A Thesis

entitled

Genome Evolution Model (GEM): Design and Application

by

Andrew McSweeny

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the

Master of Science Degree in Biomedical Sciences

_____

Dr. Alexei Fedorov, Committee Chair

_____

Dr. Robert Blumenthal, Committee Member

_____

Dr. Sadik Khuder, Committee Member

_____

Dr. Robert Trumbly, Committee Member

_____

Dr. Patricia R. Komuniecki, Dean
College of Graduate Studies

The University of Toledo

December 2010

Genome Evolution Model (GEM): Design and Application

Andrew McSweeny

University of Toledo College of Medicine

2010

Acknowledgements

I am grateful for the assistance and support of my colleagues, friends and family. Principal investigator Alexei Fedorov, PhD helped conceive and guide this project. Sam Shepard, a fellow computer scientist, gave me practical advice on bioinformatics research from a computer scientist's perspective. Ashwin Prakash, MD provided data on the location of functional regions within the DNA sequences used in GEM, and is a very wise yet understated and humble person. Dave Rearick was a source of interesting conversation on topics inside and outside the realm of science. Maryam Nabiyouni humored me, when necessary, and provided crossover and substitution data for this project. While many of my personal acquaintances find my work boring, my longtime friend Jared Rosenberg deserves credit for his interest in my project and help with graphic design. My brother, Patrick, built the foundation for a web interface to GEM, and has been another source of wisdom. My father, John McSweeny PhD, JD, has given me many pearls of wisdom about conducting research and the surviving my time served in the academic world. My mother, Marilee McSweeny, MA, MS, taught me to believe in my clarity of thought and writing ability, despite biting criticisms from my high school English teachers (she has more credentials in the study of English anyways). My uncle Jim McSweeny, MS, knows what a genome is when my extended family meets during holidays; his wife Patrice has also been very supportive of my work. I am grateful for support from my aunt Emily Erickson, MA, and my grandfather Austin McSweeny, MD. Note that my family members holding graduate degrees seem to be the ones who understand the difficulty of this undertaking! Finally, I would like to acknowledge my dwarf rabbit, Harry, for his tremendous moral support.

Table of Contents

Acknowledgements..... iii

Introduction 1

Overview of the GEM Algorithm..... 7

The GEM Algorithm in Detail 11

GEM v0.45 Example 31

Summary 36

References 39

Abstract 43

Appendix A: Structure of the GEM Java program 44

Appendix B: Source code for GEM v0.45..... 56

Introduction

Evolutionary Computing

Evolutionary computing is the process of designing algorithms that emulate the process of evolution and natural selection; these algorithms are used to search for optimal solutions to a problem using an evolutionary process. The earliest genetic algorithms (GA) were created in the 1960's and 70's. These algorithms are intended to leverage the process of evolution and natural selection to find optimal solutions to real-world problems. Genetic algorithms work by creating a population of virtual individuals, with each individual containing one or more chromosomes representing a potential solution to a problem (Goldberg 1989; Banzhaf 1998).

Genetic algorithms can be applied to any problem in which potential solutions can be represented as a string of characters. A very simple application would be to create a genetic algorithm to take a random string of 12 characters and perform a process of simulated natural selection until the string read "Hello World!" In this case, each organism has a genome of 12 characters from the ASCII character set. Trying random combinations of characters from the 128 possible in the ASCII character set would require searching among $\binom{128}{12} = 2.4 \times 10^{16}$ possible combinations of characters to find the correct solution.

However, if we define a fitness function for this problem, a genetic algorithm can be applied to this problem. For this problem, each organism's fitness is assessed by comparing the contents of the 12 character genome to the correct solution. A fitness

function for the aforementioned problem could be defined as “fitness=number of correct characters.”

The pseudocode for a typical genetic algorithm is outlined as follows by Merrit (2008).

```
create initial population
calculate each individual’s fitness
calculate average population fitness
repeat
     select highest-fitness individuals to reproduce
     select mating pairs at random
     apply crossover operator
     apply mutation operator
     calculate each individual’s fitness
     calculate average population fitness
until terminating condition
```

Once the he terminating condition for a genetic algorithm is met, execution of the algorithm stops. For example, the condition may resulting in the algorithm ending execution after a certain number of generations, or once the fitness of an individual in the population reaches a certain threshold (12 in the “Hello World!” example).

Application of a genetic algorithm to the problem of generating the string “Hello World!” would begin with a population of organisms with random 12-character sequences like “!@DFe8$vt~*” (fitness = 0). However, as the algorithm proceeds with mating, mutation and crossover, and calculation of fitness, the lowest scorings organisms are removed. Organisms with marginally better fitness survive and reproduce; genomes within the population begin to resemble the ideal solution (e.g. “H@lle Wvt~*!”; fitness = 5).

Using a genetic algorithm will produce the “Hello World!” solution (an individual with a fitness score of 12) much more quickly than trying all permutations of 12 ASCII

characters. The general idea of a genetic algorithm, therefore, is that representing solutions to a problem as "genomes" and applying an evolution-like process to these genomes produces an optimal solution more quickly than trying random solutions.

Artificial Life

Closely related to the field of evolutionary computation is the field of artificial life. Scientists in the field of artificial life also develop algorithms that mimic phenomena in the natural world. However, the goal of artificial life is not always pragmatic. Artificial life is not generally applied to optimization problems, as is evolutionary computing. Artificial life is used to model how small components can interact in a larger system when each component (i.e. cell, organism, flock) has a set of rules for how it interacts with other components and the environment (Wolfram 2002).

In a sense, a genetic algorithm meeting certain criteria can be considered an example of artificial life. Such a GA a) incorporates real natural phenomena into its modeling of mutation, crossover, mating, and selection b) uses real DNA sequence data for its genome c) has a fitness function that evaluates mutations in the DNA sequence according to the effect they would have in real life and d) is used to examine the processes governing the complex dynamics of components in a lifelike process, rather than for application to optimization problems.

Therefore, in this study, the idea of the genetic algorithm is applied to real DNA sequences from *H. sapiens*. The fitness function is derived from current understanding of real biological phenomena. For example, mutations within coding sequences generally

have a greater impact on an organism's fitness than mutations within noncoding DNA. Like *H. sapiens* and other eukaryotes, the artificial organisms have diploid chromosomes, so fitness is calculated using rules modeling Mendelian dominance.

The development of genetic algorithms and artificial life preceded the availability of large sequences of DNA by decades. Additionally, most variants of artificial life and evolutionary algorithms were conceived when understanding of functional DNA sequences and the effects of specific mutations were poorly understood not only by computer scientists and mathematicians, but biologists as well. This may explain why, as far as I am aware, no one has ever attempted such an ambitious model of artificial life.

Of course, there are several other models of DNA evolution developed using continuous-time Markov chains. Examples include the JC69 model (Jukes and Cantor 1969), the K80 model (Kimura 1980), the F81 model (Felsenstein 1981), the HKY85 model (Hasegawa, Kishino et al. 1985) model, the T92 model (Tamura 1992), the TN93 model (Tamura and Nei 1993), and the GTR (generalized time reversible) model (Tavaré 1986). These models of DNA evolution are primarily used to estimate divergence times in phylogenetics, however, and do not closely model the process of sexual reproduction or account for differences between the many types of functional DNA.

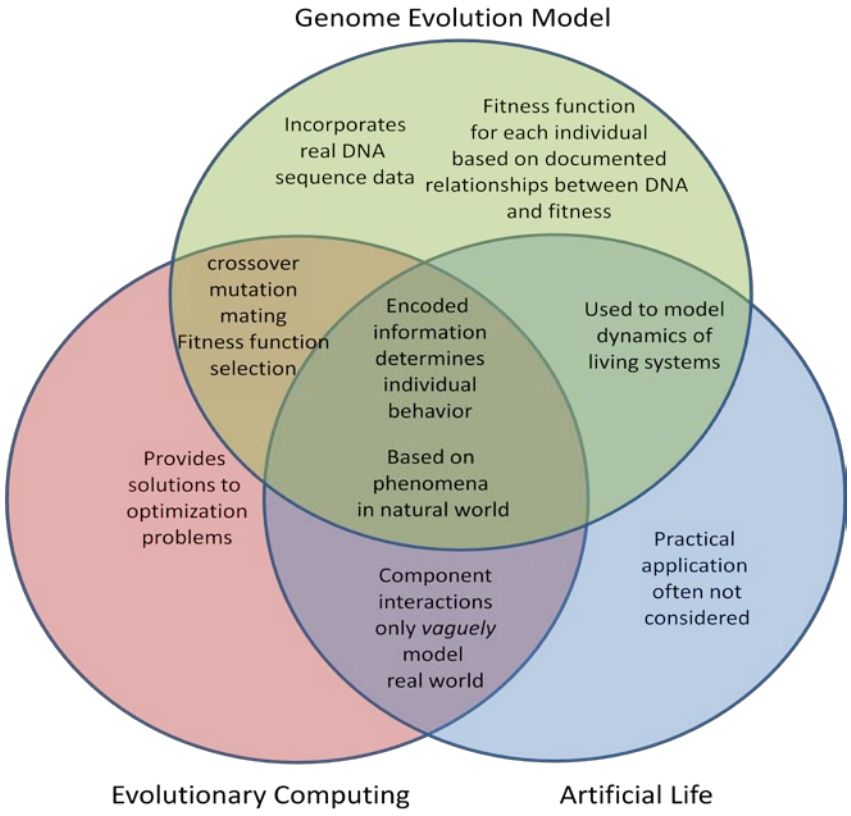
Genome Evolution Model

The unique variant of a genetic algorithm, using real DNA sequence data and a fitness function modeling real biological phenomena is termed Genome Evolution Model (GEM). The purpose of GEM is twofold. First, GEM allows observation of DNA sequence changes over the course of generations, as a result of the constraints of the fitness function. Second, GEM models the interplay of factors that affect population fitness, such as fertility (the number of offspring per mating pair) and the frequency of beneficial/deleterious mutations.

It is important to emphasize that the development of a Genome Evolution Model is in the very earliest stages. At this point, it is premature to speculate on all of the possible features and applications this model can have. The central argument of this thesis does not revolve around the immediate applications of GEM to solve problems in biology. The work of population geneticists has already provided solutions to many of the problems GEM could be immediately applied to.

The crux of this thesis is that a very basic model of DNA sequence evolution can be created by building on the foundations of evolutionary computing and artificial life, incorporating real data to make the model resemble what occurs in nature. This thesis is not a report about a series of experiments and their results; it is a declaration that a limited, yet comparatively advanced model of DNA sequence evolution can be built, and a description of how this is done. Figure 1 shows the similarities and differences between evolutionary computing, artificial life, and GEM.

Figure 1: A comparison of evolutionary computing, artificial life, and GEM

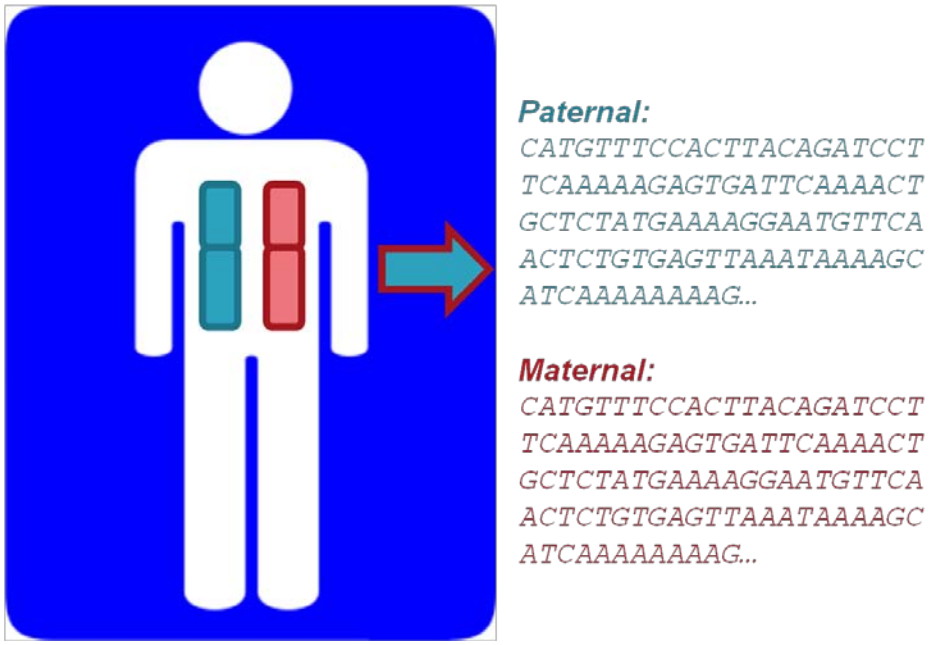


Overview of the GEM Algorithm

The GEM algorithm begins with a population of genetically identical "virtual individuals" (Figure 2). Each virtual individual has one pair of chromosomes; each chromosome is represented as a sequence of the nucleotide bases A, C, T and G. The initial population of virtual individuals is genetically identical; all individuals have the same sequence for both chromosome copies, and have fitness of 0.0.

Figure 2: GEM operates on a population of "virtual individuals"

GEM performs a simulation of DNA sequence evolution on a population of virtual individuals. Each virtual individual has a single pair of chromosomes, represented as a string of the characters A, C, T and G.



Each generation, base substitutions (mutations) are applied to the genomes of the population of artificial organisms. These substitutions can be thought of as germline mutations—they are passed on to offspring through gametes, and affect the fitness only of the offspring. Each of these base substitutions carries a measure of fitness effect in the offspring, *s* (selective advantage, selection coefficient).

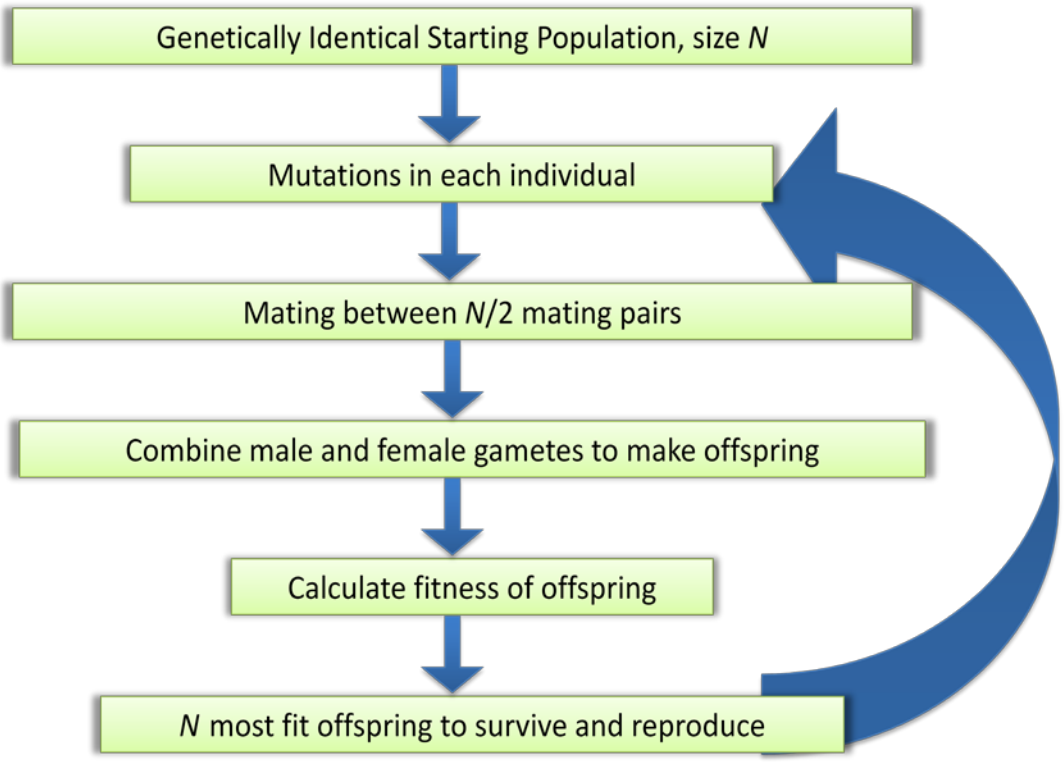
Each mutation has *s* that is positive, neutral, or negative; positive values of *s* are associated with fitness improvements in offspring, negative values with fitness decline. The overall fitness of each offspring is determined by the *s* of the mutations it inherits from its parents. The sum of all *s* values in an organism gives a rough gauge of fitness. Σs is positively correlated with fitness, however, since each organism is diploid, the total fitness is calculated according to a model of Mendelian inheritance. Therefore, some values of *s* are masked when calculating fitness. For example, an organism may have very negative value of *s* for a mutation within a certain locus of the maternal chromosome, yet this deleterious mutation may be masked by the paternal allele due to dominance.

From the initial population of *N* virtual individuals, *N/2* mating pairs are formed. For each mating pair, new virtual individuals are created by combining gametes from the male and the female in the mating pair; gametes are generated in each parent by crossing over of the respective parents' maternal and paternal DNA sequences. The population offspring become the new population; the parents are eliminated. The fitness of each offspring is calculated according to the locations and *s* values of the mutations it inherits as well as the type of inheritance (dominant, co-dominant, etc) where the mutations reside.

If the mean number of offspring per mating pair, λ_o, exceeds two, then the population size has the potential to grow exponentially each generation. As in most genetic algorithms, GEM keeps the population size constant (*N* individuals) in each generation. This is done by allowing only the *N* fittest individuals of each generation to reproduce. Thus, the remaining infertile individuals are removed from the population. Of the *N* fittest individuals allowed to reproduce, the average population fitness is calculated. This completes one generation of reproduction in GEM. The process of sexual reproduction repeats the number of generations specified by the user.

Figure 3: Overview of the GEM algorithm

GEM begins with an identical population of size N. Germline substitutions occur in each individual which are passed onto offspring who inherit these mutations. According to the mutations inherited, fitness is calculated for each offspring. The N fittest offspring become the next generation and the process repeats.



About *s* values

The selection coefficient, *s*, is the same used in the study of genetic drift, also called "selective advantage." For example, Ohta (1992) used this interpretation of *s* to define a function for the fixation probability of a mutant genotype as the product of population size and selection coefficient ($N \cdot s$). According to Ohta's model, genotypes with negative *s* would have lower probability of fixation than genotypes with positive *s*. *s* is therefore a relative measure of fitness for a genotype; selection coefficients are positive, zero, or negative real numbers. A positive selection coefficient indicates a favored genotype; negative means it is selected against; zero means it is neutral (Ridley 2004).

The *s* value used in GEM should not be confused with the traditional selection coefficient used in most population genetics, expressed as a value from 0.0 to 1.0. The more traditional calculation of *s* for a genotype is equal to $1 - W$, where *W* is the adaptive value of that genotype. *W* is the relative probability that a genotype will reproduce compared to a favored genotype. For example, an organism that has a genotype with *s* of 0.1 (W=0.9) will be produce 90 fertile progeny for every 100 produced by a favored genotype (ISCID Encyclopedia of Science and Philosophy 2010) .

The GEM algorithm in detail

GEM Initialization: Specification of initial conditions

First, the parameters for the simulation are initialized according to user specifications. The number of individuals in the simulation is set; the population size remains constant from one generation to the next. Other basic parameters include: number of generations, number of base substitutions per individual per generation, mean number of offspring per mating pair.

The algorithm requires a contiguous sequence of DNA that will represent a single chromosome in the genome for the initial population (individuals in the initial population are genetically identical). Contiguous DNA sequences (contigs) can be downloaded from the UCSC genome browser (Rhead, Karolchik et al. 2010) or the NCBI ftp site (Tatusova 2010) . For example, the first few lines of NT_011512, a 28.6 Mbp contig from Human chromosome 21, are shown Figure 4. The contig used for the simulation is referred to as the "GEM chromosome sequence contig" throughout this text.

Figure 4: GEM requires a contiguous DNA sequence (contig)

Before the GEM algorithm is executed, a contiguous DNA sequence must be specified. This will be used in all individuals in the starting population; they will be genetically identical initially. The sequence can be downloaded from the NCBI ftp site or from the UCSC genome browser. Shown are the first few lines from NT_011512, a 28.6 Mbp contig from Human chromosome 21.

```
>gi|51475294|ref|NT_011512|Hs21_11669 Homo sapiens chromosome 21
 genomic contig, reference assembly
 CATGTTTCCACTTACAGATCCTTCAAAAAGAGTGTTTCAAAACTGCTCTATGAAAAGGAATGTTCAACTC
 TGTGAGTTAAATAAAAGCATCAAAAAAAAGTTTCTGAGAATGCTTCTGTCTAGTTTTTATGTGAAGATAT
 TTCCATTTTCTCTATAAGCCTCAAAGCTGTCCAAATGTCCACTTGCAGATACTACAAAAAGAGTGTTTCA
 …
```

A bp/cM genetic map is specified for the GEM chromosome sequence contig, used for modeling crossover. This data is available from the international HapMap consortium (Frazer, Ballinger et al. 2007). An optional constant may be provided to increase or decrease the rate of crossover; the genetic distance at each point in the bp/cM map is multiplied by this constant. The HapMap bp/cM maps for each *H.sapiens* chromosome are available at http://hapmap.ncbi.nlm.nih.gov/downloads/recombination. At the time of this writing, the latest recombination rates files are for build 36 of the Human genome (hg18). A portion of the recombination rates table for *H.sapiens* chromosome 21 is shown in Table 1. The position column lists the position in bp of each genetic marker in the map, in the corresponding assembled human chromosome.

The assembled human chromosomes for hg18 may be downloaded from http://hgdownload.cse.ucsc.edu/goldenPath/hg18/chromosomes/. To create a genetic map for the GEM chromosome sequence contig, it is necessary to align the contig with

the assembled chromosome the contig is found on, and determine the position the contig occupies on the assembled chromosome.

Table 1: GEM uses a bp/cM Genetic Map

Along with a contiguous sequence of characters representing a DNA sequence (contig), GEM requires a genetic map for this sequence to be used for modeling crossover. Such maps are available from the International HapMap Consortium, although the absolute position in the chromosome must be translated to the position within the contig under investigation. Shown are the first few entries in the map file "genetic_map_chr21_b36.txt," followed by ellipses, and then the corresponding entries related to NT_011512, which has absolute position of 13,260,001-41,877,429bp within Hs21b36.

Position	COMBINED_rate(cM/Mb)	Genetic_Map(cM)
9887804	0.6247366105	0
9928594	0.6231151531	0.0254830063
9928786	0.4976528486	0.0256026445
…	…	…
13282761	0.0010000000	0.6101617220
13284914	1.2236529551	0.6101638750
13299297	1.4718452975	0.6277636754

Alignment of the NT_011512 contig with the hg18 chromosome 21 assembly, using the UCSC genome browser, indicates that NT_011512 begins at 13,260,001 bp in the hg18 chromosome 21 assembly. A map specific for this contig can be created by using spreadsheet software to recalculate the values in the "position" and "Genetic_Map" columns (Table 2).

Table 2: Genetic map for Hs21 translated into a genetic map for NT_011512

To make the HapMap genetic_map_chr21_b36.txt usable by GEM for contig NT_011512, spreadsheet software like Microsoft Excel can be used to translate positions in the chr21 assembly to positions in NT_011512. Note that the COMBINED_rate (cM/Mb) column is not needed and is therefore removed. Shown are the first few lines of the translated map. Position 1 in NT_011512 corresponds to position 13,260,001 in b36 if human chromosome 21. NT_011512 comprises 52.1cM of the total size of Hs21 (62.7cM).

position_NT_011512	Genetic_Map(cM)
1	0.000000000
17239	0.000017238
22761	0.000022760
…	…
28603059	52.145881390
28603213	52.146134203
28603290	52.146332523

For the GEM chromosome sequence contig, the positions of all protein-coding sequences are specified. These are provided in a file which gives the beginning and end of each protein coding DNA sequence in the contig. Additionally, the user can also specify the coordinates of other functional elements in the genome that will be taken into account when calculating *s* for a mutation.

One example of such functional elements are the GC-rich and GC-poor regions of mid-range inhomogeneity (MRI), as described by Prakash et al. (2009). One method to account for MRI regions when calculating *s* is to use a constant, added to the value of *s* during calculation. This constant will be added when calculating *s* for a base substitution to from {A,T} to {G,C} within a GC-rich region and subtracted for a base substitution from {G,C} to {A,T}. Conversely, this constant will be subtracted when calculating *s* for a base substitution to from {A,T} to {G,C} within a GC-poor region and added for a base

substitution from {G,C} to {A,T}. This functionality for MRI regions is built into the current specification of GEM. Future versions will allow the user to specify other regions of interest and rules for mutations within these regions.

Later in this text, I will describe how GEM breaks the chromosome sequence contig into coding and noncoding loci. Briefly, each region of CDS exons is separated into a coding locus and regions outside coding sequences become noncoding loci. The fitness, F, for noncoding loci are calculated in GEM by averaging all s values in both the maternal and paternal alleles. However, in order to model dominance, the F of coding loci are calculated according to Equation 1. This *ad hoc* equation is designed to provide a primitive model of dominance for each locus. For instance, for a locus with a=1, b=0, and c=0, the fitness of the locus will be the maximum of the fitness between the maternal and paternal alleles, providing a simple model of dominance.

Equation 1:

$$F_{coding} = a \cdot \max(\Sigma s_m, \Sigma s_p) + b \cdot \min(\Sigma s_m, \Sigma s_p) + c \cdot \text{mean}(\Sigma s_m, \Sigma s_p)$$

where Σs_m and Σs_p are the sum of all s for the maternal and paternal alleles

Therefore, a probability distribution for assigning values to coefficients a, b, and c is necessary. A single probability distribution specifies the likelihood of a coding locus having one of multiple combinations of values for a, b and c. Table 3 shows an example of such a discrete probability distribution, where 87.5% of coding loci will be completely dominant in that the fitness of the heterozygote with one dominant allele is equal to the

fitness of a homozygote with two dominant alleles (Ye, Yang et al. 2003). This pattern is in accordance with a recent estimate from our lab that one functional allele can substitute for two functional copies in approximately seven eighths (87%) of genes (Rearick, Prakash et al. 2010).

Table 3: A probability distribution for dominance coefficients

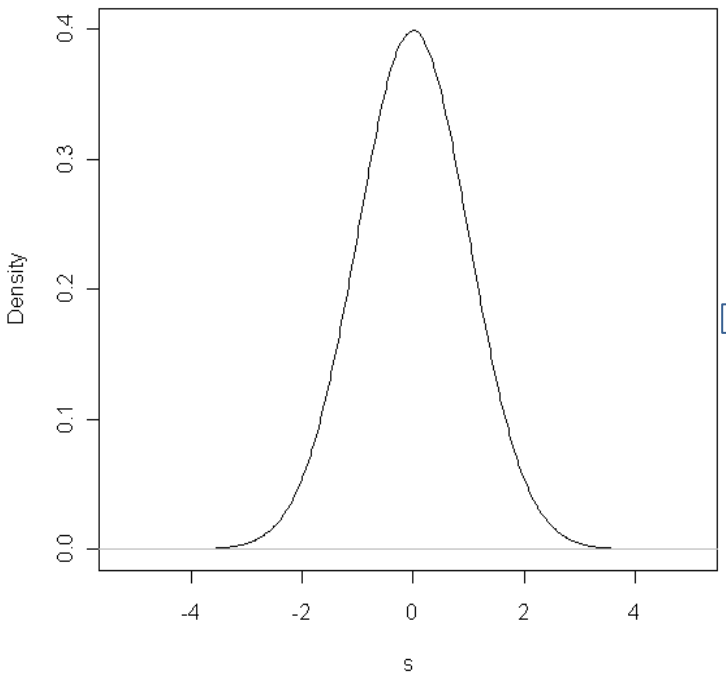
The overall fitness of each coding locus is determined according to an equation modeling dominance, with three coefficients. Each coding locus is assigned a set of values for *a*, *b*, and *c* according to a probability distribution. Shown is an example of such a probability distribution, where seven eighths (87%) of coding loci have complete dominance (i.e. fitness is equal to the maximum between maternal and paternal alleles) and one eighth (13%) of coding loci have negative dominance (i.e. fitness is equal to the minimum between maternal and paternal alleles). This distribution can be modified to consider co-dominance, by adding an entry with *a*=0, *b*=0, *c*=1.

Probability	***a* (max coefficient)**	***b* (min coefficient)**	***c* (mean coefficient)**
0.875	1.0	0.0	0.0
0.125	0.0	1.0	0.0

Two probability distributions of *s* for coding (translated) and non-coding bases are also necessary. As mentioned earlier, the chromosome sequence contig is divided into coding and non-coding regions; however the coding regions will contain non-coding intronic DNA. The *s* for mutations within these intronic sequences will be taken from the distribution of *s* for non-coding DNA, even though these sequences lie within coding loci. Probability distributions can be loaded into GEM from user-defined discrete probability distributions supplied in file. Figure 5 shows the standard normal distribution specified as a table of 20 quantiles.

Figure 5: Probability distributions for *s* are specified in tables

During initialization, GEM requires probability distributions *s*. Below is an example of a standard normal distribution (μ=0, σ=1) specified as a table of discrete values, by dividing the distribution in 20 quantiles. Values in the second column specify the probability of the *s* value in the first column.



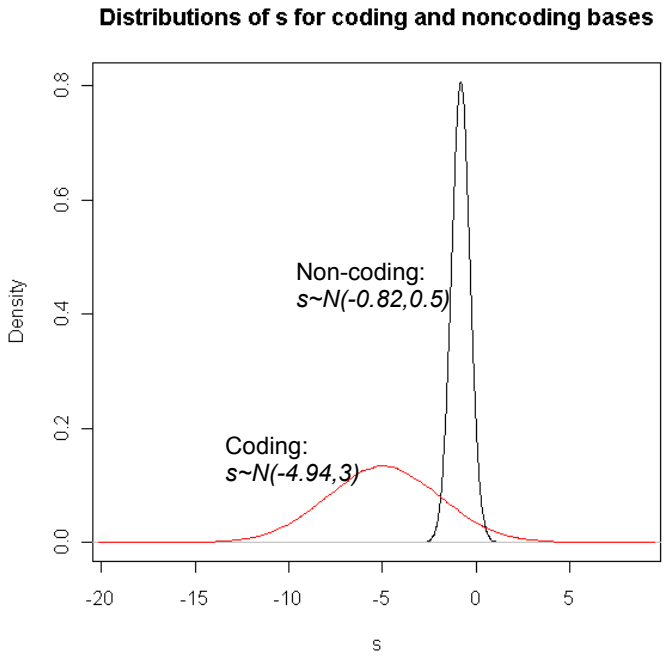
s	Probability
-2.06	0.05
-1.45	0.05
-1.15	0.05
-0.94	0.05
-0.76	0.05
-0.60	0.05
-0.45	0.05
-0.32	0.05
-0.19	0.05
-0.06	0.05
0.06	0.05
0.19	0.05
0.32	0.05
0.45	0.05
0.60	0.05
0.76	0.05
0.94	0.05
1.15	0.05
1.45	0.05
2.06	0.05

For any probability distribution, parameters can be provided to multiply the *s*-values in the distribution by a constant, and to shift the mean of the distribution to the right/left. For example, the standard normal distribution, *N(0,1)*, shown in Figure 5, could be used as the basis for two separate distributions for coding and non-coding sequences. The *s* of mutations within non-coding regions could be specified by transforming the standard normal distribution, multiplying all values by 0.5 and shifting the mean to the left by 0.82, creating the distribution *N(-0.82,0.5)*. In this case, 95% of mutations are deleterious and 5% are beneficial, albeit within a small range of values. Non-coding mutations,

comprising the vast majority of mutations, are therefore slightly deleterious, yet "nearly neutral." This concept is similar to Ohta (1973). For CDS sequences, the standard normal distribution can again be transformed so that 95% mutations are deleterious, but with a much wider distribution of values. This models the assumption that coding mutations usually have a much wider range of effect than noncoding mutations. A distribution like *N(-4.94,3)* has the same percentages of mutations beneficial/deleterious (5%/95 %) as the distribution for noncoding mutations, however the standard deviation of *s* is six times greater. The general idea is to provide a model where mutations in CDS regions are much less likely to be neutral—they will either be very deleterious or somewhat beneficial (Figure 6).

Figure 6: Probability distributions of *s* for coding and noncoding sequences

Transforming the standard normal distribution allows two probability distributions to be specified for coding and noncoding sequences. Although 95% of mutations for non-coding bases will have negative *s,* most of these *s* will be very close to neutral. The distribution for coding bases also has negative *s* for 95% of mutations; however the range of these values is much wider.



Of course discrete tables can be created for other types of probability distributions besides normal, like the reflected Gamma distribution, for example. The normal distribution provides a convenient example, but other distributions will eventually supplant this distribution once a more thorough literature review is done to determine the types of distributions *s* follows in common models of population genetics. The goal is to allow the user flexibility in specifying the distributions of *s* for coding and non-coding regions.

Base-substitution frequencies (i.e. *P(A→T)*, *P(A→C)*, etc) are specified in a table. In general, these substitution frequencies should be based on real studies of base substitution frequencies, with transitions more likely than transversions (Blake, Hess et al. 1992). An example is shown in Table 5.

Table 5: Substitution frequencies are required for GEM initialization

Shown is an example of a substitution frequencies table. Note that it is possible to specify a probability for synonymous substitutions (e.g. A→A), although there is no practical reason for doing so.

Starting base	Ending base	Probability
A	T	0.039
A	C	0.044
A	G	0.147
T	A	0.039
T	C	0.127
T	G	0.036
C	A	0.041
C	T	0.184
C	G	0.059
G	A	0.19
G	T	0.04
G	C	0.053

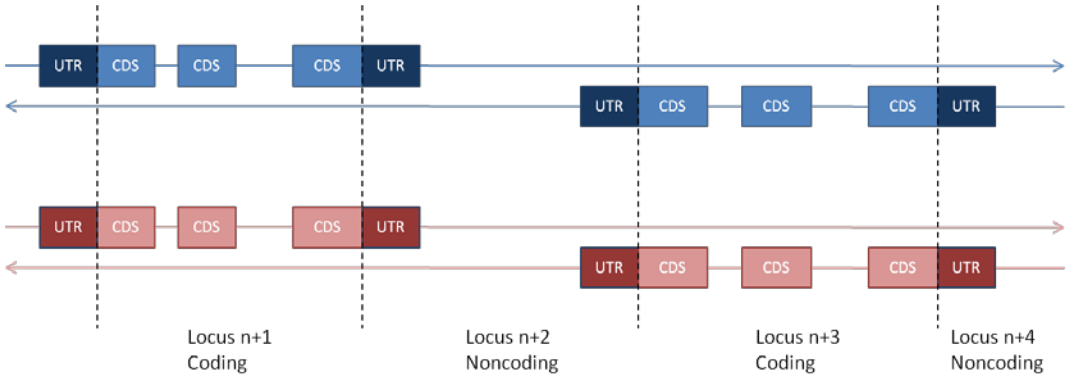
Initialization Step 1: Creation of the first individual

The GEM algorithm begins by creating an "Adam" virtual individual of which identical copies will be made to create the starting population. The virtual individual has a single pair of chromosomes each of which is represented by the chromosome sequence contig specified during initialization. The contig sequence is read into memory and coordinates of protein-coding sequences are marked along with regions mid-range inhomogeneity (MRI).

Because the GEM algorithm uses rules of dominance for calculation of fitness within coding loci, the DNA sequence is broken into coding and noncoding loci (Figure 7).

Figure 7: The chromosome sequence is broken into coding and noncoding loci

This diagram illustrates how the DNA contig loaded into GEM is broken into coding and noncoding loci. Shown is a portion of a chromosome pair from a hypothetical contig, with protein coding sequences separated from non-coding sequences. This division is necessary because a model of dominance is used to calculate fitness for coding loci (see Equation 1).



For each coding locus, a pattern of inheritance is assigned at random according to the probability distribution specified during initialization. Refer to Equation 1 and Table 3

for a complete explanation of how these patterns of inheritance work. Because patterns of inheritance are not currently annotated for most protein coding loci, GEM assigns them at random, according to a probability distribution specified during initialization (see Table 3). Therefore, different coding regions will have different patterns of dominance each time the simulation is run. Once the chromosome sequence contig has been broken into loci, these loci are stored in a special data structure; a chromosome is therefore a collection of loci. The first individual created receives two copies of this chromosome data structure.

Initialization Step 2: Generation of the *s*-value matrix for substitutions

Next, a matrix is created which contains the fitness affect for each possible substitution within a virtual individual's genome. This matrix is universal for all individuals in the population. Thus, individuals with identical sequences will have identical fitnesses, and a mutation to a specific base at a specific position will have the same s in all individuals. Note that this model does not account for epistasis.

For a population of virtual individuals with diploid genomes based on a contig of m bases, the size of this matrix will be $m \times 4$. Thus, the possible fitnesses for a base substitution at position i are given by column i in this matrix. At column i, row 1 contains the fitness for a substitution to adenine (A), row 2 for cytosine (C), row 3 for thymine (T) and row 4 for guanine (G). The substitution effect matrix is filled with values drawn from the two probability distributions specified as during initialization (see Figure 6 for these distributions); columns for bases within exons will follow the

distribution for coding regions and columns for bases within introns, untranslated regions, and intergenic regions will follow the distribution for non-coding regions. Table 6 shows a portion of a fitness matrix for the NT_011512 contig for both noncoding and coding regions.

Table 6: A matrix of s values for every possible base substitution

For a genome of size *m* GEM constructs an *m* x 4 matrix of *s* values for mutations. The values used to fill the matrix come from the probability distributions specified for both protein coding bases and noncoding bases. For NT_011512, the first protein coding region begins at position 644421. Note that the *s* for noncoding regions are close to neutral, whereas *s* for coding regions have a wider distribution (See Figure 6).

Position (bp)	1	2	3	…	644421	644422	644423
Coding?	No				Yes		
Distribution	*s~N(-0.82,0.5)*				*s~N(-4.94,3)*		
Base	**C**	**A**	**T**	…	**A**	**T**	**G**
s(→A)	-0.915	0	-0.85	…	0	-7.22	-2.66
s(→C)	0	+0.21	-0.79	…	-6.29	-4.37	-5.51
s(→T)	-0.595	-1.12	0	…	+1.24	0	-3.98
s(→G)	-0.095	-0.98	-0.66	…	-1.49	-0.59	0

Next, the MRI constant is added to or subtracted from the s values in the matrix lying within GC-rich or GC-poor MRI regions (Table 7). For bases within GC-rich MRI regions, *s* is incremented by the constant for {A,T}→{G,C} substitutions; within GC-rich region *s* is decremented by the constant for {G,C}→{A,T} substitutions. Within GC-poor MRI regions, *s* is decremented for {A,T}→{G,C} and incremented for {G,C}→{A,T}. It should be emphasized that GC MRI regions are only one of many types of functional DNA that could be used to further alter s for each genotype. GEM

currently accounts only for GC MRI regions, but flexibility for incorporating other regions will be added.

Table 7: Base substitutions within MRI regions

Within MRI regions, a constant is added to the value of *s* from the mutation matrix. This constant is specified in the initial conditions. Base substitutions that preserve or promote GC richness within GC-rich MRI will have this Δs added to *s,* substitutions that erode GC richness will have this constant subtracted. The same logic applies to GC-poor (AT-rich) regions of mid-range inhomogeneity. For NT_011512, a GC-rich MRI region begins at position 71344. Shown are the Δs for mutations in a portion of this region, using an MRI constant of 0.1.

Position (bp)	1	2	3	…	71344	71345	71346
GC MRI?	No	No	No		GC-rich	GC-rich	GC-rich
Starting Base	**C**	**A**	**T**	…	**A**	**C**	**C**
Δs(A)	0	0	0	…	0	-0.1	-0.1
Δs(C)	0	0	0	…	+0.1	0	0
Δs(T)	0	0	0	…	0	-0.1	-0.1
Δs(G)	0	0	0	…	+0.1	0	0

After the matrix is filled with values from either distribution, a fitness of zero is replaces the values in the matrix for neutral substitutions (i.e. all cells for A→A, T→T, C→C, and G→G will have zero fitness effect). This step is superfluous, perhaps, as the substitution frequencies should be set such that synonymous mutations never occur (see Table 5). However, it is possible for the same base to be mutated more than once, and therefore it is possible for a series of substitutions to return a base to its original value (e.g. A→T→A). In this case, the *s* for the genotype at this position will return to 0.

Initialization Step 3: Creation of a genetically identical starting population

Once the first virtual individual's genome is read in from file, broken into coding and noncoding loci, and loci are assigned patterns of inheritance, this virtual individual ("Adam") is copied to create a genetically identical founder population. Gender is assigned at random to each individual in this founder population, with an equal probability of male or female.

Execution Step 1: Germline Mutations

The first generation of individuals will be genetically identical. In order to create the next generation of individuals, a process of mutation is performed on each individual in the population. These mutations can be considered germline, since they do not affect the phenotype of the individual they occur in; they are transmitted in gametes. Base substitutions are performed at random positions in the maternal and paternal chromosomes; the starting base and ending base are generated at random according to a probability distribution specified by the user (see Table 5).

A position is chosen at random within the genome until the base at that position matches the starting base for the substitution. The starting base is then mutated to the ending base, according to the substitution probability distribution. The *s* for the new genotype is assigned according to the fitness table for the locus (see Tables 6 and 7).

Execution Step 2: Mating

After germline mutations are performed on individuals in the population, mating pairs are selected at random. The number of mating pairs selected is equal to one half of the population size. On average, each individual in the population will mate with one other individual. Because the pairing is done at random, however, some individuals will mate with multiple partners while others will not mate at all. To form a mating pair, random individuals are sampled until a male is found and, again, the sampled until a female is found. The number of offspring per mating pair is given by the Poisson distribution, with the Poisson parameter λ_o equal to the mean number of offspring per mating pair. This is in accordance with other models of random mating (Barton 2007).

Execution Step 3: Gametogenesis

Gametes are generated for each individual in the mating pair. The number of crossovers per gamete follows a Poisson distribution (Haldane 1919). Haldane defined the unit of genetic distance, the Morgan, as the expected number of crossovers between two loci. The mean number of crossovers per gamete, λ_x, is equal to genetic distance in Morgans (Tesler 2009). According to recombination data from the International HapMap Consortium, the NT_011512 contig has a genetic distance of 52.1cM. Therefore if the number of crossovers per meiosis is represented by a random variable X_x, $X_x \sim Poisson(\lambda_x = 0.521)$.

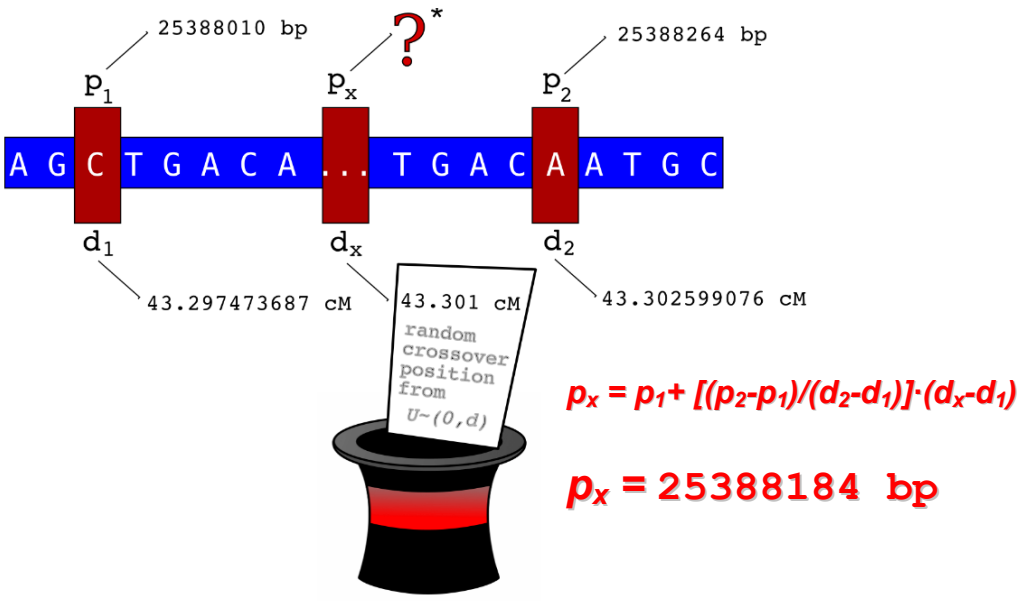
Once the number of crossovers for a gamete is determined, the locations of each of these crossovers are determined using a uniform process. If X is a random variable

representing the genetic distance of each crossover in Morgans, then $X \sim U(0,d)$ where d is the total genetic distance of the chromosome in Morgans. For each crossover, a random number is drawn between 0 and the total genetic distance (d) of the GEM chromosome sequence contig and this random number gives the genetic distance of the crossover in Morgans.

The positions of each crossover in Morgans are next converted to positions in bp, using the genetic map loaded during initialization (Figure 8). For NT_011512, a random number between 0 and d=52.1 is generated for each crossover, giving a genetic distance in Morgans, d_x. The two markers flanking this position in Morgans are located in the genetic map. If the positions of these two markers in bp are *p1* and *p2* and the distances of these markers in Morgans is *d1* and *d2*, the position in bp for the crossover, p_x, is equal to $p_1 + [(p_2-p_1)/(d_2-d_1)] \cdot (d_x-d_1)$.

Figure 8: Estimating crossover position in bp from genetic distance in NT_011512

For each crossover, a uniformly distributed random number is generated between 0 and total genetic distance of the chromosome (d) in Morgans. In this example, the number 43.301 is drawn. The nearest markers in the genetic map, with positions *p1* and *p2*, and genetic distances *d1* and *d2*, are used to estimate the position of the crossover in bp at 25388184 bp*.



After the number and position of crossovers is determined for an individual, a haploid gamete is generated for each offspring in the individual's mating pair. A gamete is created from an individual by first selecting, at random, the maternal or paternal chromosome. The original sequence is copied from the randomly chosen chromosome, up until the first crossover, where bases are then copied from the opposite chromosome until the next crossover, or the end of the chromosome (whichever comes first).

Execution Step 4: Fertilization; Birth of offspring

For each offspring in a mating pair, a gamete from the male and female are generated; these gametes join to create a new virtual individual (offspring). Each new offspring is assigned a gender at random and is pooled with the offspring from other mating pairs to form the next generation.

Execution Step 5: Assessment of offspring fitness

The new generation of offspring is assessed for fitness, based on the mutations that occurred in the parental gametes. Recall that each individual's chromosomes are broken into coding and noncoding loci at the exact same points. For each individual, overall fitness is calculated by iterating through each locus and calculating fitness, F, for that locus according a) the sums of maternal and paternal s values within the locus, Σs_m and Σs_p, and b) the pattern of inheritance that was assigned to that locus during initialization (see Equation 1). For noncoding loci, F is calculated simply as the average of the fitness of the maternal and paternal loci. For example, referring to Figure 7, locus n is noncoding, so F for this locus is the average fitness for the maternal and paternal alleles present at this locus. The next locus is coding. Its fitness is determined according to Equation 1 using the coefficients a, b, and c assigned to this locus during initialization. This process continues for every locus in the diploid chromosome pair, coding and noncoding. The net fitness for the entire chromosome pair, and thus the individual, is the sum of fitnesses for all loci, ΣF_{loci}.

Execution Step 6: Selection

After fitness is calculated for each offspring in the new generation, offspring are ranked according to their fitness. The size of the new population is truncated to the size of the population of progenitors. For example, a population of N=100 individuals, with an average of λ_o=3 offspring per mating pair, will produce an average of $N/2\times\lambda_o$ = 150 offspring. However, to keep population size constant, only the N fittest individuals are considered fertile (able to reproduce). Therefore, the less fit remaining individuals are removed from the simulation. This simple step is crucial, as this is where Darwin's process of natural selection occurs at the fullest extent possible; only the very strongest individuals are able to survive and reproduce (Darwin 1859).

Execution Step 7: Young replace the old; Go to step 1.

After truncation selection, the remaining individuals in the new generation of offspring become the new population (i.e. the parents eliminated). The process of sexual reproduction is then repeated for as many generations as specified by the user. At each generation, the average fitness of the population is calculated.

GEM v0.45 Example

Introduction

Here I will present a brief example of output from GEM v0.45. In this example, the basic GEM algorithm has been modified by adding an alternate method of calculating number of offspring for each mating pair. Normally, the mean number of offspring per mating pair λ_o is used as the parameter to a Poisson process; if X_o is a random variable representing the number of offspring for a particular mating pair, $X_o \sim \text{Poisson}(\lambda_o)$.

In the alternate scheme for calculating number of offspring per mating pair, the fitness percentile ranks of both parents (PR_m and PR_p) are considered when calculating number of offspring per mating pair and the mating pairs with higher percentile ranks are allowed to have more offspring. I call this dependence of number of offspring on the fitness of the mating pair "fitness-fertility" dependence. For a population of 100, the fittest individual has a percentile rank of 100%; the least fit individual has a percentile rank of 1%. In this alternate scheme, if X_o is a random variable representing the number of offspring for a particular mating pair, $X_o \sim Poisson(\lambda_o * (PR_m + PR_p))$.

The expected number of offspring for a given mating pair, $E(X_o)$, is the same for both schemes of calculating offspring for each mating pair, since the mean value of $PR_m + PR_p \approx 1$ for all mating pairs in a population. Therefore, the total number of offspring produced per generation is the same, on average, for both schemes. However, in the scheme where the Poisson parameter is multiplied by the sum of the percentile ranks of both parents, the couples with greater fitness will produce more offspring. Couples with fitness near the bottom percentile will produce few, if any offspring. The working

hypothesis of this simple experiment is this: populations where number of offspring is dependent on fitness of the parents will increase in fitness faster than populations where all mating pairs the same mean number of offspring (λ_o).

Methods

Two sets of trials were run in GEM to compare these methods of calculating number of offspring per mating pair. Ten trials were run with number of offspring per mating pair $X_o \sim Poisson(\lambda_o)$; ten trials were run with $X_o \sim Poisson(\lambda_o * (PR_m + PR_p))$. All trials were run with a population of 500 individuals for 10,000 generations, with 3 mutations per individual per generation, and 0 for the MRI constant.

For both sets of trials, the distribution of *s* was set to *N(-0.82,0.5)* for noncoding regions and *N(-4.94,3)* for coding regions. The genetic map specific for NT_011512 was used to model crossover. The substitution frequencies from Table 5 were used for both sets of trials. Coding regions were loaded from a file specific for NT_011512. The probability distribution for assignment of dominance to coding loci from Table 3 was used.

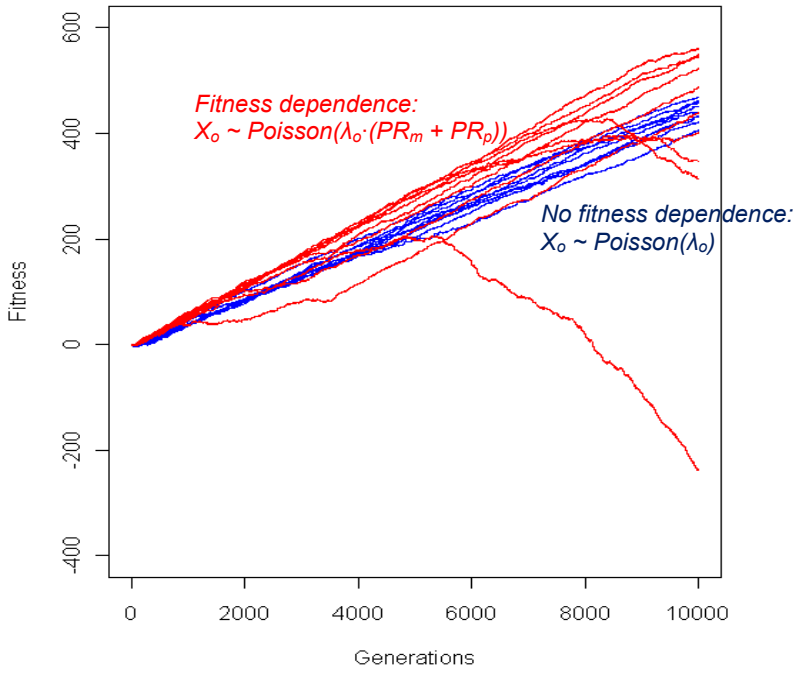
Results

Figure 9 shows the fitness of each population over the course of 10,000 generations. In populations where the fittest individuals produce the most offspring and the least fit produce little to no offspring (fitness-fertility dependence), fitness initially increases more quickly compared to the population where all mating pairs have the same

distribution for number of offspring. However, beginning at 1000 generations, some of the populations with fitness dependence begin to decline in fitness. By the end of 10,000 generations fitnesses for five of the populations with fitness-fertility dependence have dropped sharply; the four least fit populations, of twenty, are from the populations with fitness fertility dependence.

Figure 9: Results of GEM Example

The populations where the majority of offspring come from the fittest individuals show a major advantage in fitness for the first ~1000 generations. Soon after, several of these populations show a sharp decline in fitness. This result may be due to inbreeding depression, where beneficial alleles from low fitness individuals are lost from the population because these individuals have few, if any, offspring.



Conclusions

Intuitively, one may assume that the fittest individuals carry the fittest (highest *s*) genotypes and therefore allowing these individuals to produce more offspring than less fit individuals will allow the fittest genotypes to propagate in the population. With this in mind, one might expect that the fitness of a population with such a breeding scheme would increase more quickly than in a population where all mating pairs have the same distribution for number of offspring. However, Figure 9 tells a different story. For approximately the first 1000 generations, the populations with number of offspring dependant on mating pair fitness clearly increase in fitness more quickly than the population where all mating pairs have the same mean number of offspring. Soon after, fitnesses begin to decline in several of these fitness-fertility dependent populations. By the end of the simulation, the fitness-fertility dependent populations have a mean fitness of 160.7, compared to 419.4 for the populations with no fitness-fertility dependence.

A possible explanation is that these populations with sharp declines in fitness undergo a genetic bottleneck event that as a result of decreasing genetic diversity. Although the individuals with low fitness have a larger net effect of deleterious genotypes than individuals with high fitness, these individuals with low fitness still undoubtedly carry some beneficial alleles.

Allowing only the fittest mating pairs to produce large numbers of offspring, with few or little offspring produced by low fitness couples, will result in a loss beneficial alleles carried by low fitness individuals in the population. It has been shown that loss of allelic diversity from a population has a negative impact on fitness, due to inbreeding depression (Vandewoestijne, Schtickzelle et al. 2008; Markert, Champlin et al. 2010). Perhaps this

explains the phenomena observed in the artificial populations where only the fittest individuals produce sizeable numbers of offspring; alleles are lost from the population because the least fit individuals produce few offspring, and the loss of allelic diversity results in a type of inbreeding depression.

While this experiment was conducted solely for the purpose of demonstrating how parameters interact in a GEM simulation, these results are noteworthy and merit further investigation.

Summary

In this thesis, I have described the rationale for developing a model of genome sequence evolution of the course of generations, the Genome Evolution Model (GEM). The specification of an algorithm for a very ambitious model has been provided, a model genome sequence evolution through the process of natural selection. This algorithm allows observations to be made and inferences drawn about the dynamics governing base substitutions and crossover, the impact of mutations on individual and population fitness, and the interplay of factors governing natural selection and population fitness.

While GEM stands in the shadows of the pioneering work by population geneticists like Mendel, Fisher, Haldane, Wright, Muller, Kimura, Crow, Ohta, and many others, the GEM algorithm is nonetheless likely to be the most sophisticated computer algorithm-based model of population genetics in existence today. GEM simulates the genotypic and phenotypic evolution of a population of artificial organisms that have a single diploid chromosome, represented by a real 28.6Mbp DNA sequence from *H. sapiens* chromosome 21.

The current specification of GEM (v0.45) incorporates real data about base substitution frequency and crossover rate and location from experimental studies. The impact of this modeling will be realized once future versions of GEM, able to model the effects on selective advantage of protein substitutions and codon bias, are complete. A future goal of the GEM project is to incorporate better rules for calculating selective advantage when substitutions occur within protein coding regions. Synonymous mutations toward favored codons will be modeled with positive selective advantage (s); synonymous

mutations toward less favored codons will have a neutral or perhaps very slightly negative selective advantage. Mis-sense mutations, changing the amino acid encoded by a codon, will have a distribution of selective advantage based on PAM matrices (Wilbur 1985); evolutionarily favored amino acid substitutions will be assigned a higher selective advantage than less favored substitutions.

The current specification of GEM (v0.45) incorporates an enormous amount of real data on substitution frequency, crossover locations, and the location of regions of mid-range inhomogeneity (MRI). Additionally, this model incorporates a model of dominance for mutations, and allows the user to specify probability distributions for the selective advantage (s) of base substitutions inside and outside coding regions. With the incorporation of rules for mutations in protein coding regions based on PAM matrices and codon bias tables, GEM will provide a very useful model of eukaryotic sequence evolution.

In the future, GEM could be used to model how codon bias evolves over the course of hundreds of thousands of generations. Rules for calculating *s* could be specified such that synonymous mutations from less favored codons to favored codons would have a greater value of *s*. The sequences of artificial individuals could be sampled at different time points during the simulation, to examine how codon bias evolves according to the rules for calculating *s* in synonymous mutations.

Additionally, my group is interested in studying the evolution of regions of mid-range inhomogeneity. For mutations within regions of mid-range inhomogeneity, those that preserve or enrich mid-range inhomogeneity (e.g. GC richness in a GC MRI region)

should have higher values of *s* than mutations which erode the MRI region (e.g. mutations to A or T in a GC-rich MRI region). As with the codon bias example from the previous paragraph, the sequences within MRI regions of artificial individuals could be sampled at different time points, illustrating how MRI regions evolve according to the rules the user specifies.

While the demonstration experiment using GEM v0.45 was intended merely to show a practical application of the program, it provided some interesting results suggesting that GEM is capable of modeling inbreeding depression due to the loss of alleles from a population. These results may very well serve as the basis for further research and publication.

In summary, GEM is a unique and sophisticated model of DNA sequence evolution and the dynamics governing population fitness. While the GEM project is in its infancy, I anticipate that GEM will continue to generate greater interest in the scientific community. The matter of how much interest GEM will generate remains to be seen, but this project undoubtedly has a great deal of potential, and will continue to improve with refinement.

References

Banzhaf, W. (1998). Genetic programming : an introduction on the automatic evolution of computer programs and its applications. San Francisco, CA, Morgan Kaufmann Publishers.

Barton, N. H. (2007). Evolution. Cold Spring Harbor, N.Y., Cold Spring Harbor Laboratory Press.

Blake, R. D., S. T. Hess, et al. (1992). "The influence of nearest neighbors on the rate and pattern of spontaneous point mutations." J Mol Evol **34**(3): 189-200.

Darwin, C. (1859). On the origin of species by means of natural selection, or preservation of favoured races in the struggle for life. London, John Murray, Albemarle Street.

Felsenstein, J. (1981). "Evolutionary trees from DNA sequences: a maximum likelihood approach." J Mol Evol **17**(6): 368-376.

Frazer, K. A., D. G. Ballinger, et al. (2007). "A second generation human haplotype map of over 3.1 million SNPs." Nature **449**(7164): 851-861.

Goldberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning. Reading, Mass., Addison-Wesley Pub. Co.

Haldane, J. S. (1919). "The combination of linkage values and the calculation of distance between loci of linked factors." J Genet(8): 299–309.

Hasegawa, M., H. Kishino, et al. (1985). "Dating of the human-ape splitting by a molecular clock of mitochondrial DNA." J Mol Evol **22**(2): 160-174.

Int. Soc. for Complexity, I., and Design (2010). selection coefficient. ISCID Encyclopedia of Science and Philosophy.

Jukes, T. and C. Cantor (1969). Evolution of Protein Molecules. New York, Academic Press.

Kimura, M. (1980). "A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences." J Mol Evol **16**(2): 111-120.

Markert, J. A., D. M. Champlin, et al. (2010). "Population genetic diversity and fitness in multiple environments." BMC Evol Biol **10**: 205.

Merritt, D. (2008). "Genetic Algorithm." Retrieved 12 Oct, 2010, from http://www.c2.com/cgi/wiki?GeneticAlgorithm.

Ohta, T. (1973). "Slightly deleterious mutant substitutions in evolution." Nature **246**(5428): 96-98.

Ohta, T. (1992). "The Nearly Neutral Theory of Molecular Evolution." Annu. Rev. Ecol. Syst.(23): 263-286.

Prakash, A., S. S. Shepard, et al. (2009). "Evolution of genomic sequence inhomogeneity at mid-range scales." BMC Genomics **10**: 513.

Rearick, D., A. Prakash, et al. (2010). "Critical association of ncRNA with introns." Nucleic Acids Res.

Rhead, B., D. Karolchik, et al. (2010). "The UCSC Genome Browser database: update 2010." Nucleic Acids Res **38**(Database issue): D613-619.

Ridley, M. (2004). Evolution. Oxford ; New York, Oxford University Press.

Tamura, K. (1992). "Estimation of the number of nucleotide substitutions when there are strong transition-transversion and G+C-content biases." Mol Biol Evol **9**(4): 678-687.

Tamura, K. and M. Nei (1993). "Estimation of the number of nucleotide substitutions in the control region of mitochondrial DNA in humans and chimpanzees." Mol Biol Evol **10**(3): 512-526.

Tatusova, T. (2010). "Genomic databases and resources at the National Center for Biotechnology Information." Methods Mol Biol **609**: 17-44.

Tavaré, S. (1986). "Some probabilistic and statistical problems on the analysis of DNA sequences." Lec. Math. Life Sci. **17**: 57-86.

Tesler, G. P. (2009). "Poisson Distribution: Counting Crossovers in Meiosis." Retrieved 20 Oct 2010, from http://www.math.ucsd.edu/~gptesler/186/186_poisson_09-handout.pdf.

Vandewoestijne, S., N. Schtickzelle, et al. (2008). "Positive correlation between genetic diversity and fitness in a large, well-connected metapopulation." BMC Biol **6**: 46.

Wilbur, W. J. (1985). "On the PAM matrix model of protein evolution." Mol Biol Evol **2**(5): 434-447.

Wolfram, S. (2002). A new kind of science. Champaign, IL, Wolfram Media.

Ye, T. Z., R. C. Yang, et al. (2003). "Coevolution in natural pathosystems: effects of dominance on host-pathogen interactions." Phytopathology **93**(5): 633-639.

Abstract

The fields of population genetics, evolutionary computing, and artificial life have existed as separate domains for a number of decades. Here, I present an ambitious model of the relationship between DNA sequence evolution and the dynamics of sexual reproduction using concepts from these different fields. While models of DNA sequence evolution have been created using continuous-time Markov models, these models do not allow close observation of the processes of mutation, gametogenesis, and sexual reproduction; most of these models do not account for the context that base substitutions have within functional regions of a genome (e.g. coding and untranslated exons, introns, intergenic regions, isochors, promoters, etc). The Genome Evolution Model (GEM) described in this thesis models DNA sequence change in the context of fitness and selective pressure. I describe the operation of GEM, and provide an example of its application to modeling population dynamics. The included experiment, intended to demonstrate how GEM operates, also shows intriguing results suggesting that GEM may be able to model inbreeding depression as a result of decreased genetic diversity.

Appendix A: Structure of the GEM Java program

Figure 10 shows the classes in the GEM software package. The classes are divided into the default package and eight additional packages. Figure 11 shows the information flow between the most important classes in the GEM software package.

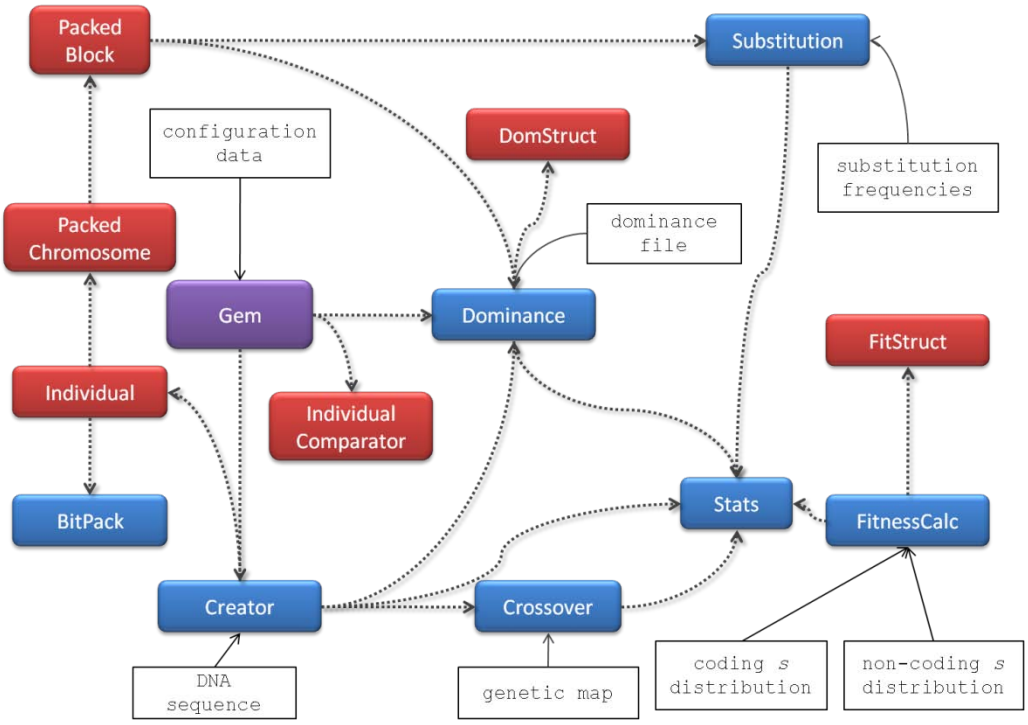
Figure 10: Packages and classes in the GEM software package

The GEM program is written in Java. It consists of 20 classes, divided into the default package and 8 packages. Shown is a list of the packages in the Gem Java program, and the classes within those packages.

```
(Default Package)
      Gem.java
      IndividualComparator.java
bitpack:
      BitPack.java
      PackedBlock.java
      PackedChromsome.java
dominance:
      Dominance.java
      DomStruct.java
engine:
      CodonUsage.java
      Crossover.java
      Stats.java
fitness:
      FitnessCalc.java
      FitStruct.java
gene:
      Exon.java
      Gene.java
individual:
      Creator.java
      Individual.java
substitution:
      SubStruct.java
      Substitution.java
tools:
      Clock.java
      Consts.java
      Sequence.java
```

Figure 11: Information flow between the major classes in GEM

Ths diagram summarizes information flow between the major classes in the GEM software package. Classes are represented as rounded rectangles. Blue classes are used in static context and are not instantiated. Red classes are instantiated into object instances in memory. The main class, Gem, is where excecution begins and is colored purple. Data files used for input to the program are represented as white rectangles. Solid arrows indicate flow of data from data files to the class where the data is stored. Dotted arrows indicate function calls from one class to another, including calls to constructors.



The main class of the GEM program is defined in `Gem.java`, in the default package. To run the GEM program, the user types "`java Gem config.in`" "`config.in`" is the name of the configuration file. The configuration file is a tab-delimited file containing parameters for the simulation and their values.

The `main` function of the `Gem` class begins by reading simulation parameters from the configuration file and storing values for these parameters in static member variables in the `Gem` class. For example, if the line "`chromosome_filename NT_011512.fa`" is read

from the configuration file, the value "`NT_011512.fa`" is stored in the static member variable `chrFileName_` in the class `Gem`.

After reading configuration parameters, the `engine.Stats` class is initialized by a call to the static `Stats.init` function from the `Gem` class. The `Stats` class contains a random number generator. There is only one random number generator used throughout the entire program. This is a static member variable of type `Random` in the `Stats.java` class. The purpose of having one random number generator is to make results repeatable by using a single seed. Additionally, instantiating multiple random number generators may sometimes cause erratic results in Java. The random number generator is initialized using the system time as the seed, although this seed can be changed to a constant value for debugging purposes.

Next the class `dominance.Dominance` is initialized with a call to its static `init` function. The init function takes one parameter: the filename giving the distribution of dominance coefficient combinations. The `Dominance` class reads this file and stores a probability for each combination of coefficients specified in the file.

The class `substitution.Substitution` is initialized with a call to its static `init` function. The `init` function takes a single parameter: the filename of the table giving the frequencies of each possible substitution. The frequency of each possible substitution is stored for later use.

Next the `individual.Creator` class is used in a static context to create a single copy of the first virtual individual. The first function call is to the static function `init`, which takes the name of the FASTA formatted file with the contig to be used as the genome for

each virtual individual. The sequence is assembled and stored. The next call is to the `readExons` function which takes the filename of the file which has all of the exons demarcated in it as an argument. The start and finish of each CDS exons of each gene in the contig is read in. Note that UTR exons are not usually marked in this file, and that untranslated portions of CDS exons are excluded. Therefore, the start of the first exon for any gene is marked according to where the first amino acid (Methionine) is encoded and the end of the last exon is marked according to where translation ends. For each gene, a new `Gene` object is created, and exon read from the file is used to create an instance of the `gene.Exon` class which is added to the appropriate Gene class. Once all exons for a particular gene have translated into instances of the Exon class, and these instances of the Exon class have been added to the appropriate Gene class, the start and end of the gene are marked with the `Gene.computeIndexes` function. Each Gene class instance is then added to a hash, once the start and end have been computed.

Iterating through the Genes in the hash allows two `boolean` arrays to be filled with values. A `boolean` array within the Creator class called `isExonic_` stores `true` for every position in the contig that is exonic and a `false` for every position that is not exonic (i.e. noncoding). A `boolean` array within the `Creator` class called `isCDS_` stores `true` for every position in the contig that is coding and `false` for every position in the contig that is noncoding.

Similarly, the `Creator.readMRI` function takes the filename of a file similar in format to the exon file as a parameter. This file contains the positions of MRI regions within the contig of interest. Currently, the code for the GEM software package only considers GC-

rich MRI regions, although this can easily be modified in the future. The array `MRI_` is first filled with zeros. Next, a +1 is stored in the array at every position within a GC-rich MRI region, and a -1 is stored at every position within a GC-poor MRI region.

The `Creator.createIndividual` function is called to create the first individual that will be used as a template for every individual in the initial population. The function takes an argument giving the maximum locus size for noncoding regions (`maxBlockSize`). Noncoding regions that are larger than this value will be broken into pieces equal in size to this value. The `Creator` class proceeds to create an instance of the `bitpack.PackedChromsome` class. To this `PackedChromosome` class are added instances of the `bitpack.PackedBlock` class. Each `PackedBlock` contains a portion of the contig sequence loaded into `Gem`; the sequence of the contig is broken into coding and noncoding portions, using the `isCDS_ boolean` array. Coding and noncoding portions are separated into different instances of `PackedBlock`. Each `PackedBlock` is initialized using a constructor that accepts the appropriate portion of the contig DNA sequence as an argument. This DNA sequence is "packed" from a series of 16 bit characters into an array of type `byte`, reducing memory usage eight-fold. For each coding `PackedBlock` that is created, a pattern of dominance is assigned by a call to the static `Dominance.getRandomDominance` function. This function returns an object of type `DomStruct` (also from the `dominance` package). Within the `DomStruct` object are the dominance coefficients described earlier. The values of the coefficients from the `DomStruct` object are used to assign values to the member variables `maxCoefficient_`, `minCoefficient_`, and `meanCoefficient_` within the new coding `PackedBlock`.

Each noncoding portion is stored as a single `PackedBlock` unless it is larger than the `maxBlockSize` argument, in which case it is broken into pieces. For example with a `maxBlockSize` of 10,000 bp, a 22,000 bp noncoding segment will be divided into three instances of `PackedBlock`: two 10,000 bp `PackedBlock` instances and one 2,000 bp `PackedBlock` instance. Once the first instance of `bitpack.PackedChromosome` has been initialized and all of the corresponding instances of `PackedBlock` for each section of sequence are added, a new instance of `individual.Indivudual` is created using two copies of the `PackedChromosome` as its genome. This `Individual` instance is stored in the `Creator` class for later use.

Next the `engine.Crossover` class is initialized with a call to its `init` function, which takes the filename for the genetic map as an argument. The `init` function reads the genetic map and stores the corresponding position in bp for each genetic distance listed in centiMorgans.

The last class to be initialized before the simulation starts is the `fitness.FitnessCalc` class. This class takes several arguments. The first argument is the sequence of letters in the genome. Also passed as arguments are the filenames for the files containing the discrete probability distributions of *s* within coding and noncoding regions. For both of these distributions, a "shift" and a "multiplier" are provided as arguments. Each discrete value in the probability distribution files provided for coding and noncoding sequences is multiplied by the multiplier and then the "shift" is added. At the time of this writing, the only probability distribution file that has been created is a discrete approximation to the standard normal distribution. In this case, the "multiplier" argument specifies the

standard deviation, and the "shift" argument specifies the mean. The last argument passed to the `FitnessCalc.init` function is the MRI constant. For a contig of size m bases, The `FitnessCalc` class creates an m×4 matrix of *s* values for every possible substitution. The matrix is filled with *s* values drawn from the distribution for coding and noncoding sequences. The `FitnessCalc` class refers to the `Creator.MRI_` array to determine whether a given base position is within an MRI region. Those substitutions within MRI regions will have the MRI constant added or subtracted, depending on the base substituted. This is described in the GEM algorithm chapter.

At this point `Gem` class has stored the parameters from the configuration file. Calls from the `Gem` class initialized the `Stats`, `Dominance`, and `Substitution` classes for their use in static context. The `Creator` class has been initialized and a single instance of the `Individual` class (`Creator.adam`) has been created. The `Crossover` and `FitnessCalc` classes have also been initialized for use in static context. The next step is for the `Gem` class static member function `runSimulation` to execute. The `runSimulation` function is the workhorse of the `Gem` class. Once the `runSimulation` function is done executing, the program exits.

The `runSimulation` function first creates an instance of the `IndividualComparator` class from the default package. The `IndividualComparator` is a subclass of the `java.util.Comparator` interface. A custom `Comparator` is necessary in order to sort a collection of custom objects. Because the population of `Individual` objects must be sorted by fitness each generation, a custom `Comparator` is needed to compare the fitness of one `Individual` to another.

The next step in the `Gem.runSimulation` function is the creation of a `Vector` object (from `java.util.Vector`) called `population_` to hold all of the individuals in the population. The size of the `population_` vector is set equal to the population size specified earlier in the configuration file. Next, this vector is filled with copies of the "Adam" individual from the `Creator` class. Each of these new Individual is assigned a value for member variable `gender_` of `Consts.MALE` or `Consts.FEMALE` at random.

At this point, a population of identical individuals with no mutations has been created. Now, execution proceeds to the main for loop of the `runSimulation` function, the "generation loop." This `for` loop increments an integer variable called `gen` representing the current generation. The loop executes until `gen` reaches the number of generations specified in the configuration file.

Within the generation loop, the first step is to create temporary `Vector` called `nextGeneration` is created to which the offspring will be added after mating. The next step is to apply mutations to sequences within the `maternal_` and `paternal_` instances of `PackedChromosome` of the `Individual` objects in the `Vector` `population_`.

For each `Individual` object in the population vector, the member function `randomMutation` of class `Individual` is called. The `randomMutation` function takes one argument: the number of mutations to apply. For each mutation in the `Individual`, one of the two `PackedChromosome` objects, `maternal_` or `paternal_`, is selected. The appropriate `PackedChromsoome` member function (`maternal_.randomMutation` or `paternal_.randomMutation`) is then called. The member function `randomMutation` of the `PackedChromosome` class generates a random position somewhere along the entire

length of the chromosome in which is used to select a `PackedBlock` within which the mutation will occur (the sequence data is divided into blocks).

Just as with the `Individual` and `PackedChromosome` classes, the `PackedBlock` class also has a member function `randomMutation` which is called when the `PackedBlock` within which the mutation will occur is selected. The first step of this function is to call the static member function `getPair` from the class `substitution.Substitution`. Recall that the `Substitution` class is initialized with the probabilities of each possible substitution (P(A→T), P(A→C), P(A→G), etc). The function `getPair` returns a random substitution to be performed within the `PackedBlock`, in the form of a `String` with two characters. For example, if `getPair` returns "`CT`," the substitution C→T is to be performed. Random positions within the `PackedBlock` are tried until the correct starting base is found. For example, for C→T, random positions are found until the base "C" is found at that position. A new substitution is generated if the correct starting base cannot be found after sampling many positions within the `PackedBlock`, although this is very rare.

Each `PackedBlock` has a member `TreeSet` (from `java.util.TreeSet`) called `mutationIndices_` which stores the location of every mutation within the sequence in the `PackedBlock`. Once a position with the correct starting base is found within the `PackedBlock` being mutated, its position is added to the `TreeSet` `mutationIndices_`. Next, the sequence data stored within the `PackedBlock` is modified to reflect the substitution. The *s* value for the mutation is retrieved from the substitution matrix stored in the `FitnessCalc` class, using the `FitnessCalc.getFitnessAt` function. Each

`PackedBlock` has a member variable called `fitness_`. The value of `fitness_` is incremented by the *s* value of the mutation.

The `randomMutation` function exits from the `PackedBlock` instance, and the `randomMutation` exits from the `PackedChromosome` instance. If more mutations are to be performed in the `Individual` object where the `randomMutation` function was called, the `maternal_` or `paternal_ PackedChromosome` objects are again chosen at random, and the member function `randomMutation` of `PackedChromosome` is again called. When no more mutations are to be performed in the `Individual`, the next `Individual` from the `population_` vector is selected for mutation.

Once all `Individual` objects in the `population_` vector have been mutated, the process of mating begins. For a population of size *N*, a for loop is executed for *N/2* iterations to create *N/2* mating pairs. Within this for loop, random indices within the `population_` vector are sampled until a female is found; random indices are next sampled until a male is found. One male and one female comprise a mating pair; once a mating pair is formed, the number of offspring per mating pair is calculated. The number of offspring is determined by a call to the `Stats.poisson` function. The mean number of offspring is given as an argument to the poisson function, which then returns a random number from a Poisson process with lambda equal to the mean number of offspring.

For each of the x_o offspring to be created, a gamete is generated from the mother and father `Individual` objects using the member function of the `Individual` class `getGamete`. The `getGamete` function is called for both the male and female `Individual` objects. The `getGamete` function returns a `PackedChromosome` object. These two

`PackedChromosome` objects, returned from the male and female `getGamete` function are joined to create a new offspring instance of the `Individual` class. Each new offspring requires a separate call to the `getGamete` functions from the parents.

The `getGamete` function begins by determining the number of crossovers that will occur between the parental chromosomes. This is determined by a call to static member function `getNumCrossovers` of the class `engine.Crossover`. The genetic distance in Morgans of the contig, according to the genetic map that was used to initialize the Crossover class, is used as the lambda parameter to a Poisson process which generates a random number of crossovers. If the number of crossovers is zero, either the `PackedChromosome maternal_` or `paternal_` is selected at random and returned as the gamete. Otherwise, the maternal or paternal sequence is chosen at random to begin the gamete. The position of each crossover is given in bp by a call to the static member function `Crossover.getRandomCrossover`. This function first calculates the position of the crossover as a random number between 0 and the genetic distance of the sequence contig, in Morgans. Using the genetic map loaded previously into the `Crossover` class, during initialization, the position in bp is calculated and returned from the function. At each crossover point, the current sequence (maternal or paternal) is switched in the gamete. The gamete is generated by a process of adding `PackedBlock` instances from the `maternal_` and `paternal_ PackedChromosome` objects to a new `PackedChromosome` instance. Most crossovers will occur in the middle of a `PackedBlock`. When this happens, a new `PackedBlock` containing a portion of sequence from the same block number maternal and paternal `PackedBlock` sequences is created by a call to the `BitPack.CrossBlocks` function, which returns a new `PackedBlock`. This hybrid

`PackedBlock` is added to the gamete in the same fashion that maternal and paternal instances of `PackedBlock` are added to create a complete gamete `PackedChromosome`, which is returned from the `getGamete` function.

The process of selecting a male and a female repeats for *N/2* mating pairs, each offspring is added to the `nextGeneration` vector. The reference for the `population_` vector is changed to point to the memory location of `nextGeneration` vector and the reference for the `nextGeneration` vector is set to `null`. Thus, the contents `nextGeneration` vector, containing all of the offspring, becomes the new `population_` vector. The `population_` vector is sorted using the `IndividualComparator` class. The `population_` vector is then truncated to the value of `pSize_`.

This completes one generation of evolution in `Gem`. The main loop of the `runSimulation` function runs for as many generations as specified by the user.

Appendix B: Source code for GEM v0.45

Gem (default package)

```
// Gem.java, default package
// Copyright 2010 Andrew McSweeny
// All rights reserved

import individual.Creator;
import individual.Individual;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Hashtable;
import java.util.StringTokenizer;
import java.util.Vector;
import java.util.Collections;

import dominance.Dominance;

import substitution.Substitution;
import tools.Clock;
import tools.Consts;

import engine.Crossover;
import engine.Stats;
import fitness.FitnessCalc;


class Gem {

        //Variables read from configuration file
        public static String codonFileName_;
        public static String chrFileName_;
        public static String exonFileName_;
        public static String mutationFileName_;
        public static String crossoverFileName_;
        public static String fitProbFileName_;
        public static String CDSfitProbFileName_;
        public static String MRIFileName_;
        public static String dominanceFilename_;
        public static float MRIConstant_;

        public static boolean fertilityFitnessDependence_;
        public static boolean verbose_;

        public static int maxSize_=Consts.MAX_INT;
        public static int crossoverMultiplier_;

        private static int maxBlockSize_;
        private static int mutPerGen_=100;
        private static int nGen_=1000;
        private static int pSize_=100;

        private static int offspringPerMatingPair_=3;
        public static float fitnessShift_=0;
        public static float fitnessMultiplier_=0;
        public static float CDSfitnessShift_=0;
        public static float CDSfitnessMultiplier_=0;

        //set to true if max_size is set
        public static boolean truncate_=false;

        //Population of Individuals
        private static Vector <Individual> population_;
```

```
//Population fitness
private static float pFitness_=0.0f;

public static void runSimulation() {

        //Comparator used to sort population Vector
        IndividualComparator pc=new IndividualComparator();

        //Create vector to hold population
        population_=new Vector<Individual>(pSize_);

        if (verbose_){
                Clock.startClock();
                System.out.print("Creating first individual: ");
        }

        Individual i=Creator.getIndividual();
        if (verbose_){
                System.out.println(Clock.getStepTime());
                System.out.print("Initializing population vector: ");
        }

        //Create a population of identical individuals
        for (int pi=0; pi < pSize_; pi++) {
                population_.add(pi, new Individual(i));
        }
        if (verbose_)
                System.out.println(Clock.getStepTime());

        //Holds offspring, population_ will reference this at
        // the end of a generation
        Vector <Individual> nextGeneration;

        int ovOff=0;
        int ovPair=0;

        for (int gen=1; gen <=nGen_; gen++) {

                float meanOffspring=0.0f;
                Clock.startClock2();
                if (verbose_)
                        System.out.println("[Generation: "+gen+"]");

                nextGeneration=new Vector<Individual>();

                if (verbose_)
                        System.out.print("Mutating population: ");

                for (int pi=0; pi < pSize_; pi++){
                        population_.
                                elementAt(pi).randomMutation(mutPerGen_);
                }
                if (verbose_)
                System.out.println(Clock.getStepTime());

                int pSizeNew=0;

                if (verbose_)
                        System.out.print("Creating offspring: ");

                for (int pi=0; pSizeNew < pSize_ ||
                        pi < (pSize_/2);++pi) {

                        int mIdx;
                        int pIdx;
                        for (;;){
                                mIdx=Stats.nextInt(pSize_);
                                if (population_.elementAt(mIdx).getGender()
                                                ==Consts.FEMALE)
                                        break;
```

```
        }
        for (;;){
                pIdx=Stats.nextInt(pSize_);
                if (population_.elementAt(pIdx).getGender()
                                ==Consts.MALE)
                        break;
        }

        ovPair++;
        int n_offspring;


        if (fertilityFitnessDependence_ & gen>1){
                n_offspring=Math.round(
                        new Float(
                        (population_.elementAt(mIdx).
                                getFitnessPercentileRank()/2
                         +population_.elementAt(pIdx).
                                getFitnessPercentileRank()/2
                         )
                         *2
                         *Stats.poisson(offspringPerMatingPair_)
                        )
                );
        } else{
                n_offspring=
                        Stats.poisson(offspringPerMatingPair_);
        }
        if (n_offspring < 0) n_offspring=0;

        meanOffspring+=n_offspring;
        ovOff+=n_offspring;
        int no=Stats.poisson(n_offspring);

        for (int x=0; x < no;++x) {
                Individual offspring=new Individual(
                        population_.elementAt(mIdx).getGamete() ,
                        population_.elementAt(pIdx).getGamete());
                nextGeneration.add(offspring);
                pSizeNew++;
        }
}
if (verbose_){
        System.out.println("Mean # offspring : "
                        +meanOffspring/(pSize_/2));
        System.out.println("Overall Mean # offspring : "
                        +ovOff/(double)ovPair);
        System.out.println(Clock.getStepTime());
}
population_.removeAllElements();
population_=nextGeneration;
nextGeneration=null;

if (verbose_){
        System.out.print ("Sorting offspring by fitness: ");
}
Collections.sort(population_ , pc);
if (verbose_){
        System.out.println(Clock.getStepTime());
        System.out.print
                ("Truncating offspring population: ");
}

population_.setSize(pSize_);
population_.trimToSize();

int rank=pSize_;
for(int x=0; pSizeNew < pSize_ || x < pSize_ ; x++){
        population_.elementAt(x).setFitnessPercentileRank(
                        new Float(new Float(rank-0.5)/pSize_));
        rank--;
```

```
                }
                if (verbose_){
                        System.out.println(Clock.getStepTime());
                        System.out.print
                                ("Calculating population fitness: ");
                }
                pFitness_=0.0f;

                for (int pi=0; pi < pSize_;++pi){
                        pFitness_+=population_.elementAt(pi).getFitness();
                }

                pFitness_/=pSize_;

                if (verbose_)
                        System.out.println(Clock.getStepTime());

                long totalStepTime=Clock.getStepTime2();

                if (verbose_)
                        System.out.println(
                                        "Total time for one generation (s): "
                                        +(totalStepTime/1000));

                long d=totalStepTime * (nGen_-gen)/1000/60/60/24;
                long h=totalStepTime * (nGen_-gen)/1000/60/60 % 24;
                long m=totalStepTime * (nGen_-gen)/1000/60 % 60 ;
                long s=totalStepTime * (nGen_-gen)/1000 % 60;

                if (verbose_)
                        System.out.println("Estimated time for "+(nGen_
                                        - gen)+" generations: "+d+"d "+h+"h "+
                                        m+"m "+s+"s");
                System.out.println ((gen)+","+pFitness_);
        }
        System.out.println("Overall Mean # offspring : "
                        +ovOff/(double)ovPair);
}

public static void readConfigFile(final String fileName)
{
        StringTokenizer st=new StringTokenizer("");
        Hashtable<String, String> config=
                new Hashtable<String, String>();
        try {
                BufferedReader in=
                        new BufferedReader(new FileReader(fileName));
                while(in.ready())
                {
                        String line=new String();
                        line=in.readLine();
                        st=new StringTokenizer(line);

                        String key=st.nextToken();
                        String value=st.nextToken();

                        config.put(key,value);

                        System.out.println(key+" "+value);
                }
        } catch (IOException ioe){
                System.exit(0);
        }
        try {
                maxSize_=
                        (new Integer(config.get("max_size"))).intValue();
                truncate_=true;
        } catch (Exception e){
                maxSize_=Consts.MAX_INT;
        }
        try {
```

```
                codonFileName_=config.get("codon_filename");
        } catch (Exception e){
        }
        try {
                verbose_=new Boolean(config.get("verbose"));
        } catch (Exception e){
                verbose_=false;
        }
        try {
                dominanceFilename_=config.get("dominance_filename");
        } catch (Exception e){
        }
        try {
                fertilityFitnessDependence_=
                        new Boolean(
                                        config.get("fertility_fitness_dependence"));
        } catch (Exception e){
                fertilityFitnessDependence_=false;
        }

        try {
                mutPerGen_=
                        (new Integer(
                                config.get(
                                        "mutations_per_generation"))).intValue();
                pSize_=
                        (new Integer(
                                config.get("population_size"))).intValue();
                nGen_=
                        (new Integer(
                                        config.get("num_generations"))).intValue();
                exonFileName_=config.get("exon_filename");
                chrFileName_=config.get("chromosome_filename");
                MRIFileName_=config.get("MRI_filename");
                offspringPerMatingPair_=
                        (new Integer(
                                config.get(
                                                "offspring_per_mating_pair"))
                                                        ).intValue();
                MRIConstant_=(
                        new Float(
                                config.get( "MRI_constant"))).floatValue();
                mutationFileName_=
                        config.get("mutation_filename");
                crossoverFileName_=config.get("crossover_filename");
                crossoverMultiplier_=
                        (new Integer(
                                        config.get(
                                                "crossover_multiplier"))).intValue();
                maxBlockSize_=(
                        new Integer(
                                        config.get("max_block_size"))).intValue();
                fitProbFileName_=config.get("fitness_filename");
                CDSfitProbFileName_=config.get("CDS_fitness_filename");
                fitnessShift_=
                        (new Float(config.get("fitness_shift")));
                CDSfitnessShift_=
                        (new Float(config.get("CDS_fitness_shift")));
                fitnessMultiplier_=
                        (new Float(config.get("fitness_multiplier")));
                CDSfitnessMultiplier_=
                        (new Float(config.get("fitness_multiplier")));

        } catch (Exception e){
                System.out.println("Error reading config file");
        }
}

public static void main (String args[]){
        if (args.length > 0){
                readConfigFile(args[0]);
```

```
                        Stats.init();
                        Dominance.init(dominanceFilename_);
                        if (truncate_){
                                Creator.init (chrFileName_, maxSize_);
                        } else {
                                Creator.init (chrFileName_);
                        }
                        Substitution.init(mutationFileName_);
                        Creator.readExons(exonFileName_, Creator.getChrSize());
                        Creator.readMRI(MRIFileName_, Creator.getChrSize());
                        Creator.createIndividual(maxBlockSize_);
                        Crossover.setMultiplier(crossoverMultiplier_);
                        Crossover.init(crossoverFileName_, Creator.getChrSize());
                        FitnessCalc.init(Creator.getStrChr(), fitProbFileName_,
                                        fitnessShift_, fitnessMultiplier_,
                                        CDSfitProbFileName_, CDSfitnessShift_,
                                        CDSfitnessMultiplier_, MRIConstant_);
                        runSimulation();
                        return;
                } else {
                        System.out.println("Usage: Gem <configfile.in>");
                }
        }
}
```

IndividualComparator (default package)

```
// IndividualComparator.java, default package
// Copyright 2010 Andrew McSweeny
// All rights reserved

import java.util.Comparator;
import individual.Individual;

class IndividualComparator implements Comparator <Individual> {

	public int compare ( Individual i1, Individual i2 ){
		float fitness1 = i1.getFitness();
		float fitness2 = i2.getFitness();

		if ( fitness1 < fitness2 ){
			return 1;
		} else if ( fitness1 > fitness2 ){
			return -1;
		} else {
			return 0;
		}
	}
}
```

bitpack.BitPack

```
// BitPack.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package bitpack;

import java.util.TreeSet;

public class BitPack {

	public static byte integerToByte( int i){
		if ( i > 127){
			i-=256;
			return (byte)i;
		} else {
			return (byte)i;
		}
	}

	public static byte integerToByte( double i){
		if ( i > 127){
			i-=256;
			return (byte)i;
		} else {
			return (byte)i;
		}
	}

	/**
	 * Performs crossover between two PackedBlocks of nucleotide
	 * data
	 *
	 * Crossover occurs AFTER position, so position will never be
	 * the last character in the sequence
	 *
	 * @param pb1
	 * @param pb2
	 * @param position
	 * @return
	 */
	public static PackedBlock CrossBlocks (PackedBlock pb1,
			PackedBlock pb2,
				int position){

		if (position == 0) return pb2;
		int numBytes = pb1.getNumBytes();

		byte [] bytes1 = pb1.getBytes();
		byte [] bytes2 = pb2.getBytes();

		byte [] bytes3 = new byte [numBytes];

		// this conditional handles situations where PackedBlocks
		// can be split by  whole bytes
		// e.g. position == 7,        crossover occurs right after end of
		//                                          2nd block
		//                                          so copy first two blocks
		//                                          then copy the rest of them
		if (position % 4 == 3 ){

			// block where crossover occurs
			int crossBlockIdx = position / 4 + 1;

			// copy bytes from 1st PackedBlock into the result block
			// e.g. crossBlock is at pos 2 (3rd byte)
			//             copy first 2 bytes
			System.arraycopy(bytes1, 0, bytes3, 0, crossBlockIdx);
```

```
        // copy last bytes 2nd PackedBlock into the result block

        // e.g. crossBlock is at pos 2 (3rd byte)
        //      start at pos 2
        //             total number of bytes is 8 (last index is 7)
        //             from pos 2 to pos 7 (2,3,4,5,6,7) is 6 bytes
        // (numBytes .8.-2)

        System.arraycopy(bytes2, crossBlockIdx,
                        bytes3, crossBlockIdx,
                        numBytes-crossBlockIdx);
        return new PackedBlock(pb1,bytes3);
} else {
        // position within crossover block after where crossover
        // occurs
        // 3 , 2 , 1  ... will never be 0
        int bitPairSignificance = 3-(position % 4);

        // copy bytes from 1st PackedBlock into the result block
        // e.g. crossBlock is at pos 2 (3rd byte)
        //             copy first 2 bytes
        int crossBlockIdx = position / 4;
        // actually the index of the byte where the crossover is

        System.arraycopy(bytes1, 0, bytes3, 0, crossBlockIdx);

        // mask 1 will be &'d with the first half
        // mask 2 will be &'d with the second half
        // results will be added

        byte mask1 = 0;
        byte mask2 = 0;

        // array of masks 00000011 - 11000000
        byte [] masks = new byte [4];

        for (int bps = 0; bps < 4; bps++){
                // 3 = 11 binary
                masks[bps] = BitPack.integerToByte(
                                Math.pow(4, bps)*3);
        }

        // position within crossover block after where
        // crossover occurs
        // 3 , 2 , 1  ... will never be 0
        switch (bitPairSignificance){
                case 3:
                        //128 64 32 16 8 4 2 1
                        // 7    6  5    4 3 2 1 0

                        // mask1 = 11000000           =       192
                        // mask2 = 00111111
                        mask1 = masks[3];
                        mask2 = (byte) (masks[2]
                                + masks[1] + masks[0]);
                        break;
                case 2:
                        // mask1 = 11110000
                        // mask2 = 00001111
                        mask1 = (byte) (masks[3] + masks[2]);
                        mask2 = (byte) (masks[1] + masks[0]);
                        break;
                case 1:
                        // mask1 = 11111100
                        // mask2 = 00000011
                        mask1 = (byte) (masks[3]
                        + masks[2] + masks[1]);
                        mask2 = (byte) (masks[0]);
                        break;
        }
```

```
                        // result is the block with proper crossover

                        // copy last bytes 2nd PackedBlock into the result block
                        // e.g. crossBlock is at pos 2 (3rd byte)
                        //      start at pos 3
                        //             total number of bytes is 8 (last index is 7)
                        //             from pos 3 to pos 7 is 5 bytes (numBytes-3)
                        bytes3[crossBlockIdx] = (byte)(
                                       (mask1&bytes1[crossBlockIdx])
                                       + (mask2&bytes2[crossBlockIdx]) );

                        System.arraycopy(bytes2, crossBlockIdx+1,
                                                    bytes3, crossBlockIdx+1,
                                                    numBytes-(crossBlockIdx+1));

                        PackedBlock pb3 = new PackedBlock(pb1,bytes3);
                        pb3.mutationIndices_ =
                                new TreeSet<Integer>(
                                       pb1.mutationIndices_.headSet(position, true));
                        pb3.mutationIndices_.
                           addAll(
                                pb2.mutationIndices_.tailSet(position, false));
                        return pb3;
                }
        }
}
```

bitpack.PackedBlock

```
// PackedBlock.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package bitpack;

import java.util.TreeSet;

import dominance.DomStruct;
import dominance.Dominance;

import substitution.Substitution;
import tools.Consts;

import engine.Stats;
import fitness.FitnessCalc;
import bitpack.BitPack;

public class PackedBlock {

        public TreeSet<Integer> mutationIndices_;

        public byte [] bytes_;

        private int startIdx_;
        private int endIdx_;
        private boolean isCDS_;

        private float minCoefficient_;
        private float maxCoefficient_;
        private float meanCoefficient_;


        public float getMinCoefficient() {
                return minCoefficient_;
        }

        public void setMinCoefficient(float minCoefficient) {
                this.minCoefficient_ = minCoefficient;
        }

        public float getMaxCoefficient() {
                return maxCoefficient_;
        }

        public void setMaxCoefficient(float maxCoefficient) {
                this.maxCoefficient_ = maxCoefficient;
        }

        public float getMeanCoefficient() {
                return meanCoefficient_;
        }

        public void setMeanCoefficient_(float meanCoefficient) {
                this.meanCoefficient_ = meanCoefficient;
        }


        private int numLetters_;
        private int numBytes_;

        private float fitness_ = 0.0f;

        private int blockNumber_;

        public void setStartIdx(int startIdx) {
                this.startIdx_ = startIdx;
```

```
}

public void setEndIdx(int endIdx) {
	this.endIdx_ = endIdx;
}

public void setFitness(float fitness){
	this.fitness_ = fitness;
}

public int getNumBytes() {
	return numBytes_;
}

public int getEndIdx() {
	return endIdx_;
}

public int getBlockNumber() {
	return blockNumber_;
}

public int getNumLetters(){
	return numLetters_;
}

public byte [] getBytes(){
	return bytes_;
}

/**
 *  Prints sequence of PackedBlock using getLetter() function
 *  Useful to confirm that getLetter() function works
 *
 */
public void printSequence2(){
	for (int x = 0;        x < numLetters_; ++x){
		System.out.print(this.getLetter(x));
	}
}

public int getStartIdx() {
	return startIdx_;
}


/**
 *  Prints sequence of PackedBlock without using getLetter()
 *  method
 *  Useful for comparison
 */
public void printSequence(){

	// Bit pair Significance
	// Most significant pair    first : 3
	//                                                  second : 2
	//                                                   third : 1

	//
	// for 11, 3 * 4^3 = 192 = 11000000
	//         3 * 4^2 = 48  = 00110000
	//         3 * 4^1 = 12  = 00001100
	//         3 * 4^0 = 3   = 00000011

	// for 10, 2 * 4^3 = 128 = 10000000
	// etc

	int bitPairSignificance = 3;
	int byteIdx = 0;
	for (int x = 0;        x < numLetters_; ++x){
```

```
                if ( ((byte) BitPack.integerToByte(
                                Math.pow(4,bitPairSignificance)*Consts.G) &
                                        bytes_[byteIdx])
                                        == BitPack.integerToByte(
                                                Math.pow(
                                                        4,bitPairSignificance)
                                                        *Consts.G )){
                        System.out.print("G");
                } else if ( ((byte) BitPack.integerToByte(
                                Math.pow(4,bitPairSignificance)*Consts.C) &
                                bytes_[byteIdx])
                                == BitPack.integerToByte(
                                        Math.pow(4,bitPairSignificance)*Consts.C )){
                        System.out.print("C");
                } else if ( ((byte) BitPack.integerToByte(
                                Math.pow(4,bitPairSignificance)*Consts.T) &
                                bytes_[byteIdx])
                                == BitPack.integerToByte(
                                        Math.pow(4,bitPairSignificance)*Consts.T )){
                        System.out.print("T");

                } else {
                        System.out.print("A");
                }

                bitPairSignificance--;
                if ( bitPairSignificance == -1 ){
                        bitPairSignificance = 3;
                        byteIdx ++;
                }
        }
}

/**
 *
 * returns a character from the binary data in the PackedBlock
 *
 * @param position            0-based position of the character in the
 *                                      block
 * @return                          character at this position
 */
public char getLetter(int position){
        // convert 0-based index of letter in sequence to index
        // for a byte
        int byteIdx = position / 4;

        // i.e. if position is 5, byte 5/4=0, 5%4=1,
        // significance = 3-1=2

        // 3 2 1 0  3 2 1 0
        // 00000000 00110000

        // 0 1 2 3  4 5 6 7
        // pos

        int bitPairSignificance = 3 - (position % 4);

        if ( ((byte) BitPack.integerToByte(
                Math.pow(4,bitPairSignificance)*Consts.G) &
                        bytes_[byteIdx]) == BitPack.integerToByte(
                                Math.pow(4,bitPairSignificance)*Consts.G )){
                return ('G');
        } else if (((byte)BitPack.integerToByte(
                Math.pow(4,bitPairSignificance)*Consts.C) &
                        bytes_[byteIdx])== BitPack.integerToByte(
                                Math.pow(4,bitPairSignificance)*Consts.C )){
                return ('C');
        } else if ( ((byte) BitPack.integerToByte(
                Math.pow(4,bitPairSignificance)*Consts.T) &
                        bytes_[byteIdx]) == BitPack.integerToByte(
                                Math.pow(4,bitPairSignificance)*Consts.T )){
```

```
                return ('T');
        } else {
                return ('A');
        }
}

/**
 * Computes fitness for the entire PackedBlock using the
 * static context
 *
 * FitnessCalc class
 */
public void computeFitness(){
        fitness_ = FitnessCalc.getFitness(this);
}

public float getFitness(){
        return fitness_;
}

public void randomMutation (int numMutations){
        for (int x = 0; x < numMutations; x++){
                randomMutation();
        }
}

public void randomMutation (){
        String s = Substitution.getPair();

        int position;
        int bitPair;

        int c=0;
        char chr;

        while(true) {
                position = Stats.nextInt(numLetters_);
                chr = this.getLetter(position);
                if (s.charAt(0) == chr){
                        break;
                }
                c++;
                if (c > 100) {
                        // starting base not found
                        // get new pair
                        s = Substitution.getPair();
                        c = 0;
                }
        }

        this.mutationIndices_.add(position);

        if (s.charAt(1) == 'A'){
                bitPair = Consts.A;
        } else if (s.charAt(1) == 'C'){
                bitPair = Consts.C;
        } else if (s.charAt(1) == 'T'){
                bitPair = Consts.T;
        } else {// s.charAt(1) == 'G'
                bitPair = Consts.G;
        }

        int byteIdx = position / 4;
        int bitPairSignificance = 3 - (position % 4);

        // i.e. if position is 5, byte 5/4=0, 5%4=1,
        // significance = 3-1=2
        // 3 2 1 0  3*2*1 0
        // 00000000 00110000

        // 0 1 2 3  4 5 6 7
```

```
        // position

        byte bMask = -1;

        bMask-=((byte)BitPack.integerToByte(Math.pow(4,
                        bitPairSignificance)*3));

        /**
         * Determines which block a character is found in
         */

        float oldfitness = FitnessCalc.getFitnessAt(this, position);
        bytes_[byteIdx] &= bMask;
        bytes_[byteIdx] += BitPack.integerToByte(
                        Math.pow(4, bitPairSignificance)*bitPair);
        float newfitness = FitnessCalc.getFitnessAt(this, position);

        fitness_ -= oldfitness;
        fitness_ += newfitness;
}

public PackedBlock(PackedBlock pb, byte [] pbData){
        this.isCDS_ = pb.isCDS_;
        this.maxCoefficient_ =pb.getMaxCoefficient();
        this.minCoefficient_ = pb.getMinCoefficient();
        this.meanCoefficient_ = pb.getMeanCoefficient();
        this.startIdx_ = pb.startIdx_;
        this.endIdx_ = pb.endIdx_;
        numLetters_ = pb.numLetters_;
        numBytes_ = pb.numBytes_;
        this.bytes_ = pbData;
        this.fitness_ = pb.fitness_;
        this.blockNumber_ = pb.blockNumber_;
        this.mutationIndices_ = new TreeSet<Integer>();
}

public PackedBlock(PackedBlock pb){
        this.isCDS_ = pb.isCDS_;
        this.maxCoefficient_ =pb.getMaxCoefficient();
        this.minCoefficient_ = pb.getMinCoefficient();
        this.meanCoefficient_ = pb.getMeanCoefficient();
        this.startIdx_ = pb.startIdx_;
        this.endIdx_ = pb.endIdx_;
        numLetters_ = pb.numLetters_;
        numBytes_ = pb.numBytes_;
        this.bytes_ = new byte[numBytes_];
        System.arraycopy(pb.bytes_,0,this.bytes_,0,numBytes_);
        this.fitness_ = pb.fitness_;
        this.blockNumber_ = pb.blockNumber_;
        this.mutationIndices_ =
                new TreeSet<Integer>(pb.mutationIndices_);
}

public PackedBlock (String sequence, int blockNumber,
                int startIdx,boolean isCDS, int ID){
        // this constructor should only be used to create blocks
        // from original sequence
        this.isCDS_ = isCDS;

        if (isCDS){
                DomStruct dr = Dominance.getRandomDominance();
                this.maxCoefficient_ = dr.getMaxCoeff();
                this.minCoefficient_ = dr.getMinCoeff();
                this.meanCoefficient_ = dr.getMeanCoeff();
        } else {
                this.maxCoefficient_ = 0;
                this.minCoefficient_ = 0;
                this.meanCoefficient_ = 1;
        }

        this.mutationIndices_ = new TreeSet<Integer>();
```

```
                this.fitness_ = 0.0f;
                this.blockNumber_ = blockNumber;
                this.startIdx_ = startIdx;

                numLetters_ = sequence.length();

                numBytes_ = numLetters_ / 4;

                if ( (numLetters_ % 4) != 0){
                        numBytes_++;
                }
                bytes_ = new byte[numBytes_];

                bytes_[0] = 0; // zero out the first byte
                int bitPairSignificance = 3;
                int packIdx = 0;

                for (int i = 0; i < numLetters_; ++i){
                        if ( Character.toUpperCase(sequence.charAt(i)) == 'C'){
                                bytes_[packIdx] +=
                                        BitPack.integerToByte(
                                                Math.pow(4,bitPairSignificance)*Consts.C);
                        } else if ( Character.toUpperCase(sequence.charAt(i))
                                        == 'T'){
                                bytes_[packIdx] +=
                                        BitPack.integerToByte(
                                                Math.pow(4,bitPairSignificance)*Consts.T);
                        } else if ( Character.toUpperCase(sequence.charAt(i))
                                        == 'G'){
                                bytes_[packIdx] +=
                                        BitPack.integerToByte(
                                                Math.pow(4,bitPairSignificance)*Consts.G);
                        }
                        bitPairSignificance--;
                        if ( bitPairSignificance == -1 ){
                                bitPairSignificance = 3;
                                if ( packIdx+1 < numBytes_ ){
                                        packIdx ++;
                                        bytes_[packIdx] = 0;
                                }
                        }
                }
        }
}
```

bitpack.PackedChromosome

```
// PackedChromosome.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package bitpack;

import individual.Creator;
import java.util.Vector;
import engine.Stats;


public class PackedChromosome {

	private int numLetters_;
	private int numBlocks_ = 0;

	public void setNumBlocks(int numBlocks) {
		this.numBlocks_ = numBlocks;
	}

	public Vector <PackedBlock> blocks_;

	/* @param index		index of character in chromosome
	 * @return		block number containing the character
	 */
	static int blockNumber(int index){
		return index;
	}


	public void randomMutation (){

		/**
		 * Determines which block a character is found in
		 */
		int position = Stats.nextInt(numLetters_);
		int bn = Creator.getPosBlock(position);

		PackedBlock pb = new PackedBlock(this.blocks_.elementAt(0));
		try {
			pb = new PackedBlock(this.blocks_.elementAt(bn));
		} catch ( Exception e){

		}
		pb.randomMutation();
		blocks_.setElementAt(null, bn);
		blocks_.setElementAt(pb, bn);
	}

	public int getNumLetters(){
		return this.numLetters_;
	}

	public PackedChromosome (PackedChromosome pc)
	{
		this.numLetters_ = pc.numLetters_;
		this.numBlocks_ = pc.numBlocks_;

		blocks_ = new Vector<PackedBlock>();
		for (int i = 0; i < pc.numBlocks_; ++i){//
			this.addBlock(pc.getBlock(i));
		}
	}

	public Vector<PackedBlock> getBlocks(){
		return blocks_;
```

```
        }

        int getNumBlocksAdded(){
                return blocks_.size();
        }

        public PackedChromosome( int numLetters ){
                // Constructor doesn't add blocks, only creates an empty PC
                this.numLetters_ = numLetters;
                blocks_ = new Vector<PackedBlock>();
        }

        public void addBlock(PackedBlock pb){
                blocks_.add(pb);
        }

        public PackedBlock getBlock(int idx){
                return blocks_.elementAt(idx);
        }

        public void setBlock(PackedBlock pb, int idx){
                blocks_.setElementAt(null, idx);
                blocks_.setElementAt(pb, idx);
        }

        public int getNumBlocks() {
                return numBlocks_;
        }
}
```

dominance.Dominance

```
// Dominance.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package dominance;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Iterator;
import java.util.Vector;

import engine.Stats;

public class Dominance {

	static Vector<DomStruct> dominanceProbabilities_;

	public static void init(String filename){
		dominanceProbabilities_ = new Vector<DomStruct>();
		try {
			BufferedReader in =
				new BufferedReader(new FileReader(filename));
			String line;

			while(in.ready()){

				line = in.readLine();
				if (line.charAt(0) == '#')
					continue;
				String [] tokens = line.split("\t");

				float probability =
					(new Float(tokens[0])).floatValue();
				float mean = (new Float(tokens[1])).floatValue();
				float max = (new Float(tokens[2])).floatValue();
				float min = (new Float(tokens[3])).floatValue();

				DomStruct dp =
					new DomStruct(probability, mean, max, min);
				dominanceProbabilities_.add(dp);

			}
		} catch (IOException e) {
			// TODO Auto-generated catch block
			// TODO find sandwich
			e.printStackTrace();
		}
	}

	public static DomStruct getRandomDominance(){
		float prob = Stats.nextFloat(); // cumulative probability
		Iterator<DomStruct> i = dominanceProbabilities_.iterator();
		float cumProb = 0.0f;

		DomStruct d = null;
		while (i.hasNext()){
			d = i.next();
			cumProb += d.getProb();
			if (cumProb >= prob) {
				break;
			}
		}
		return d;
	}
}
```

dominance.DomStruct

```
// Domstruct.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package dominance;

public class DomStruct {

	public float getMaxCoeff() {
		return maxCoeff_;
	}
	public float getMinCoeff() {
		return minCoeff_;
	}
	public float getMeanCoeff() {
		return meanCoeff_;
	}
	public float getProb() {
		return prob_;
	}
	DomStruct( float frequency, float mean, float max, float min){
		prob_ = frequency;
		meanCoeff_ = mean;
		maxCoeff_ = max;
		minCoeff_ = min;
	}
	float prob_;
	float maxCoeff_;
	float minCoeff_;
	float meanCoeff_;
}
```

engine.Crossover

```
// Crossover.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package engine;

import individual.Creator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;

import tools.Sequence;

public class Crossover {

	public static boolean truncate_=false;

	private static double maxCm_=0.0;
	private static int lastIdx_=0;

	private static int [] intKeys_;
	private static double [] doubleValues_;

	private static int [] lookupKey_;

	private static int multiplier_=1;

	public static void setMultiplier(int multiplier) {
		Crossover.multiplier_=multiplier;
	}

	public static int getNumCrossovers(){
		return Stats.poisson(maxCm_/100*multiplier_);
	}

	public static int getRandomCrossover(){

		double randCm=Stats.nextDouble() * maxCm_;

		int guessIdxHigh=lastIdx_;
		int guessIdxLow=0;
		int guessIdx;

		int ctr=0;
		while(true){
			ctr ++;
			if(ctr > 100000){
				System.out.println("stuck in infinite loop");
			}
			guessIdx=guessIdxLow +(guessIdxHigh-guessIdxLow) / 2;
			if(doubleValues_[guessIdx] <= randCm &&
					doubleValues_[guessIdx+1] >= randCm){
				break;
			}
			if(doubleValues_[guessIdx] < randCm){
				// guess was low
				guessIdxLow=guessIdx;
			}
			if(guessIdx > 0 && doubleValues_[guessIdx] > randCm){
				// guess was high
				guessIdxHigh=guessIdx;
			}
		}
```

```
        // get info from guess idx
        double sCM=doubleValues_[guessIdx];
        double eCM=doubleValues_[guessIdx+1];
        int sBp=intKeys_[guessIdx];
        int eBp=intKeys_[guessIdx+1];

        Float fltBpAdded=
                new Float((randCm-sCM)*(eBp-sBp) /(eCM-sCM));

        int bpAdded=Math.round(fltBpAdded);
        if(sBp +bpAdded > 300000){
                int i=0;
                i++;
        }
        int pos=sBp + bpAdded;
        if(pos < Creator.getChrSize())
                return pos;
        else
                return Creator.getChrSize()-1;
}


public static void init(String crossOverFileName, int genomeSize){

        try {
                BufferedReader in=
                        new BufferedReader(new FileReader(crossOverFileName));
                String line;
                int linesRead=0;

                HashMap <Integer, Double> hm=new HashMap<Integer, Double>();

                while(in.ready()){
                        line=in.readLine();
                        linesRead++;
                        if(linesRead == 1){
                                continue;
                        }

                        String [] tokens=line.split("\t");

                        int bp=new Integer(tokens[0]).intValue();
                        double cM=new Double(tokens[1]).doubleValue();

                        hm.put(bp, cM);
                        if(bp > genomeSize){
                                break;
                        }
                }

                ArrayList <Integer> keys=new ArrayList<Integer>(hm.keySet());
                Collections.sort(keys);
                Iterator <Integer> it=keys.iterator();

                intKeys_ =new int[keys.size()];
                doubleValues_=new double[keys.size()];

                int i=0;
                for(i=0; it.hasNext(); i++){
                        intKeys_[i] =(it.next()).intValue();
                        doubleValues_[i]=hm.get(intKeys_[i]).doubleValue();
                }
                int lastIdx=i-1;
                lastIdx_=lastIdx;
                int lastKey=intKeys_[lastIdx];

                int realEnd=genomeSize;
                double realCmDiff=
                        (lastKey-realEnd)*
                        (doubleValues_[lastIdx]-doubleValues_[lastIdx-1])
```

```
                                        /(intKeys_[lastIdx]-intKeys_[lastIdx-1]);

                        doubleValues_[lastIdx] -= realCmDiff;

                        maxCm_=doubleValues_[lastIdx];
                        intKeys_[lastIdx]= realEnd;
                        lookupKey_=new int[genomeSize];

                        int k=0;
                        for(i=1; i <= genomeSize; i++){
                                lookupKey_[Sequence.bpToIdx(i)]=k;
                                if(i<genomeSize && i >= intKeys_[k+1]-1){
                                        k++;
                                }
                        }
                        k=0;
                } catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

engine.Stats

```
// Stats.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package engine;

import java.util.Random;

public class Stats {
        static Random r;
        public static void init(){
                r = new Random();
        }

        public static int nextInt(){
                return r.nextInt();
        }

        public static int nextInt(int n){
                return r.nextInt(n);
        }

        public static double nextDouble(){
                return r.nextDouble();
        }

        public static float nextFloat(){
                return r.nextFloat();
        }

    public static int poisson(double c) {
        // c is the intensity (lambda)
        int x = 0;
        double t = 0.0;
        for (;;) {
                t -= Math.log(r.nextDouble())/c;
                if (t > 1.0)
                        return x;
                x++;
        }
    }
}
```

fitness.FitnessCalc

```
// FitnessCalc.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package fitness;

import engine.Stats;
import individual.Creator;
import java.util.Iterator;
import java.util.Vector;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.lang.Character;
import java.lang.Math;

import tools.Consts;

import bitpack.PackedBlock;

public class FitnessCalc {

        public static float Afits_[];
        public static float Cfits_[];
        public static float Tfits_[];
        public static float Gfits_[];

        public static float fractionBeneficial_;
        public static float MRIConstant_;


        public static Vector<FitStruct> fitnessProbabilities_;
        public static Vector<FitStruct> CDSfitnessProbabilities_;

        /**
         * Initializes FitnessCalc static object
         * no longer takes blocksize as a parameter as
         * block starts and finishes will somehow be used to choose
         * the correct block and offset
         *
         * do we need an individual?  no, get it from Creator.
         *
         * Creates a matrix of mutation fitness values according to
         * the fraction beneficial mutations
         *
         * @param sequence
         * @param fracben
         */

        public static void readCDSProbabilities(String filename){
                CDSfitnessProbabilities_ = new Vector<FitStruct>();
                try {
                        BufferedReader in =
                                new BufferedReader(new FileReader(filename));
                        String line;

                        while(in.ready()){

                                line = in.readLine();
                                String [] tokens = line.split("\t");

                                float fitness = (new Float(tokens[0])).floatValue();
                                float probability =
                                        (new Float(tokens[1])).floatValue();
                                FitStruct fp = new FitStruct(fitness,probability);
                                CDSfitnessProbabilities_.add(fp);
                        }
```

```
        } catch (IOException e) {
                e.printStackTrace();
        }
}

public static void readProbabilities(String filename){
        fitnessProbabilities_ = new Vector<FitStruct>();
        try {
                BufferedReader in =
                        new BufferedReader(new FileReader(filename));
                String line;

                while(in.ready()){
                        line = in.readLine();
                        String [] tokens = line.split("\t");

                        float fitness = (new Float(tokens[0])).floatValue();
                        float probability =
                                (new Float(tokens[1])).floatValue();

                        FitStruct fp = new FitStruct(fitness,probability);

                        fitnessProbabilities_.add(fp);
                }
        } catch (IOException e) {
                e.printStackTrace();
        }
}

public static float getRandomCDSFitness(){

        float prob = Stats.nextFloat(); // cumulative probability

        Iterator<FitStruct> i = CDSfitnessProbabilities_.iterator();

        float cumProb = 0.0f;

        FitStruct f = null;
        while (i.hasNext()){
                f = i.next();
                cumProb += f.getProbability();
                if (cumProb >= prob) {
                        break;
                }
        }
        return f.fitness_;
}


public static float getRandomFitness(){

        float prob = Stats.nextFloat(); // cumulative probability

        Iterator<FitStruct> i = fitnessProbabilities_.iterator();

        float cumProb = 0.0f;

        FitStruct f = null;
        while (i.hasNext()){
                f = i.next();
                cumProb += f.getProbability();
                if (cumProb >= prob) {
                        break;
                }
        }
        return f.fitness_;
}

public static void init (String sequence,
                String filename, float fitnessShift,
                float fitnessMultiplier, String CDSfilename,
```

```
                float CDSfitnessShift, float CDSfitnessMultiplier,
                float MRIConstant){

        MRIConstant_ = MRIConstant;
        System.out.print ("Initializing fitness table [");
        readProbabilities(filename);
        readCDSProbabilities(CDSfilename);
        int numFits = sequence.length();

        Afits_ = new float[numFits];
        Cfits_ = new float[numFits];
        Tfits_ = new float[numFits];
        Gfits_ = new float[numFits];

        for ( int x = 0; x < numFits; x++){
                char seqChar = Character.toUpperCase(sequence.charAt(x));

                if ( x % (numFits/10) == 0 ) System.out.print(".");

                if (Creator.isExonic_[x]){
                        Afits_[x] =
                                getRandomCDSFitness()*CDSfitnessMultiplier
                                + CDSfitnessShift;
                        Cfits_[x] =
                                getRandomCDSFitness()*CDSfitnessMultiplier
                                + CDSfitnessShift;
                        Tfits_[x] =
                                getRandomCDSFitness()*CDSfitnessMultiplier
                                + CDSfitnessShift;
                        Gfits_[x] =
                                getRandomCDSFitness()*CDSfitnessMultiplier
                                + CDSfitnessShift;
                } else {
                        Afits_[x] = getRandomFitness()*fitnessMultiplier
                                + fitnessShift;
                        Cfits_[x] = getRandomFitness()*fitnessMultiplier
                                + fitnessShift;
                        Tfits_[x] = getRandomFitness()*fitnessMultiplier
                                + fitnessShift;
                        Gfits_[x] = getRandomFitness()*fitnessMultiplier
                                + fitnessShift;
                }
                if (seqChar == 'A') {
                        Afits_[x] = 0;
                        Cfits_[x] += Creator.MRI_[x]*MRIConstant_;
                        Gfits_[x] += Creator.MRI_[x]*MRIConstant_;
                } else if (seqChar == 'C') {
                        Cfits_[x] = 0;
                        Tfits_[x] -= Creator.MRI_[x]*MRIConstant_;
                        Afits_[x] -= Creator.MRI_[x]*MRIConstant_;
                } else if (seqChar == 'T') {
                        Tfits_[x] = 0;
                        Cfits_[x] += Creator.MRI_[x]*MRIConstant_;
                        Gfits_[x] += Creator.MRI_[x]*MRIConstant_;
                } else {                // seqChar == 'G'
                        Gfits_[x] = 0;
                        Tfits_[x] -= Creator.MRI_[x]*MRIConstant_;
                        Afits_[x] -= Creator.MRI_[x]*MRIConstant_;
                }
        }
        System.out.println ("] done");
}

/**
 * Converts integer to byte, unsigned representation
 *
 * 00000000 - 01111111        represent 0 - 127 in Java's "byte"
 *                                      primitive
 * 10000000 - 11111111        represent -128 to -1 although we want
 *                             them to       represent 128 - 255. Therefore
 *                             we subtract 256 from the integer, i,
```

```
 *                            to store it in unsigned representation
 *                           i.e.    128-256=-128   -> 10000000
 *                           255-256=-1             -> 11111111
 *
 * @param i            integer to store as byte
 * @return             byte representation of integer (unsigned)
 */

public static byte i_to_ub( int i){
        if ( i > 127){
                i-=256;
                return (byte)i;
        } else {
                return (byte)i;
        }
}


/**
 * Converts integer to byte, unsigned representation
 *
 * 00000000 - 01111111        represent 0 - 127 in Java's "byte"
 *                                        primitive
 * 10000000 - 11111111        represent -128 to -1 although we want
 *                            them to        represent 128 - 255. Therefore
 *                            we subtract 256 from the integer, i,
 *                            to store it in unsigned representation
 *                           i.e.    128-256=-128   -> 10000000
 *                           255-256=-1             -> 11111111
 * @param i            double to store as byte, actually not a double
 *                           precision number, but a *positive integer*,
 *                           probably returned by Math.pow
 * @return             byte representation of i (unsigned)
 */
public static byte i_to_ub( double i){
        if ( i > 127){
                i-=256;
                return (byte)i;
        } else {
                return (byte)i;
        }
}

/**
 * @param pb                  PackedBlock containing base for which
 *                            fitness is needed
 * @param position            position within block (not chromosome
 *                            sequence)
 * @return                    fitness value for specified position
 */
public static float getFitnessAt(PackedBlock pb, int position){

        int pos = pb.getStartIdx() + position;

        float fitness = 0.0f;

        byte [] pbData = pb.getBytes();

        int packIdx = position / 4;
        int bitPairSignificance = 3 - (position % 4);

        if ( ((byte) i_to_ub(
                Math.pow(4,bitPairSignificance)*Consts.G)
                        & pbData[packIdx])
                        == i_to_ub(
                                Math.pow(4,bitPairSignificance)*Consts.G )){
                fitness = Gfits_[pos];
        } else if ( ((byte) i_to_ub(
                Math.pow(4,bitPairSignificance)*Consts.C)
                        & pbData[packIdx])
                        == i_to_ub(
```

```
                                        Math.pow(4,bitPairSignificance)*Consts.C )){
                        fitness = Cfits_[pos];
                } else if ( ((byte) i_to_ub(
                        Math.pow(4,bitPairSignificance)*Consts.T)
                                & pbData[packIdx])
                                == i_to_ub(
                                        Math.pow(4,bitPairSignificance)*Consts.T )){
                        fitness = Tfits_[pos];
                } else {
                        fitness = Afits_[pos];
                }
                return fitness;
        }

        /**
         *
         * Calculate fitness for an entire PackedBlock
         *
         * @param pb                  PackedBlock for which you need fitness
         * @param blockNumber  blockNumber in the prototype Indvidual
         * @return                            fitness value for specified PackedBlock
         */
        public static float getFitness (PackedBlock pb){

                float fitness = 0.0f;
                Iterator<Integer> i = pb.mutationIndices_.iterator();
                int idx;
                while (i.hasNext()){
                        idx = i.next().intValue();
                        fitness += getFitnessAt(pb, idx);
                }
                return fitness;
        }
}
```

fitness.FitStruct

```
// FitStruct.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package fitness;

public class FitStruct {
	float fitness_ = 0.0f;
	float probability_ = 0.0f;

	public float getFitness() {
		return fitness_;
	}

	public float getProbability() {
		return probability_;
	}

	protected FitStruct(float fitness, float probability ){
		fitness_ = fitness;
		probability_ = probability;
	}
}
```

gene.Exon

```
// Exon.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package gene;

import tools.Consts;
import tools.Sequence;

public class Exon {

	int number_;

	int start_;
	int end_;
	int strand_;
	int frame_;

	int idxStart_;
	int idxEnd_;

	String sequence_;

	public int getStart() {
		return start_;
	}

	public int getEnd() {
		return end_;
	}

	public int getIdxStart() {
		return idxStart_;
	}

	public int getIdxEnd() {
		return idxEnd_;
	}

	public int getStrand() {
		return strand_;
	}

	public Exon (int start, int end,
			int number, int frame, String sequence){
		sequence_ = sequence;
		start_ = start;
		end_ = end;

		frame_ = frame;

		if (start>end){		// minus strand
			idxEnd_ = Sequence.bpToIdx(start);
			idxStart_ = Sequence.bpToIdx(end);
			strand_ = Consts.MINUS;
		} else {
			idxStart_ = Sequence.bpToIdx(start);

			idxEnd_ = Sequence.bpToIdx(end);
			strand_ = Consts.PLUS;
		}
	}
}
```

gene.Gene

```
// Gene.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package gene;
import java.util.Vector;
import tools.Consts;

public class Gene {

	private int strand_;

	private int idxStart_ = Consts.MAX_INT;
	private int idxEnd_ = 0;

	public String name_;
	Vector<Exon> exons_;

	public int getStrand() {
		return strand_;
	}

	public Vector<Exon> getExons() {
		return exons_;
	}

	public int getIdxStart() {
		return idxStart_;
	}

	public int getIdxEnd() {
		return idxEnd_;
	}

	public Gene(String name, int strand){
		name_ = name;
		strand_ = strand;
		exons_ = new Vector<Exon>();
	}

	public void addExon(Exon e){
		exons_.add(e);
	}

	public void computeIndexes(){
		for (int i = 0; i < exons_.size(); i ++){
			if ( exons_.elementAt(i).getIdxStart() < idxStart_){
				idxStart_ = exons_.elementAt(i).getIdxStart();
			}
			if ( exons_.elementAt(i).getIdxEnd() > idxEnd_){
				idxEnd_ = exons_.elementAt(i).getIdxEnd();
			}
		}
	}
}
```

individual.Creator

```
// Creator.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package individual;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Vector;

import dominance.DomStruct;

import tools.Consts;
import tools.Sequence;

import bitpack.PackedBlock;
import bitpack.PackedChromosome;

import gene.Exon;
import gene.Gene;

public class Creator {

        static Vector<DomStruct> dominanceProbabilities_;
        static Individual adam_;

        private static int maxSize_ = Consts.MAX_INT;
        public static String strChr_;

        public static int [] posBlock_;
        public static int [] posOffset_;

        private static int chrSize_;

        public static boolean plusExon [];
        public static boolean minusExon [];
        public static boolean plusIntron [];
        public static boolean minusIntron [];
        public static boolean plusPromoter [];
        public static boolean minusPromoter [];
        public static boolean plusIntergenic [];
        public static boolean minusIntergenic [];

        public static boolean isCDS_[];
        public static boolean isExonic_[];

        public static String [] CDS_ ;

        public static byte MRI_[];

        public static String [] geneNames_;
        public static int [] frame_;

        static HashMap <String, Gene> genesHash_;
        static Vector<Gene> plusGenesVector_ = new Vector<Gene>();
        static Vector<Gene> minusGenesVector_ = new Vector<Gene>();

        public static int nextBase_[];


        public static void readExons(String exonFileName, int size){
                genesHash_ = new HashMap <String, Gene> ();
```

```
nextBase_ = new int[size];
Arrays.fill(nextBase_, -1);
isCDS_ = new boolean[size];
isExonic_ = new boolean[size];
posBlock_ = new int[size];
posOffset_ = new int[size];

frame_ = new int[size];

//geneNames_ = new String[size];
//CDS_ = new String[size];
//Arrays.fill(CDS_, null);
// frame = new byte[size];

Arrays.fill(frame_, -1);
Arrays.fill(isExonic_, false);

try {
        BufferedReader in =
                new BufferedReader(new FileReader(exonFileName));
        String line;
        String geneName = null;
        String strStrand;

        int intStrand = 0;

        while(in.ready()){
                line = in.readLine();
                String [] tokens = line.split("\t");

                String [] positionTokens =
                        (new String(tokens[2])).split("\\.\\.");
                String [] geneTokens =
                        (new String (tokens[3])).split("_");
                String [] exonTokens =
                        (new String (geneTokens[1])).split("ex");

                int exonNum = new Integer(exonTokens[1]);

                geneName = geneTokens[0];

                int start = new Integer(positionTokens[0]);
                int end =  new Integer(positionTokens[1]);

                if ( start > size || end > size)
                        break;

                int frame = new Integer(tokens[4]);

                strStrand = new String (tokens[1]);

                int idxStart;
                int idxEnd;

                String exonSeq;

                if (strStrand.equals("plus") ){
                        intStrand = Consts.PLUS;
                        idxStart = Sequence.bpToIdx(start);
                        idxEnd = Sequence.bpToIdx(end);
                        exonSeq = strChr_.substring(idxStart, idxEnd+1);
                } else {
                        intStrand = Consts.MINUS;
                        idxStart = Sequence.bpToIdx(end);
                        idxEnd = Sequence.bpToIdx(start);
                        exonSeq =
                                new StringBuffer(
                                        strChr_.substring(
                                                idxStart,idxEnd+1)).
                                reverse().toString();
                        exonSeq = exonSeq.replace('A', 'X');
```

```
                                exonSeq = exonSeq.replace('T', 'A');
                                exonSeq = exonSeq.replace('X', 'T');
                                exonSeq = exonSeq.replace('C', 'Y');
                                exonSeq = exonSeq.replace('G', 'C');
                                exonSeq = exonSeq.replace('Y', 'G');
                        }


                        if (genesHash_.get(geneName)==null){
                                Gene g = new Gene(geneName,intStrand);
                                genesHash_.put(geneName, g);
                        }


                        Exon e = new Exon(start,end,exonNum,frame,exonSeq);

                        for (int i = e.getIdxStart(); i<=e.getIdxEnd(); i++){
                                isExonic_[i] = true;
                        }
                        genesHash_.get(geneName).addExon(e);
                }
                ArrayList <String> keys =
                        new ArrayList<String>(genesHash_.keySet());
                Iterator<String> it = keys.iterator();
                while (it.hasNext()){
                        String gn = it.next();
                        genesHash_.get(gn).computeIndexes();
                        if( genesHash_.get(gn).getStrand() == Consts.PLUS)
                                plusGenesVector_.add(genesHash_.get(gn));
                        else if (genesHash_.get(gn).getStrand()
                                == Consts.MINUS)
                                minusGenesVector_.add(genesHash_.get(gn));
                        for (int i = genesHash_.get(gn).getIdxStart();
                                i <= genesHash_.get(gn).getIdxEnd(); i++){
                                isCDS_[i] = true;
                        }
                }
        } catch (IOException e) {
                // TODO Auto-generated catch block
                // TODO find sandwich
                e.printStackTrace();
        }
        Iterator<Gene> pgv = plusGenesVector_.iterator();
        while (pgv.hasNext()){
                Gene g = pgv.next();
                Iterator<Exon> ei = g.getExons().iterator();
                while (ei.hasNext()){
                        Exon e = ei.next();
                        for (int i=e.getIdxStart(); i<= e.getIdxEnd(); i++){
                                nextBase_[i]=i+1;
                        }
                }
        }

}


public static Vector<Gene> getGenesVector() {
        return plusGenesVector_;
}

static char [] chromosome;

public static HashMap<String, Gene> getGenes(){
        return genesHash_;
}

public static int getPosBlock(int pos) {
        return posBlock_[pos];
}
```

```
public static int getPosOffset(int pos) {
	return posOffset_[pos];
}

public static void readMRI(String MRIFileName, int size){
	MRI_ = new byte[size];
	try {
		BufferedReader in =
			new BufferedReader(new FileReader(MRIFileName));
		String line;

		while(in.ready()){
			line = in.readLine();
			String [] tokens = line.split("\t");
			int start =
				Sequence.bpToIdx(
					new Integer(tokens[0]).intValue());
			int end =
				Sequence.bpToIdx(
					new Integer(tokens[1]).intValue());

			if (end < size){
				if ( tokens[3].equals("GC_rich")){
					for (int x = start ; x <= end; x++)
						MRI_[x] = 1;
				} else {
					for (int x = start ; x <= end; x++)
						MRI_[x] = -1;
				}
			}
		}
	} catch (IOException e) {
		e.printStackTrace();
	}
}

public static PackedChromosome getMaternalChromsome(){
	return adam_.getMaternalChromosome();
}

public static int getFrame(int pos)  {
	if (frame_[pos] >= 0){
		throw new IllegalArgumentException("frame must be 0 - 2");
	} else {
		return frame_[pos];
	}
}

public static void createIndividual(int maxBlockSize){

	int startIdx = 0;
	int endIdx = 0;

	int blockNumber = 0;

	// creates a new packedChromosome to which blocks will be
	// added
	PackedChromosome pc = new PackedChromosome(chrSize_);

	int x = 0;

	boolean state = isCDS_[x];
	// STATE is false when intergenic

	int id = 0;

	while (x < chrSize_){
		int offset = 0;
		int intergenicSize = 0;
		while(x < chrSize_ && isCDS_[x] == state ){
			posBlock_[x] = blockNumber;
```

```
                        // array gives packedblock number for every
                        // position in genome
                        posOffset_[x] = offset;
                        offset++;
                        x++;
                        intergenicSize++;
                        if (state == false && intergenicSize == maxBlockSize)
                                break;
                        // if intergenic, and size is larger than maximum
                        // intergenic size
                }
                startIdx=endIdx;
                endIdx=x;
                String seq = strChr_.substring(startIdx, endIdx);
                id++;
                PackedBlock pb =
                        new PackedBlock(seq,blockNumber,startIdx,state,id);

                pb.setStartIdx(startIdx);
                pb.setEndIdx(endIdx-1);
                pc.addBlock(pb);
                blockNumber++;
                if (x < chrSize_)
                        state = isCDS_[x];
        }
        pc.setNumBlocks(blockNumber);
        System.out.println("Number of blocks : " + blockNumber);
        adam_ = new Individual (pc,pc);
}

// RETURN TRUE if either plus OR minus strand is
// intronic/exonic/promoter
public static boolean isGenic(int pos){
        if (    plusExon[pos] || minusExon[pos] || plusIntron[pos] ||
                        minusIntron[pos] || plusPromoter[pos]
                        || minusPromoter[pos] ){
                return true;
        } else{
                return false;
        }
}

public static int getChrSize() {
        return chrSize_;
}

public static String getStrChr() {
        return strChr_;
}

public static Individual getIndividual(){
        return adam_;
}

/**
 * Reads nucleotide data from chrFileName into the string strChr
 * @param chrFileName filename containing a single contig
 */
public static void init ( String chrFileName, int maxSize){
        maxSize_ = maxSize;
        init (chrFileName);
}

/**
 * Reads nucleotide data from chrFileName into the string strChr
 * @param chrFileName filename containing a single contig
 */
public static void init ( String chrFileName ) {
        try {
                BufferedReader in =
                        new BufferedReader(new FileReader(chrFileName));
```

```
                        String line = null;

                        int linesRead = 0;

                        while(in.ready())
                        {
                                line = in.readLine();
                                linesRead ++;
                                if ( line.charAt(0) == '>' ){
                                        continue;
                                }
                                if (chrSize_ + line.length() <= maxSize_){
                                        chrSize_ += line.length();
                                } else {
                                        break;
                                }
                                line = null;
                        }
                        in.close();

                        chromosome = null;
                        chromosome = new char[chrSize_];

                        in = null;
                        in = new BufferedReader(new FileReader(chrFileName));
                        int counter = 0;
                        while(in.ready())
                        {
                                line = in.readLine();
                                if ( line.charAt(0) != '>'){
                                        if ( counter + line.length() > maxSize_){
                                                break;
                                        }
                                        line.getChars(0,line.length(),chromosome,counter);
                                        counter += line.length();
                                }
                        }
                        in.close();
                        in = null;
                        strChr_ = new String(chromosome);
                } catch (IOException ioe){
                        ioe.printStackTrace();
                }
        }
}
```

individual.Individual

```
// Individual.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package individual;

import java.util.Arrays;
import java.util.Vector;
import java.lang.System;

import tools.Consts;


import engine.Crossover;
import engine.Stats;

import bitpack.BitPack;
import bitpack.PackedBlock;
import bitpack.PackedChromosome;


public class Individual {
	// fields

	int gender_;
	float fitnessPercentileRank_;

	public float getFitnessPercentileRank() {
		return fitnessPercentileRank_;
	}

	public void setFitnessPercentileRank(
			float fitnessPercentileRank){
				this.fitnessPercentileRank_ = fitnessPercentileRank;
	}

	private PackedChromosome maternal_;
	private PackedChromosome paternal_;

	public float fitness = 0.0f;

	// methods

	public int getGender() {
		return gender_;
	}

	public void setGender(int gender) {
		this.gender_ = gender;
	}
	private static int flip(int one_or_zero){
		switch(one_or_zero){
			case 0: one_or_zero = 1; break;
			case 1: one_or_zero = 0; break;
		}
		return one_or_zero;
	}

	public PackedChromosome getMaternalChromosome(){
		return this.maternal_;
	}

	public PackedChromosome getGamete(){
		// NEEDS TO BE COMPLETELY REWRITTEN

		int numCrossovers = Crossover.getNumCrossovers();
		int numLetters = maternal_.getNumLetters();
```

```
int startChr = Stats.nextInt(2);
int currentChr = startChr;

PackedChromosome gamete = null;
PackedChromosome copyChr = null;

PackedBlock pbTmp;

switch ( startChr ){
case (Consts.MATERNAL):
       gamete = new PackedChromosome(maternal_);
       copyChr = paternal_;
       break;
case (Consts.PATERNAL):
       gamete = new PackedChromosome(paternal_);
       copyChr = maternal_;
       break;
}

if (numCrossovers == 0) return gamete;

// create an array to hold crossover locations
// since crossover occurs between pairs of locations,
// last crossover position is the end of the chromosome
int [] crossovers = new int[numCrossovers+1];

for (int x = 0; x < numCrossovers; x++){
       crossovers[x] = Crossover.getRandomCrossover();
}

crossovers[numCrossovers] = numLetters - 1;

// sort crossovers, since we will be taking position
// pairs from index 0 to numLetters-1
Arrays.sort(crossovers);

// randomly determine which chromosome to start with
// 0: maternal, 1: paternal
int startIdx = 0;
int endIdx = -1;
int blockIdx = 0;

currentChr = flip(currentChr);

////
//// Crossover loop
////

for (int x = 0; x <= numCrossovers; x++){
       if (x > 0 && Creator.getPosBlock(crossovers[x]) ==
              Creator.getPosBlock(crossovers[x-1])){
              continue;
       }
       // System.out.println( "Crossover : " + x );
       currentChr = flip(currentChr);
       startIdx = endIdx+1;
       endIdx = crossovers[x];
       // get last character index for this piece

       if (startIdx >= endIdx) { continue; }
       if (startIdx > crossovers[numCrossovers]) { break; }
       // break out if start index is greater than
       // index of last crossover in array

       // switch between maternal <-> paternal

       if (startIdx == 0) {   // on the first crossover
              while ( gamete.getBlock(blockIdx).getEndIdx()
                     < endIdx ){
```

```
                        // find block where crossover occurs
                        blockIdx++;
                }
                if (blockIdx > 0 &&
                                gamete.getBlock(blockIdx-1).getEndIdx()
                                == endIdx){
                        // if previous blocks end idx is equal to
                        // crossover endidx, continue the loop
                        continue;
                } else {
                        int offset =
                                endIdx -
                                gamete.getBlock(blockIdx).getStartIdx();
                        if (gamete.getBlock(blockIdx) !=
                                copyChr.getBlock(blockIdx)){
                                pbTmp = BitPack.CrossBlocks(
                                        gamete.getBlock(blockIdx),
                                        copyChr.getBlock(blockIdx), offset);
                                pbTmp.computeFitness();
                                gamete.setBlock(pbTmp, blockIdx);
                                continue;
                        }
                }
        } else {
                // not the first crossover pair
                ///
                /// Copying from the "Other" Chromosome
                //
                if (currentChr != startChr){
                        // if we just switched in, to copying data from
                        // the non-Gamete chromosome, check if our next
                        // crossover is in the same block
                        if (blockIdx < gamete.getNumBlocks() &&
                                        gamete.getBlock(blockIdx).getEndIdx()
                                        > endIdx ){
                                // if it is,  more flipping
                                int offset =
                                        endIdx -
                                gamete.getBlock(blockIdx).getStartIdx();
                                if (gamete.getBlock(blockIdx) !=
                                        copyChr.getBlock(blockIdx)){
                                        pbTmp = BitPack.CrossBlocks(
                                                copyChr.getBlock(blockIdx),
                                        gamete.getBlock(blockIdx),offset);
                                                pbTmp.computeFitness();
                                        gamete.setBlock(pbTmp, blockIdx);
                                }
                                continue;
                        } else {
                                while ( blockIdx < gamete.getNumBlocks() &&
                                        gamete.getBlock(blockIdx).getEndIdx()
                                                <=      endIdx ){
                                        gamete.setBlock(copyChr.
                                                        getBlock(blockIdx),
                                                        blockIdx);
                                        blockIdx++;
                                }
                                if (blockIdx > 0 &&
                                                gamete.getBlock(blockIdx-1).
                                                        getEndIdx()== endIdx){
                                        continue;
                                } else if (blockIdx < gamete.getNumBlocks()
                                                &&
                                                blockIdx > 0){
                                        int offset =
                                                endIdx-
                                                gamete.getBlock(blockIdx-1).
                                                        getEndIdx()-1;
                                        if (gamete.getBlock(blockIdx) !=
                                                copyChr.getBlock(blockIdx)){
                                                pbTmp = BitPack.CrossBlocks(
```

```
                                                copyChr.getBlock(blockIdx),
                                                gamete.getBlock(blockIdx),
                                                        offset);
                                                pbTmp.computeFitness();
                                                gamete.setBlock(pbTmp,
                                                        blockIdx);
                                                continue;
                                        }
                                }
                        }
                ////
                //// Copying from the Me Template Chromosome
                ///
                } else { // current chromosome is the gamete
                        if ( blockIdx < gamete.getNumBlocks() &&
                                        gamete.getBlock(blockIdx).getEndIdx()
                                                > endIdx ){
                                int offset =
                                        endIdx - gamete.getBlock(blockIdx).
                                                getStartIdx();
                                if (gamete.getBlock(blockIdx) !=
                                        copyChr.getBlock(blockIdx)){
                                        pbTmp = BitPack.CrossBlocks(
                                                gamete.getBlock(blockIdx),
                                                copyChr.getBlock(blockIdx),
                                                offset);
                                        pbTmp.computeFitness();
                                        gamete.setBlock(pbTmp, blockIdx);
                                }
                                continue;
                        } else {
                                while ( blockIdx < gamete.getNumBlocks()
                                                &&
                                        gamete.getBlock(blockIdx).getEndIdx()
                                                <= endIdx ){
                                        // copy nothing in
                                        blockIdx++;
                                }
                                if (blockIdx > 0 &&
                                                gamete.getBlock(blockIdx-1).
                                                        getEndIdx() ==
                                                        endIdx){
                                        continue;
                                } else if (blockIdx < gamete.getNumBlocks()
                                                &&
                                                blockIdx > 0 ){
                                        int offset =
                                                endIdx-
                                                gamete.getBlock(blockIdx-1)
                                                .getEndIdx()-1;
                                        if (gamete.getBlock(blockIdx)!=
                                                copyChr.getBlock(blockIdx)){
                                                pbTmp = BitPack.CrossBlocks(
                                                gamete.getBlock(blockIdx),
                                                copyChr.getBlock(blockIdx),
                                                        offset);
                                                pbTmp.computeFitness();
                                                gamete.setBlock(pbTmp,
                                                        blockIdx);
                                        }
                                        continue;
                                }
                        }
                }
                        }
                }
        return gamete;
}

public float getFitness(){
        return fitness;
```

```
	}

	public void randomMutation(int numMutations){
		for (int x = 0; x < numMutations; x++){
			switch (Stats.nextInt(2)){
				case 0:
					maternal_.randomMutation();
					break;
				case 1:
					paternal_.randomMutation();
					break;
			}
		}
	}

	public void computeFitness(){
		// fitness = maternal_.getFitness() + paternal_.getFitness();

		fitness = 0.0f;
		Vector<PackedBlock> mBlocks = maternal_.getBlocks();
		Vector<PackedBlock> pBlocks = paternal_.getBlocks();
		float mFitness;
		float pFitness;

		for (int x = 0; x < mBlocks.size(); x++){
			PackedBlock mBlock = mBlocks.elementAt(x);
			PackedBlock pBlock = pBlocks.elementAt(x);
			mFitness = mBlock.getFitness();
			pFitness = pBlock.getFitness();
			float maxCoeff = mBlock.getMaxCoefficient();
			float minCoeff = mBlock.getMinCoefficient();
			float meanCoeff = mBlock.getMeanCoefficient();
			fitness += Math.max(mFitness, pFitness)*maxCoeff +
				Math.min(mFitness, pFitness)*minCoeff +
					(mFitness+pFitness)/2*meanCoeff;
		}
	}

	public Individual (PackedChromosome maternal,
					PackedChromosome paternal){

		this.gender_ = Stats.nextInt(2);

		this.maternal_ = new PackedChromosome(maternal);
		this.paternal_ = new PackedChromosome(paternal);

		this.computeFitness();
	}

	public Individual (Individual i){

		this.gender_ = Stats.nextInt(2);

		maternal_ = new PackedChromosome(i.maternal_);
		paternal_ = new PackedChromosome(i.paternal_);

		// this constructor only used at beginning of population
		// initiation
		this.fitness = 0.0f;
	}

	public void printFitness(){
		System.out.println( fitness );
	}
}
```

substitution.Substitution

```
// Substitution.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package substitution;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Iterator;
import java.util.Vector;

import engine.Stats;

public class Substitution {

	static Vector<SubStruct> mutationProbabilities_;

	public static void init(String mutationFileName){

		mutationProbabilities_ = new Vector<SubStruct>();
		try {
			BufferedReader in =
				new BufferedReader(new FileReader(mutationFileName));
			String line;
			while(in.ready()){
				line = in.readLine();
				String [] tokens = line.split("\t");

				String startingBase = tokens[0];
				String endingBase = tokens[1];
				float probability = (new Float(tokens[2])).floatValue();

				SubStruct mp =
					new SubStruct(startingBase,endingBase,probability);

				mutationProbabilities_.add(mp);
			}
		} catch (IOException e) {
			// TODO Auto-generated catch block
			// TODO find sandwich
			e.printStackTrace();
		}
	}

	public static String getPair(){
		float prob = Stats.nextFloat(); // cumulative probability
		Iterator<SubStruct> i = mutationProbabilities_.iterator();
		float cumProb = 0.0f;

		SubStruct m = null;
		while (i.hasNext()){
			m = i.next();
			cumProb += m.getProbability();
			if (cumProb >= prob) {
				break;
			}
		}
		return new String (m.getStartingBase() + m.getEndingBase());
	}
}
```

substitution.SubStruct

```
// SubStruct.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package substitution;

public class SubStruct {

        public String startingBase_;
        public String endingBase_;
        float probability_;

        public String getStartingBase() {
                return startingBase_;
        }

        public String getEndingBase() {
                return endingBase_;
        }

        public float getProbability() {
                return probability_;
        }

        SubStruct( String startingBase, String endingBase,
                        float probability) {
                startingBase_ = startingBase;
                endingBase_ = endingBase;
                probability_ = probability;
        }
}
```

tools.Clock

```
// Clock.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package tools;

public class Clock {

	static long current;
	static long start;
	static long elapsed;

	static long current2;
	static long start2;
	static long elapsed2;

	public static void startClock2() {
		start2 = System.currentTimeMillis();
	}

	public static long getStepTime2(){

		current2 = System.currentTimeMillis();

		elapsed2 = current2 - start2;
		start2 = current2;
		return elapsed2;
	}

	public static void startClock() {
		start = System.currentTimeMillis();
	}

	public static long getStepTime(){

		current = System.currentTimeMillis();

		elapsed = current - start;
		start = current;
		return elapsed;
	}
}
```

tools.Consts

```
// Consts.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package tools;

public final class Consts {

	public static final int PLUS = 0;
	public static final int MINUS = 1;

	public static final int MATERNAL = 0;
	public static final int PATERNAL = 1;

	public static final int DOMINANT = 2;
	public static final int NEGDOMINANT = 1;
	public static final int INTERGENIC = 0;

	public static final int MALE = 0;
	public static final int FEMALE = 1;

	public static final int MAX_INT = 2147483647;

	public static final int A = 0;
	public static final int C = 1;
	public static final int T = 2;
	public static final int G = 3;

	private Consts(){
		//prevents the native class from
		//calling this constructor
	    throw new AssertionError();
	}
}
```

tools.Sequence

```
// Sequence.java
// Copyright 2010 Andrew McSweeny
// All rights reserved

package tools;

public class Sequence {

	// Converts 1-based positions from data files to indexes in arrays
	public static int bpToIdx(int pos){
		return pos-1;
	}
}
```