		free(anNumGTypes);
		free(acStart);
		free(afTotalDistGTypes);
		free(afTotalDistStart);
```

```
            free(anTotalNumberGTypes);

            fclose(fd);
            fclose(fnum);
            fclose(fav);
      }
}

/*
//////////////////////////////////////////////////
this function makes a single mutation step
*/

int MakeMutationStep(char* pcGenotype, BOOL bRandMut)
{
      int rplace = -1;
/*    check whether we are going to try to make a mutation or not*/
      if (!bRandMut || (rand() / (float)RAND_MAX) <= MUTATCUTOFF)
      {
            rplace = (int)((rand()/(double)RAND_MAX) * (DIMENSION -
1));
/*           randomly choose the place in the array
            flip the number
*/
            pcGenotype[rplace] ^= 1;
/*          check that it survives*/
            if (!CheckSurvival(pcGenotype))
            {
/*                didn't survive, rollback*/
                  pcGenotype[rplace] ^= 1;
                  rplace = -1;
            }
      }

      return rplace;
}

/*
//////////////////////////////////////////////////
this function fills first genotype before first run
and tests it for being in a cluster
*/

void InitializeGenotype()
{
      char* pcGenotype = aGenotypes[0];
      int nCurrDistance = 0, nCount = 0;
      int i;

/*    run until we reach required distance within TRIALSTEPs*/
      while(nCurrDistance < TRIALDISTANCE)
      {
            int i = 0;

            nCount++;
```

```
            if ((nCount % 10) == 0)
            {
/*                  having problems, try to reinitialize the sequence*/
                    srand( (unsigned)time( NULL ) );
            }

            nCurrDistance = 0;
/*          initially fill the genotype array*/
            while(!i || !CheckSurvival(pcGenotype))
            {
                    for (i = 0; i < DIMENSION; i++)
                    {
/*                          rand() on UNIX has a bug - each successive call
alternates lowest bit
                            Have to use a middle bit instead
*/
                            char nTemp = (rand()>>10) & 1;
                            pcGenotype[i] = nTemp;
                    }
/*                  it randomly fills the arrays with 0 and 1*/
            }
/*          make a copy of the original genotype*/
            memcpy(acStart,pcGenotype,SIZECHARDIMENSION);
/*          now, make TRIAL steps and see where we are*/
            for(i = 0; i < TRIALSTEPS && nCurrDistance < TRIALDISTANCE;
i++)
            {
/*                  make regular mutations, don't use random generator to
restrict mutations*/
                    int nChangedCell =
MakeMutationStep(pcGenotype,FALSE);
                    if (nChangedCell != - 1)
                    {
                            nCurrDistance += pcGenotype[nChangedCell] ==
acStart[nChangedCell] ? -1 : 1;
                    }
            }
      }
/*    we've found a genotype on the cluster,
      now set it to the Original to start our mutations
*/
      memcpy(pcGenotype,acStart,SIZECHARDIMENSION);
/*    prepare the array of 1s*/
      for(i = 0; i < DIMENSION; i++)
      {
            anOnes[i] = (int)acStart[i];
      }
}

/*
//////////////////////////////////////////////////
this function changes distances and other service arrays
doesn't change the genotypes array
*/
```

```
void ChangeDistances(enumReason eReason, int nMutation, int
nGenotypeInd, int nChangedCell)
{
/*    this function should be called BEFORE the change happened
      in cases for adding/removing
      it's more convenient for us to have function for mutations steps
      without a call to changing distances in the function's body (we
don't need it
      if we're checking positioning on the cluster), so the call to
ChangeDistances happens
      AFTER the change in case of mutations (which means that we need
to use inverted value for the mutated cell)
*/
      int nDelta = 0, nDeltaStart = 0;
      const char* pGType = aGenotypes[nGenotypeInd];
      switch(eReason)
      {
            case REASON_MUTATION:
/*                a mutation has happened
                  should have the mutated cell passed
*/
                  nDelta = ((anOnes[nChangedCell]<<1) -
anNumGTypes[nMutation]) *
/*                      use inverted value for the mutated cell - need
to use old value, before mutation*/
                        (((pGType[nChangedCell]^1)<<1) - 1) - 1;
/*                distance to Start always changes by 1, either +1 or -
1*/
                  nDeltaStart = pGType[nChangedCell] !=
acStart[nChangedCell] ?
                        1 : -1;
                  anOnes[nChangedCell] += pGType[nChangedCell] != 0 ? 1
: -1;
                  break;
            case REASON_ADDGTYPE:
/*                compute new distance for this mutation according to
the formula for mutation
                  use anAvDistancesGTypes, anAvDistancesStart, anOnes
*/
                  {
                        int i;
                        for(i = 0; i < DIMENSION; i++)
                        {
                              nDelta += pGType[i] *
                                    (anNumGTypes[nMutation] -
(anOnes[i]<<1)) + anOnes[i];
                              nDeltaStart += pGType[i] != acStart[i] ?
                                    1 : 0;
/*                            now change the array of 1s*/
                              anOnes[i] += pGType[i] != 0 ? 1 : 0;
                        }
                  }
                  break;
            case REASON_REMGTYPE:
/*                a genotype has been removed
```

```
                compute new distance for this mutation according to
the formula for genotypes remove
                use anAvDistancesGTypes, anAvDistancesStart, anOnes
*/
                {
                        int i;
                        for(i = 0; i < DIMENSION; i++)
                        {
                                nDelta += pGType[i] *
                                        (anNumGTypes[nMutation] -
(anOnes[i]<<1)) + anOnes[i];
                                nDeltaStart += pGType[i] != acStart[i] ?
                                        1 : 0;
/*                              now change the array of 1s*/
                                anOnes[i] -= pGType[i] != 0 ? 1 : 0;
                        }
                        nDelta = -nDelta;
                        nDeltaStart = -nDeltaStart;
                }
                break;
        }
        anAvDistancesStart[nMutation] += nDeltaStart;
        anAvDistancesGTypes[nMutation] += nDelta;
}

/*
//////////////////////////////////////////////////
tries to add and remove genotype(s)
*/

void TryAddRemoveGenotype(int nMutation)
{
/*      if we added and immediately removed the same genotype, do nothing
        if we added a genotype and removed a different one, insert new
one instead of deleted
        if only added or removed - do full cycle
*/
/*      don't add or remove if we have 0 gtypes currently*/
        int i, nNumGTypes = anNumGTypes[nMutation];
        for(i = 0; i < nNumGTypes;)
        {
/*              depending on the flag we have either Death/Birth order, or
Birth/Death order*/
                if (BIRTHDEATHORDER)
                {
                        if ((rand() / (double)RAND_MAX) < BIRTHCUTOFF)
                        {
/*                              add gtype*/
                                AddGenotype(i,nMutation);
                        }
                        if ((rand() / (double)RAND_MAX) < DEATHCUTOFF)
                        {
/*                              remove the genotype
                                don't advance to the next genotype because we
just shifted all remaining gtypes to the left
```

```
*/
                        RemoveGenotype(i,nMutation);
                        nNumGTypes--;
                  }
                  else
                  {
/*                      advance to the next genotype*/
                        i++;
                  }
            }
            else
            {
                  if ((rand() / (double)RAND_MAX) < DEATHCUTOFF)
                  {
/*                      remove the genotype
                        don't advance to the next genotype because we
just shifted all remaining gtypes to the left
*/
                        RemoveGenotype(i,nMutation);
                        nNumGTypes--;
                  }
                  else
                  {
                        if ((rand() / (double)RAND_MAX) < BIRTHCUTOFF)
                        {
/*                            add gtype*/
                              AddGenotype(i,nMutation);
                        }
/*                      advance to the next genotype*/
                        i++;
                  }
            }
      }
}

/*
////////////////////////////////////////////////////
function to initialize necessary variables for next mutation step
*/

void PrepareForNextStep(int nMutation)
{
      if (nMutation < MUTATIONS)
      {
            anAvDistancesGTypes[nMutation + 1] =
anAvDistancesGTypes[nMutation];
            anAvDistancesStart[nMutation + 1] =
anAvDistancesStart[nMutation];
            anNumGTypes[nMutation + 1] = anNumGTypes[nMutation];
      }
}


/*
////////////////////////////////////////////////////
```

```
main module
*/

void main()
{
      char buff[250];
      int i;
      int nRun, nTotalRunsNumber = 0;
/*    initialize all global arrays and coefficients*/
      InitGlobals();

      printf("Please, wait\n");

      for(nRun = 0; nRun < RUNS; nRun++)
      {
            int i;
/*          show that we are performing a run...*/
            sprintf(buff,"Run: %d\n",nRun);
            printf(buff);
/*          create genotype array initially, will add new as needed*/
            AddGenotype(-1,0);
/*          fill the first one*/

            InitializeGenotype();
/*          copy the filled array into the second slot*/

            AddGenotype(0,0);
/*          prepare all arrays for the run*/
            PrepareForNextStep(0);

            for (i = 1; i <= MUTATIONS && anNumGTypes[i] > 0; i++)
            {
                  int nGenotypeInd;
/*                try to mutate all genotypes in turn*/
                  for(nGenotypeInd = 0; nGenotypeInd < anNumGTypes[i];
nGenotypeInd++)
                  {
                        int nChangedCell =
MakeMutationStep(aGenotypes[nGenotypeInd],TRUE);
/*                      check whether our mutation was successful*/
                        if (nChangedCell != -1)
                        {

      ChangeDistances(REASON_MUTATION,i,nGenotypeInd,nChangedCell);
                        }
                  }
                  TryAddRemoveGenotype(i);
                  PrepareForNextStep(i);
            }

/*          end of mutations loop*/
/*          we're not interested in runs with too few genotypes end*/
/*          depending on our Birth/Death order we calculate totals
differently*/
```

```
            if (BIRTHDEATHORDER ? anAvDistancesGTypes[MUTATIONS] > 0 :
anNumGTypes[MUTATIONS] >= nMinSpeciesNum)
            {
                  nTotalRunsNumber++;

                  for (i = 0; i <= MUTATIONS; i++)
                  {
                        int nNumDist;
      /*                average values for distances
                        number of distances between each 2 gtypes
                        is computed as number of combinations C(n,2) =
n!/(2! * (n - 2)!) = (n * (n - 1)) / 2
      */
                        nNumDist = 1;
                        if (anNumGTypes[i] > 2)
                        {
                              nNumDist = (anNumGTypes[i] *
(anNumGTypes[i] - 1)) / 2;
                        }
      /*                average distances among genotypes in the
system*/
                        afTotalDistGTypes[i] += anAvDistancesGTypes[i]
/ (float)nNumDist;
      /*                average distances to the initial genotype*/
                        afTotalDistStart[i] += anNumGTypes[i] != 0 ?
anAvDistancesStart[i] / (float)anNumGTypes[i] : (float)0.0;
      /*                number of genotypes*/
                        anTotalNumberGTypes[i] += anNumGTypes[i];
                  }
            }
/*          perform all necessary clean-up after run*/
            CleanAfterRun(FALSE);
/*          change global coefficient z*/
            Z += DEFZ;
      }
/*    end of runs*/
/*    output to files*/

      nTotalRunsNumber = nTotalRunsNumber != 0 ? nTotalRunsNumber : 1;
      for (i = 0; i <= MUTATIONS; i+= STEPSOUT)
      {
/*          output to the file average distance among gtypes over all
runs*/
            sprintf(buff,"%f\t",
(afTotalDistGTypes[i]/nTotalRunsNumber));
            fwrite(buff,sizeof(char),strlen(buff),fd);
/*          output to the file average distances to the initial
genotype over all runs*/
            sprintf(buff,"%f\t",
(afTotalDistStart[i]/nTotalRunsNumber));
            fwrite(buff,sizeof(char),strlen(buff),fav);
/*          output to the file number of genotypes over all runs*/
            sprintf(buff,"%f\t",
(anTotalNumberGTypes[i]/(float)nTotalRunsNumber));
            fwrite(buff,sizeof(char),strlen(buff),fnum);
```

```
        }
/*      end of output*/
        fwrite("\n", sizeof(char), 1,fd);
        fwrite("\n", sizeof(char), 1,fav);
        fwrite("\n", sizeof(char), 1,fnum);
/*      empty line*/
/*      perform all necessary clean-up after runs*/
        CleanAfterRun(TRUE);
}
```

VITA

Liliya Goldshmid was born in Saint-Petersburg (back then - Leningrad), Russia, January 5, 1973. She graduated from the high school in the summer of 1990, and later that year her family moved to the United States. In the fall of 1991 she entered the Belmont University in Nashville, Tennessee. In the Spring of 1995 she received a Bachelor of Science degree with double major in Mathematics and Computer Science. She joined the graduate program in the University of Tennessee in the Fall of 1995 as a graduate assistant. She worked with Dr. Gavrilets, Dr. Vose and Dr. Rosinski on modeling speciation, extinction and morphological evolution on holey hypercube and graduated with a Master of Science degree in August of 1998.