2009

MSG: A Gap-Oriented Genetic Algorithm for Multiple Sequence Alignment

Philip Heller
San Jose State University

MSG: A GAP-ORIENTED GENETIC ALGORITHM FOR MULTIPLE SEQUENCE ALIGNMENT

A Writing Project
Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfillment
of the Requirement for the Degree
Master of Science

by
Philip Heller
May 2009

SAN JOSĒ STATE UNIVERSITY

The Undersigned Committee Approves the Writing Project Titled

MSG: A GAP-ORIENTED GENETIC ALGORITHM FOR MULTIPLE SEQUENCE ALIGNMENT

by
Philip Heller

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Sami Khuri,	Department of Computer Science	Date
Dr. Martina Bremer,	Department of Mathematics	Date
Dr. David Taylor,	Department of Computer Science	Date

APPROVED FOR THE UNIVERSITY

Associate Dean Office of Graduate Studies and Research Date

ABSTRACT

MSG: A GAP-ORIENTED GENETIC ALGORITHM FOR MULTIPLE SEQUENCE ALIGNMENT

by
Philip Heller

Traditional Multiple Sequence Alignment (MSA) Algorithms are deterministic. Genetic algorithms for protein MSA have been documented. However, these are not able to exceed in all cases the scores obtained by Clustal-W, the freely available de-facto standard. My solution, called “MSG”, places gaps rather than amino acids. The algorithm is multi-tribal, uses only a few very simple operators with adaptive frequencies, and jumpstarts one population from the Clustal-W solution. Results are reported for 14 data sets, on all of which MSG exceeds the Clustal-W score.

Acknowledgements

To begin at the beginning, first thanks go to James Casaletto, who introduced me to bioinformatics a few years ago over a beer at Siren’s in Port Townsend, WA, on the way south after his honeymoon. I said, “What’s bioinformatics?” and he said, “You should meet this guy Sami…”

Next, thanks to my colleague Doug Merritt, who seems to have read and understood every published paper on every known topic, and takes delight in offering extremely helpful suggestions.

Next I want to acknowledge Professors Martina Bremer and David Taylor, who have kindly been on my committee despite heavy teaching and advising commitments. They have provided valuable perspectives on my project that I never could have come up with on my own.

Lastly, let me try to adequately thank Professor Sami Khuri, who has been my advisor and my friend throughout my M.S. career. His advice has been profound. For example, the “jumpstarted tribe” idea (see Chapter 4) was his idea; it guarantees that my algorithm can do no worse than the de facto standard tool for solving the multiple sequence alignment problem. Prof. Khuri has always been available with insight, kindness, and even, when I needed it the most, food. I have proof that the bioinformatics academic community has deep respect for his mentorship. This year the U.C. Santa Cruz Biomolecular Engineering Department, one of the world’s premier bioinformatics research institutions, accepted three of his students into their Ph.D. program. I am one of them.

List of Tables

List of Figures

Chapter 1: Background on the MSA Problem

In this chapter I provide a brief background for non-biologists, and then explain the importance of the general MSA problem. Readers with an understanding of genetic sequences and alignments may skip the background material.

Background for Non-Biologists

Cells contain long single-molecule strands, called *chromosomes*, composed of deoxyribonucleic acid (DNA). DNA consists of a double-helix structural skeleton that can hold a linear sequence of units called *nucleotides* or *bases*. There are 4 nucleotide species: adenine, cytosine, guanine, and thymine, abbreviated as A, C, G, and T. Thus a chromosome can be represented as a string on the alphabet {ACGT}. The human cell contains 46 chromosomes with a total of about 3 billion nucleotides; their total uncoiled length is nearly 2 meters. (Per cell! Humans have ~50 trillion cells, which gives us 10^{11} km of chromosomes. That's 1% of a light year.)

Some substrands of DNA are *genes*. Genes are blueprints for manufacturing proteins, which implement most structure and function in the body. Like chromosomes, proteins are linear molecules with a structural skeleton that holds a variety of subunits. Unlike chromosomes, proteins use amino acids as subunits. There are 20 amino acids, so proteins are represented as strings on a size-20 alphabet: {ACDEFGHIKLMNPQRSTVWY}. Each triplet of DNA bases corresponds to an amino acid (with some redundancy, as there are 64 possible base triplets). Cells use a variety of molecular resources to translate gene information into proteins.

Genes are imperfect data bases. Chromosomes undergo mutation, especially when they replicate prior to cell division: bases can be inserted into or deleted from a gene, or one base type can be replaced by another. If the modification happens in a germ cell, then the mutation and its anatomic consequences can be passed to offspring. If such mutations confer benefit to their organism, they can eventually spread through a population. As an isolated population acquires more and more novel genetic traits, it eventually becomes a new species. Related genes in related species typically have subtly different sequences; the same can be said of the proteins coded by those genes. For example, all animal species that have blood have some kind of globin gene, which encodes the globin protein that carries oxygen through the blood. The precise base sequence of the globin gene (and therefore the precise amino acid sequence of the protein) varies from species to species.

Biology assumes that all species evolved from a common ancestor. Each species evolves in its own way under pressure from its unique ecological niche. Differences between related genetic sequences reflect the amount of time they have had to evolve independently. Similarities between related sequences provide clues about the common ancestor.

A sequence alignment is a presentation of related sequences. The sequence are written horizontally. Order is preserved, but hyphens may be inserted at any point. For example, here are two possible alignments of the strings “DOGGY”, “DONKEY” and “MONKEY”. The strings at the bottom are called *consensus* strings; consensus is determined by simple column-by-column majority rule.

```
DOG-GY          -D-OG-GY
DONKEY          -D-ONKEY
MONKEY          M-O-NKEY
======          ========
DONKEY          MDOONKEY
```

The first alignment suggests that the common ancestor is “DONKEY”. A substitution in the first position created “MONKEY”, while a substitution and a deletion resulted in “DOGGY”.

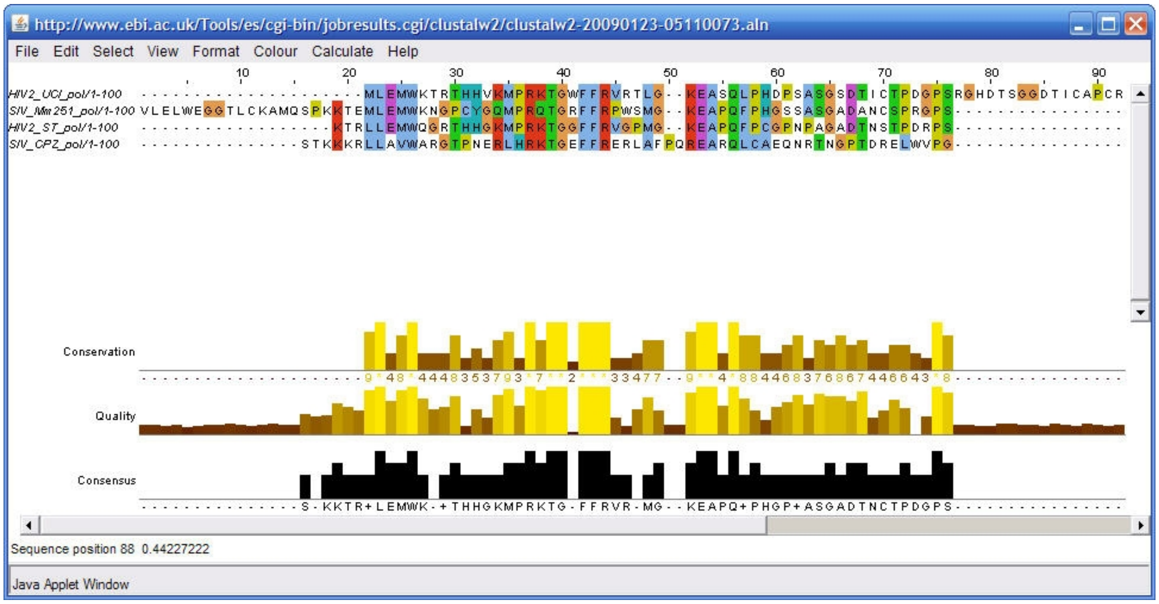
We sense that the second alignment is somehow inferior to the first. It’s feasible, but more mutation events must be assumed in order to explain the sequences. Clearly a huge number of alignments are possible for any data set. We need an objective measure of the quality of an alignment. In a biological context, this measure should penalize unlikely evolutionary events in a manner that is consistent with their actual (un)likelihood. Special care must be taken with protein sequences, where certain substitutions are significantly less probable than others. There are reliable standard tools for scoring protein alignments.

This background should be enough to prepare the reader for all the material that follows.

Importance of Multiple Sequence Alignment

In Werner Herzog's documentary film "Encounters at the End of the World", biologist Jan Pawlowski is shown comparing the DNA of several previously unknown aquatic microbial species that he collected from the ocean beneath the Antarctic ice cap. The data on his laptop screen is easily recognizable as a set of genetic sequences. Viewers with bioinformatics experience can recognize his application as JalView, an auxiliary tool to the popular ClustalW multiple sequence alignment program. Pawlowski is using Multiple Sequence Alignment (MSA) to establish evolutionary relationships among his new species.

Figure 1.1 shows a sample JalView screen. The top half of the image is characteristic of MSAs: rows of character sequences are stacked vertically to reveal regions of species-to-species similarity. The degree of similarity can be enhanced by inserting gaps into the sequences. Indeed, as this paper will show, the entire MSA problem can be reduced to optimal placement of these gaps.



http://www.ebi.ac.uk/Tools/es/cgi-bin/jobresults.cgi/clustalw2/clustalw2-20090123-05110073.aln

```
                                  10        20        30        40        50        60        70        80        90
HIV2_UCI_pol/1-100     ---------------------MLEMWKTRTHHVKMPRKTGWFFRVRTLG--KEASQLPHDPSASGSDTICTPDGPSRGHDTSGGDTICAPCR
SIV_Mm251_pol/1-100    VLELWEGGTLCKAMQSPKKTEMLEMWKNGPCYGQMPRQTGRFFRPWSMG--KEAPQFPHGSSASGADANCSPRGPS----------------
HIV2_ST_pol/1-100      ------------------KTRLLEMWQGRTHHGKMPRKTGGFFRVGPMG--KEAPQFPCGPNPAGADTNSTPDRPS----------------
SIV_CPZ_pol/1-100      ---------------STKKKRLLAVWARGTPNERLHRKTGEFFRERLAFPQREARQLCAEQNRTNGPTDRELWVPG----------------

Conservation
                       ---------------------9*48*4448353793*7**2***33477--9**4*8844683768674466 43*8----------------
Quality

Consensus
                       ---------------S-KKTR+LEMWK-+THHGKMPRKTG-FFRVR-MG--KEAPQ+PHGP+ASGADTNCTPDGPS----------------

Sequence position 88  0.44227222

Java Applet Window
```

Figure 1.1: Sample JalView Screenshot (`www.ebi.ac.uk/Tools/clustalw2`)

MSA applications are not limited to the life sciences. However, very few classes in the physical world of 3 dimensions can be well represented by long linear sequences of symbols that need aligning. Aside from comparisons of text , MSA applications tend to be oriented to genetics, and to operate on sequences of nucleotides or amino acids.

The most obvious use of MSA is to measure the similarities or differences among sequences. For example, in the early stages of HIV infection, infected cells present a protein called HLV on their membrane surfaces. The host's immune system can recognize HLV and attack the infected cells. However, as the disease matures into its chronic phase, the virus mutates so that 3rd amino acid of HLV changes from Threonine ("T") to Asparginine ("N"). The new version of HLV is unrecognizable by the immune system, and the virus can reproduce widely. Multiple sequence alignment of a patient's viral DNA can indicate the degree of mutation.

In addition to the value it contributes in its own right, MSA is the first step in building two major bioinformatics tools: phylogenetic trees and Profile Hidden Markov Models.

Phylogenetic trees illustrate the evolutionary descent of related species or characteristics as they evolve from a common ancestor. Degree of evolution is inferred from degree of divergence from the consensus. For example, Figure 1.2 [35] shows a phylogenetic tree for a membrane ion pump protein gene.

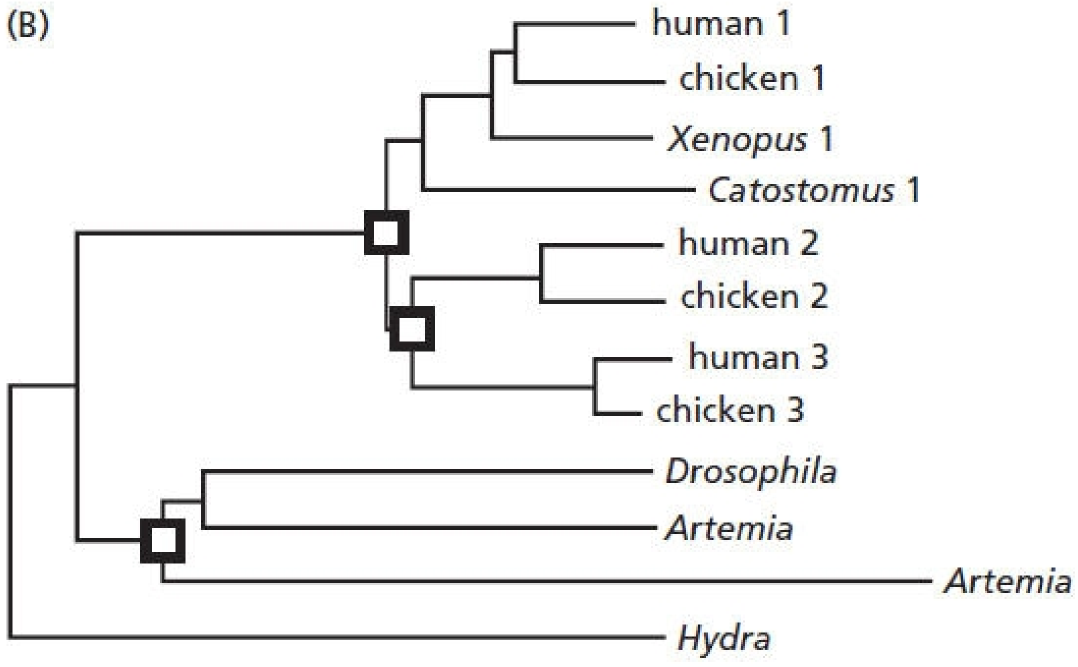


Figure 1.2: A Phylogenetic Tree

The tree reveals a surprising result, which is exactly what we want from a useful tool: version 1 of the human protein is more closely related to chicken 1 than it is to human 2.

Many tree-construction algorithms begin by aligning their sequences. And ClustalW (the de facto standard MSA tool) automatically generates phylogenetic trees.

The other bioinformatics tool that uses MSA as a starting point is the Profile Hidden Markov Model (PHMM). General Hidden Markov Models (HMMs) are pattern recognizers that can tolerate considerable input variability. [32] A Profile HMM uses a protein MSA to model (i.e. “profile”) the general pattern expressed by the individuals. The PHMM can then compute how well a new sequence matches the profile.

So we see that MSAs are important *per se*, and are also starting components of other vital tools. This paper presents a promising new technique for computing protein MSAs. The technique, which uses a Genetic Algorithm, is explained in detail in Chapter 3. But first, Chapter 2 explains Genetic Algorithms.

Chapter 2: Genetic Algorithms

My project is at its heart a genetic algorithm (GA). This chapter explains what a GA is. The subsequent two chapters – my literature review and the explanation of my algorithm – will be clear in the context of the information given here.

The Need for Heuristics

Genetic algorithms are heuristic. In other words, they are the weapon of last resort against nearly intractable problems. This idea is best understood by looking at the limitations of the other algorithm families: exact algorithms and approximate algorithms.

An exact algorithm always produces a correct answer for its input set. Clearly this is preferred above all other options, provided the algorithm exists, is known, and can execute in reasonable time. Unfortunately, this is often not the case. An upper-bound rule of thumb is that if a polynomial (or faster) algorithm can be found, you have a chance; otherwise you don't have a chance. That is, with worse than polynomial complexity, a data set of reasonable size can't be processed in reasonable time. The situation is even rougher in the life sciences, where the data sets are quite large (the longest human chromosome contains $2.45x10^8$ units of information). Even low-order polynomial algorithms can be too slow with such input sizes.

When an exact and efficient algorithm cannot be found, the first option is an approximate algorithm. This type of algorithm produces a result that might not be optimum, but is guaranteed to be within some tolerance of the exact answer [19]. As long as the tolerance is tolerable, an approximate solution is satisfactory. A good rationale for approximate algorithms appears in [33]: "It may help to think of the correctness of an algorithm as a goal that we seek to seek to optimize in conjunction with other goals." Here the other goal is to still be alive when the algorithm terminates.

In the absence of an exact or approximate algorithm, the only remaining option is a heuristic approach. Heuristic algorithms are simply commonsense rules that are known to yield generally acceptable results in acceptable time. Heuristics are difficult to analyze. They offer no guarantee regarding proximity to the exact answer. They can require extensive tuning, and poor tunings can produce strikingly poor results. However, in the right circumstances the results can be strikingly good.

Genetic algorithms are heuristics. This should come as no surprise, since they imitate life processes; there is nothing less precise than life. The next section examines the workings of GAs.

The GA Metaphor

“Genetic” in “genetic algorithm” is a metaphor. GAs imitate certain genetic reproductive processes to explore large multi-dimensional solution spaces. A more accurate name might be “genetically inspired algorithm.” The three processes that are imitated are *crossover* and *mutation*, which increase variety within a population; and *survival of the fittest*.

Recall that a chromosome is a linear molecule containing genes, which contain instructions for making proteins, which in turn implement individual traits. Most cells are *diploid* – that is, they contain two versions of each chromosome. One version is inherited from each parent, so the chromosomes are not necessarily alike. Variation is possible within a gene. The different possible values a gene can take on are called *alleles*. For example, one allele of the human beta-globin gene produces healthy blood, while an allele common in people of black African descent causes sickle-cell anemia.

At this point, computer scientists might be imagining chromosomes as 1-dimensional arrays of type `gene`. This is a useful abstraction, provided one remembers that chromosomes are organic molecules that tend to interact with their environment; the environment in this case is a cell nucleus containing (in the human case) 45 other chromosomes. During cell division, chromosomes pair up and align. When this happens, pieces of the DNA can exchange places as shown in Figure 3.1.

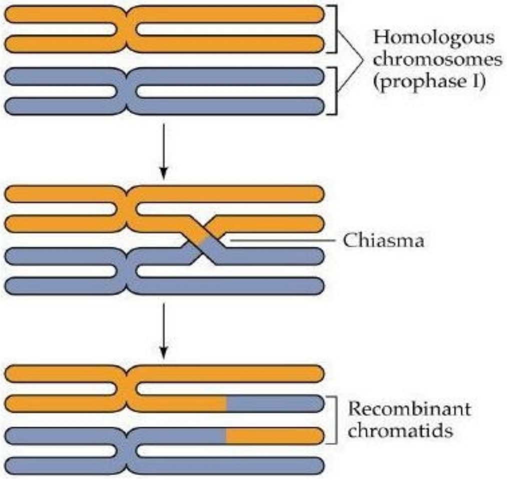


Figure 2.1: Crossover [http://www.biology.duke.edu/noorlab/pix/crover.jpg]

The result is a pair of chromosomes, both of which are different from both parental versions. If the crossover happens in the gonads, which produce egg and sperm cells, the new versions can be passed on to the offspring and enter the general population. The mechanism of crossover allows a much richer variety of chromosomes in the species than if parental chromosomes were always passed on to offspring intact. So crossover produces variety in populations, and GAs mimic crossover to produce variety in their metaphorical populations.

The second mechanism exploited by GAs is mutation. Information-bearing nucleic acids (the As, Cs, Gs, and Ts) can undergo chemical reactions and turn into other nucleic acids. More complex reactions can insert DNA strands into, or delete strands from, a chromosome. Some mutations are innocuous, while others produce a new allele, possibly with completely new traits. GAs generally model substitution mutations, and leave insertions and deletions alone.

When GAs are applied to the MSA problem, there is risk of confusion regarding the meaning of mutation. An MSA's consensus is a guess as to the nearest common ancestor of the data. This ancestor would have evolved via all possible mechanisms, and an MSA points out insertions, deletions, and substitutions. However, the GA that computes the MSA uses *metaphorical* genes that undergo *metaphorical* mutations (but only metaphorical substitution mutations). The ambiguity arises because in this case the problem domain is the metaphrand of the GA's own metaphor. Any ambiguity in this paper is the fault of the author, who will do his best to be precise.

Survival of the fittest is an oversimplification of Darwinian evolution. What really happens is that some mutations confer benefit in the form of increased likelihood of surviving to maturity and producing offspring. These offspring have a chance of inheriting the mutant gene and manifesting its corresponding trait, so that they too are more likely to breed. Eventually the trait might spread throughout the population.

With this understanding of crossover, mutation, and survival, we can show how genetic algorithms work.

Genetic Algorithms: Data and Operators

The big idea of genetic algorithms is to parameterize feasible solutions to the problem at hand. If all the parameters can be represented in a 1-dimensional data structure of fixed size, the problem is a candidate for solution by genetic algorithm. (If the representation is multidimensional or of variable size, a GA solution might still be possible, though implementation will be complicated.)

In its simplest form, a GA represents points in solution space as arrays of bits. The algorithm begins by randomly generating a “population” of (typically) 100 such arrays, where each array represents one individual member of the population. The array can be thought of as the individual’s chromosome. At this point the algorithm enters a loop, where each iteration consists of:

- Evaluating each individual’s fitness
- Selecting individuals that will be allowed to “breed”
- Breeding to generate new individuals
- Mutation
- Merging the newly-bred offspring to create the next generation

As GAs were originally proposed [15], the evaluation formula (or *fitness function*) is the only component of the algorithm that contains knowledge about the problem domain. It is here, and only here, that the 0s and 1s in the individual are treated as having domain-specific meaning.

Numerous options are available for determining the breeding pool. Typically a weighted roulette wheel technique is used. With this technique, an individual’s probability of participating in a breeding event is proportional to its relative fitness. Relative fitness is normalized so that the sum of all normalized fitnesses is 1, and a conceptual roulette wheel is modeled. Each individual is represented by a wedge of the wheel whose fractional size is the normalized relative fitness. The wheel is “spun” twice, once for each parent, and the winners breed to create new individuals.

Breeding is modeled by the crossover metaphor. The simplest crossover function chooses a location on the chromosome. All data bits from that point on are swapped between the two mating individuals. Note that two offspring are generated per mating event. More sophisticated schemes involve multiple crossovers.

After the new individuals are generated, they undergo mutation: one or more bits on the chromosome have a small but non-zero chance of being inverted.

There are numerous options for determining membership in the next generation. All new offspring are retained (otherwise there is no point in generating them). The next generation can consist purely of offspring, or, under a policy known as *elitism*, some of the fittest members of the previous generation move verbatim to the next generation [23].

The process continues until some criterion is met, at which point the highest scoring individual ever generated is returned as the solution. Termination can be caused by a timeout, or when some desired fitness score is reached, or when the software detects zero or insignificant score improvement over some number of generations.

Chapter 3: Literature Review

In this chapter I review past and current literature on the general topic of Genetic Algorithms, and on their specific application to the MSA problem. These works fall into three categories: explorations of theory; reports of successful applications of GAs in a wide variety of problem domains; and several papers on the applications of GAs to MSAs.

Literature on Genetic Algorithm Theory

Literature on GA theory begins with a single seminal book by John Holland [15], published in 1975. Neither its title (*Adaptation in Natural and Artificial Systems)* nor its subtitle (*An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*) hints that the book is really a proposal for a radically new family of algorithm: a family that is both probabilistic and heuristic in the extreme. The book is highly theoretical in nature, with deep mathematical analysis; yet it is the basis for a huge body of applications, spanning more than a quarter-century, in many problem domains (e.g. [11, 18, 20, 22, 23, 27, 28]). Once the GA became established, other references (e.g. [19, 23, 24]) provided practical implementation guidelines based on experience.

The gist of Holland's proposal is the ecological analogy: "It's often useful to think of the environment … in terms of environmental niches, each of which can be exploited by an appropriate set of phenotypic characteristics." [15]. Competing solutions in a population should be represented as binary strings. These strings are generated by, and later serve as input to, various *operators*. Here *operator* means a function that implements crossover, mutation, or selection for breeding. The theoretical basis for the validity of the GA concept is the Schema Theorem, which considers wildcard binary strings on the alphabet {0, 1, *}. Since the * character can represent 0 or 1, a string with many *s represents a large family of values. Such a string is called a *schema* (pl. *schemata*). The Schema Theorem identifies schemata with hyperplanes that partition the multidimensional solution space, and states that "schemata with above-average fitness (especially short, low order schemata), increase their frequency in the population each generation at an exponential rate when rare." ([2]).

While theoretically useful, the schema metaphor has its limitations. One reason for this seems intuitively clear: a good metaphor should be easy to visualize, but a hyperplane is *impossible* to visualize! Of the recent theoretical works that I surveyed [2, 3, 8, 17, 24, 25], one [24] briefly mentions schemata while the rest make no mention of them. Of the "success story" articles that I review in the next section, none uses the Schema Theory to justify the chosen approach.

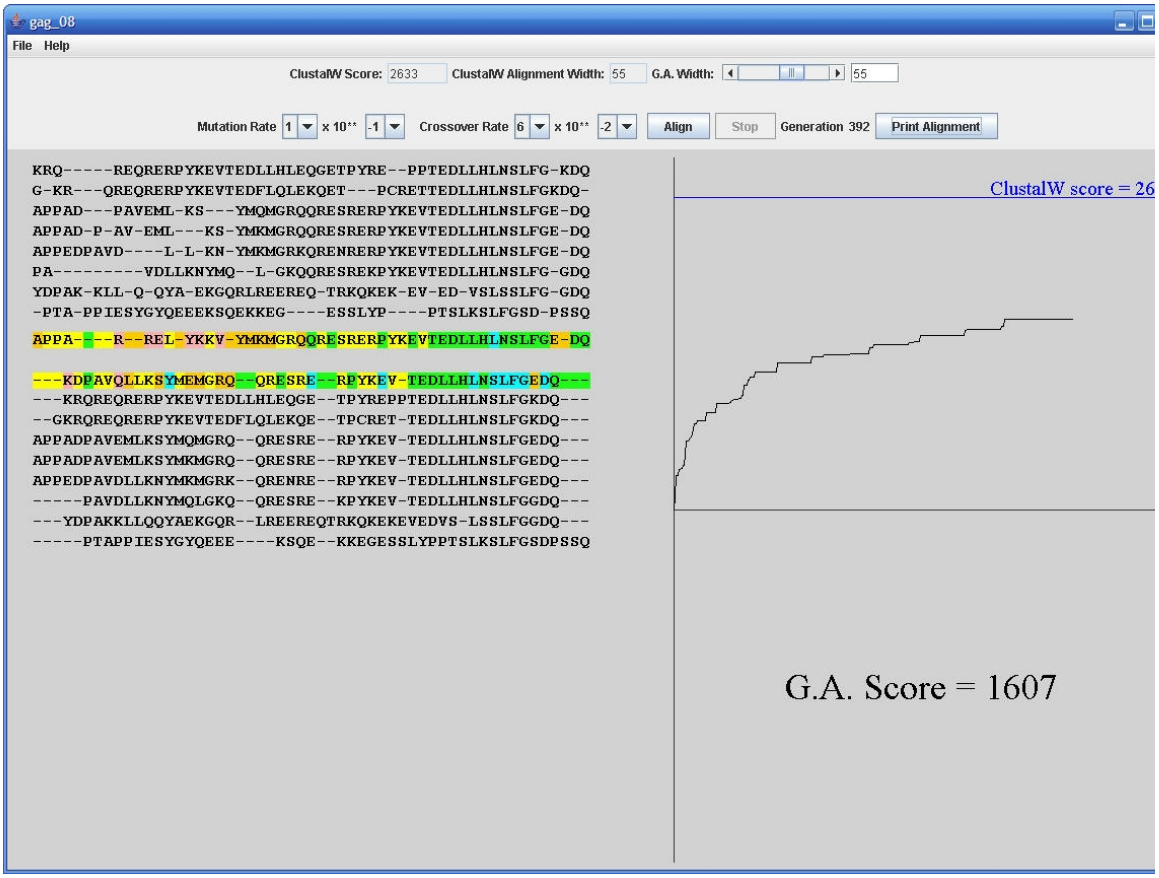
A more useful metaphor is that of hill climbing. For a problem with N parameters, N-dimensional solution space is imagined as an (N+1)-dimensional terrain, where the extra dimension is the fitness score. The goal of the GA is to explore this space so as to optimize fitness - in other words to climb as close as possible to the top of the highest possible hill. If we consider only 2 of the N parameter dimensions, we have a metaphor that can fit in our 3-D imaginations. [5] and [25] make especially good use of this metaphor. To extend the idea, the GA's initial random population can be thought of as randomly scattered across the hilly terrain. Crossover implements exchange of information regarding worthwhile hills, while small favorable mutations implement small steps up the local hill. This explains the saying that Genetic Algorithms involve tradeoffs between "exploration vs. exploitation" [5]. Exploration is the choosing of hills, exploitation is the climbing of hills.

Modern common practice goes beyond Holland's initial proposal, where chromosomes were strictly binary strings, and mutation and crossover rates were constant. The most common variation on the original GA theme involves chromosome encoding. Holland called for chromosomes to be sequences of bits. The fitness function might (and frequently does) construe certain subsequences as numbers. However, the operators (mutation, crossover) are not aware of this structure. Thus within bits that represent a numeric value, mutation and crossover are equally likely at any position. For example, a mutation that flips a 2^{10} bit (effectively adding or subtracting 1K) is just likely as a mutation that flips a 2^{0} bit (effectively adding or subtracting 1). Some authors (e.g. [11]) recommend using GrayCode rather than binary representation, while maintaining Holland's bit-level independence. Others, including all those summarized in the next section, simply use an array of ints for their chromosomes. This imposes restrictions on the operators: crossover only happens on int boundaries, while mutation generally increments or decrements a value. In other words, the operators use knowledge of the format of the chromosome.

A second variation is proposed in [5] to solve a problem highlighted by [10]: premature convergence. In the hill-climbing metaphor, this happens when the entire population climbs the same wrong hill. More technically, the operators stop producing offspring that are better (or much better) than their parents. The return on investment for executing a generation diminishes too early. The solution proposed in [5] is to forbid mating between partners whose Hamming distance falls below some threshold. Note that with a numerical chromosome as above rather than a binary chromosome, the notion of Hamming distance must be modified.

A third technique, also intended to forestall convergence, is known as *tribes* or *islands* [M, N]. Initially, multiple independent populations ("tribes") are generated. These only breed internally (as if each is isolated on an island) until some criterion is met. Then the fittest individuals from each tribe are gathered into a breeding set, and execution continues as in the standard algorithm.

One last technique, *adaptation* ([5, 23], can also be seen as a way to postpone convergence. Adaptation is easiest to understand with a diagram:



ClustalW Score: 2633 ClustalW Alignment Width: 55 G.A. Width: 55

Mutation Rate 1 x 10** -1 Crossover Rate 6 x 10** -2 Generation 392

```
KRQ-----REQRERPYKEVTEDLLHLEQGETPYRE--PPTEDLLHLNSLFG-KDQ
G-KR---QREQRERPYKEVTEDFLQLEKQET---PCRETTEDLLHLNSLFGKDQ-
APPAD---PAVEML-KS---YMQMGRQQRESRERPYKEVTEDLLHLNSLFGE-DQ
APPAD-P-AV-EML---KS-YMKMGRQQRESRERPYKEVTEDLLHLNSLFGE-DQ
APPEDPAVD----L-L-KN-YMKMGRKQRENRERPYKEVTEDLLHLNSLFGE-DQ
PA---------VDLLKNYMQ--L-GKQQRESREKPYKEVTEDLLHLNSLFG-GDQ
YDPAK-KLL-Q-QYA-EKGQRLREEREQ-TRKQKEK-EV-ED-VSLSSLFG-GDQ
-PTA-PPIESYGYQEEEKSQEKKEG----ESSLYP----PTSLKSLFGSD-PSSQ

APPA----R--REL-YKKV-YMKMGRQQRESRERPYKEVTEDLLHLNSLFGE-DQ

---KDPAVQLLKSYMEMGRQ--QRESRE--RPYKEV-TEDLLHLNSLFGEDQ---
---KRQREQRERPYKEVTEDLLHLEQGE--TPYREPPTEDLLHLNSLFGKDQ---
--GKRQREQRERPYKEVTEDFLQLEKQE--TPCRET-TEDLLHLNSLFGKDQ---
APPADPAVEMLKSYMQMGRQ--QRESRE--RPYKEV-TEDLLHLNSLFGEDQ---
APPADPAVEMLKSYMKMGRQ--QRESRE--RPYKEV-TEDLLHLNSLFGEDQ---
APPEDPAVDLLKNYMKMGRK--QRENRE--RPYKEV-TEDLLHLNSLFGEDQ---
-----PAVDLLKNYMQLGKQ--QRESRE--KPYKEV-TEDLLHLNSLFGGDQ---
---YDPAKKLLQQYAEKGQR--LREEREQTRKQKEKEVEDVS-LSSLFGGDQ---
-----PTAPPIESYGYQEEE----KSQE--KKEGESSLYPPTSLKSLFGSDPSSQ
```

Figure 3.1: Convergence

The graph at the right of the figure plots GA's score (vertical) against generation number (horizontal). The track is typical: strong early performance, followed by diminishing returns as convergence sets in. Since convergence is caused by falling variation among individuals, its is worthwhile [28] to detect the onset of convergence, and to counteract it by increasing crossover and mutation probabilities. This will boost the population's variation. Convergence sets in when all individuals are climbing the same hill, or very similar hills. Adaptation must happen soon enough, and must introduce enough variation, that some individuals are moved onto new and promising hills. There is no way to predict *a priori* what the curve-flattening threshold should be, or what the new increased mutation and crossover probabilities should be.

This concludes my literature survey on general Genetic Algorithm theory. Next I will summarize my review of works on successful applications of GAs to a variety of problem domains.

Literature on Genetic Algorithm Applications

I read a sampling of documents on GA applications to problems unrelated to MSA. My primary objective was to learn general hints regarding practical implementation. I also wanted to see if I could generalize the kind of problem that yields well to the GA approach.

My sampling of problem domains included Combinatorics [20, 22], Graph Theory [5, 27], Digital Signal Processing [23], Control Systems [23], and Speech Recognition [23]. In all cases but one, I found that the algorithms described were highly tuned to the problem domain; thus there was little I could gather by way of general implementation hints beyond the subtext: adapt the GA to the domain. This was especially true in the choice of mutation and crossover operators. (For instance, the Speech Recognition example in [23] uses a mutation operator that imposes a sonic distortion on a gene that represents a larger distortion. This operator is highly semantic, treating the gene's meaning rather than its 0/1 content.) The exceptional case, [20], used very little domain information; solution was in the spirit of a statement by the same author (Khuri) in a different source: "GAs make few assumptions about the problem domain and can thus be applied to a broad range of problems."

It would be easy to conclude that design of operators should take into account as much domain-specific knowledge as possible. (It seems there isn't a term for such operators; in this paper I call them *semantic operators*.) However, I believe overuse of semantic operators would be naïve. All the papers I read are documents of success. Failure stories don't get published; we have no data concerning GA projects that incorporated domain knowledge and subsequently weren't able to produce good results. Use of semantic operators seems a risk that should not be undertaken for its own sake. The downside of the risk is that the GA might actually become worse at exploring its terrain. That is, the sophisticated operators might impose too much restriction on exploration, thus accelerating convergence. I concluded that the best approach was to start with as few domain assumptions as possible. Domain knowledge should be incorporated only if it demonstrably improves performance. Use of sophisticated operators is an engineering decision that requires justification on engineering grounds.

As for generalizing the kind of problem that yields well to GAs, there doesn't seem to be any single type of appropriate domain. On the contrary, we find statements such as "… they have proven to be of notable usefulness in solving optimization problems of all kinds" [24] or "… well suited for finding global optima in complex search spaces." [5] However, my survey drove me to three conclusions, which the authors probably felt were so obvious that they didn't need to be stated:

1) Genetic Algorithms are heuristic and nondeterministic. As such, they should only be used when exact solutions are unavailable, inefficient, or impossible.
2) GAs should be applied to problems with large multi-dimensional solution spaces.
3) GAs should be applied to problems whose solutions can be cleanly mapped to linear genes.

These conditions are necessary but not sufficient. That is, if your problem has no exact solution and a large multi-dimensional solution space, and maps well to a gene, then you have a chance with a GA. If not, then you don't have a chance.

Literature on Genetic Algorithms Applied to Multiple Sequence Alignment

Two research teams (Notredame & Higgins [26] and Zhang & Wong [31]) have published significant results on the use of GAs for solving MSA problems. A third team (Horng, Lin, Liu & Kao) has published a minor result that is relevant to this paper. In this section I summarize the work of these teams.

The Notredame team created a program called SAGA ("Sequence Alignment by Genetic Algorithm"). SAGA's overall structure is quite complex. The authors state that "We found that a simple GA, applied in a straightforward fashion to the alignment problem, was not very successful". The gene representation is not described at all. Generation population size is 100. SAGA uses a rich set of operators, ranging from quite simple to sophisticated and semantic; the paper describes them broadly without going into detail. For example, one crossover operator is "uniform crossover". This usually means that the number of crossover points is unlimited, with all loci on the gene being equally probable independent candidates. However, the SAGA paper goes on to say that "exchanges are promoted between zones of homology." It is unclear how this is implemented, or what effect it has on GA execution. Other operators insert gaps, shuffle blocks, and perform local optimization.

SAGA uses “dynamic operator scheduling”, which seems to be synonymous with adaptation (which postpones convergence; see first section in this chapter). With dynamic operator scheduling, an operator’s likelihood of being applied is a function of its recent track record. Thus operator probability is constantly monitored and adapted.

SAGA uses several scoring functions. We can surmise that the intention is to allow meaningful comparison with other tools. For example, SAGA can mimic CLUSTAL-W’s range of scoring functions, so it is meaningful to compare optimum scores produced by the two algorithms on identical data sets. In all cases, columns are evaluated by pairwise comparison of non-gap members, with a standard matrix such as BLOSUM62 providing scores. Gaps are penalized via an affine function. Since I use both of these techniques in my own project, they will be described in the next chapter.

The paper reports results for two groups of test cases. In the first group, 9 data sets contained 4-8 sequences, with lengths of 60-296. (I assume that there isn't significant sequence length variance within the data sets.) Optimal alignments are known for all these sets. SAGA’s results were compared to those of MSA. (MSA is a freely distributed sequence alignment package that runs under UNIX. Unlike CLUSTAL-W, it is intended to be downloaded into the user’s computer. It uses limited resources, but has limited input size capacity.[21]) In all 9 cases, SAGA either matched MSA’s score or fell short by only an insignificant amount.

The second test group was intended to be more challenging: 4 data sets, containing 10-32 sequences of length 144-281, were aligned, and scores were compared to CLUSTAL-W scores. These data sets are extensions of those in the first group, but are large enough that MSA can't handle them. SAGA fell just short of CLUSTAL-W (1.2% in the worst case.)

What are we to make of SAGA? Clearly it is the product of extensive creativity. The lack of detail about representation and operators makes it difficult to take away specific ideas for new GA/MSA implementations. The authors evidently made a commitment to sophisticated semantic operators. The multitude of operators enabled dynamic scheduling, which might have been of great benefit. The fact that SAGA didn’t improve on established scores is not a detriment: the established scores are mostly known to be optimal, so SAGA can't possibly improve on them. It would be good to see more results, with data whose optimal scores are not known. After all, it is only with those data sets that new algorithms can hope to contribute improved scores. It would be good to see a demonstration that SAGA is not overspecialized for sequences whose optimal alignment is known or knowable.

The second major research effort is reported by Zhang & Wong in [31]. The authors divide the MSA problem into two steps. First, corresponding matching regions are identified to create what the authors call a pre-alignment. Second, the pre-alignment is converted to a finished alignment. Only the first step is implemented by GA; conversion is executed by a deterministic module.

Zhang & Wong's program (it has no code name) has a much simpler operator set than SAGA. There is a single crossover operator and a single mutation operator. Both are deeply semantic, and the paper provides extensive rigorous justification for their choice. There is no adaptation or dynamic operator scheduling. The scoring function is based on pairwise comparison of non-gap column elements; the scoring matrix is SG, a precursor of the BLOSUM matrices that are in current use [9]. Results are reported for 7 data sets of 2-8 sequences. Average sequence length in all cases was ~340 (notice that these are significantly longer than SAGA's test sequences). The GA scores were 20%-30% lower than the corresponding CLUSTAL-W scores, but the GA ran ~50x faster.

Zhang & Wong's program is clearly customized for fast execution on certain specialized kinds of input. As they say in their paper, "… when sequences have long match blocks, the genetic algorithm is more advantageous than CLUSTAL-W in terms of efficiency". Again, as with SAGA, we have a promising result, but we should not jump to premature conclusions. The timing results are spectacular. On the other hand, it isn't clear how much of the efficiency is contributed by the GA, which is only one component of the overall algorithm. Moreover, the mean execution time went from ~30 seconds (CLUSTAL-W) to ~.8 seconds (Zhang & Wong). But 30 seconds is a tolerable delay, especially given CLUSTAL-W's higher scores. Score, not execution time, is the valuable commodity.

The last paper in my survey is by Horng, Wu, Lin, & Yang [17]. It describes an extremely simple gap-oriented approach. Finding this paper was a great disappointment to me, because I came up with the same idea last year (2008), while working on a GA/MSA project for a graduate Bioinformatics class. Horng et al got there first; they use a chromosome that simply contains gap locations. Since my project uses the same chromosome design, a detailed explanation is deferred until the next chapter.

Horng et al only aligned nucleotide sequences. This is a much simpler problem than protein alignment, since the alphabet size is 4 instead of 20. The scoring function is simply a count of identity columns, with affine gap penalties. A variety of semantic operators are used: gaps can be merged or broken up. Test sequence length ranged up to 8810 (compare to <= 340 for the other two papers). For 7 data sets, scores either matched CLUSTAL-W's scores or fell short by only a few percent.

The coincidence of results with CLUSTAL-W's scores suggests that, like the SAGA team, this team chose sequences for which CLUSTAL-W provides near-optimal scores. Otherwise there is no way to explain the proximity of results, or the lack of scores that significantly out-performed CLUSTAL-W.

This section has reviewed the 3 known papers on using GAs to solve MSA problems. We can conclude that it can be done, that algorithms across a wide range of sophistication (from the simplicity of Horng et al to the richness of SAGA) have produced publishable results, and that no GA has yet significantly beaten CLUSTAL-W. In the next chapter I describe my own GA design.

Academic publications are not intended to be introductory primers for novices. In all my readings, there was no explicit advice as to how to develop a GA. However, I found one unintended nugget of great value in the SAGA article [26]. In a section on mutation sites, we read, "Experience shows that, while monitoring the search, areas containing gaps are …" The opening clause implies that the SAGA team monitored the progress of their algorithm's search, learned from experience, and adapted their algorithm accordingly.

Chapter 4: Implementation of MSG

In this chapter I describe the major implementation features of my program, which I call MSG. The name is an acronym for “**M**ultiple **S**equence, **G**enetic”. The implication is that, like the flavor enhancer, my software enhances the alignment experience, although it is known to cause headaches and raise blood pressure.

Chromosome

I am not the first developer to represent gap locations in a Genetic Algorithm Chromosome. Horng et.al. published results in 2005. However, they only tested their GA on nucleotide sequences, and their results matched or fell just short of Clustal-W (see [17]). Encouraged by my own prior work in this area, I decided to explore applications of gap-based MSA GAs to small test sets of protein sequences. (The size restriction was forced by the available computing power.)

A gap-based chromosome specifies gap locations in an alignment. The chromosome is an array of integers whose length is the number of gaps. This is a simple and elegant structure, but it has a weakness: the information in a chromosome is with respect to the alignment width. To see why this is so, let n = the number of sequences in a data set and w = a feasible alignment width (i.e. at least as wide as the widest sequence). Let k = the total number of characters in all the sequences. Then the number of gaps = chromosome length = nw – k. Thus a chromosome length is only applicable to a given alignment width.

There is no obvious way to use a fixed-length chromosome to represent a variety of alignment widths. Notredame et. al. and Zhang et.al. didn’t need to overcome this problem, because their GAs were not gap oriented. The Horng et. al. paper does not mention their solution. My own solution is to push the width issue to a layer of the software that is unrelated to the chromosome design. In MSG, all chromosomes at any moment are all the same size, and describe alignments that are all the same width. (The width solution is described below, in the section *Alignment Width*.)

With a fixed alignment width w, the number of gaps to be inserted into each sequence is also fixed: it is simply w minus the sequence width. If sequence S_i is the length of the ith sequence, then (w - S_i) gaps are to be inserted into that sequence. The chromosome locations that describe those gaps are easily determined by simple addition. Figure 4.1 shows the chromosome layout for a simple example.

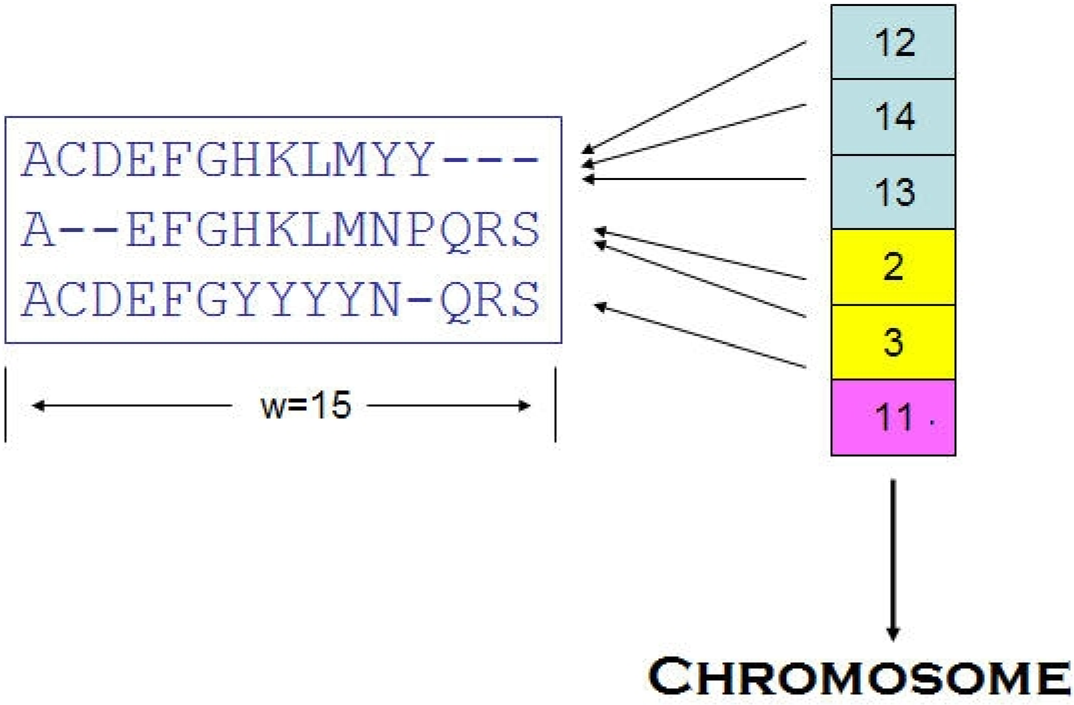


Figure 4.1: Chromosome Layout

In Figure 4.1 the first sequence needs 3 gaps, the second needs 2, and the last needs 1. Overall 6 gaps must be placed, so the chromosome's length is 6. The alignment width is 15, so gaps can be located in any column from 0 through 14. Gap locations in the chromosome are not required to appear in ascending order: note that "14" precedes "13". In fact, the locations for any sequence are not even required to be unique. Interpretation software described below (see "Fitness Function" section) ensures that for any sequence, any combination of gap locations can be interpreted meaningfully.

Chromosomes are implemented as a Java class that can initialize itself to a random value, participate in crossover and mutation, and compute its own fitness score. The class has a full implementation of the Java hashing and equals() contracts [12]. Standard comparison is by fitness score; sorting collections such as TreeMap and TreeSet are thus able to store and retrieve in fitness order in O(log n) time [6].

Fitness Function

As soon as a chromosome is generated, its fitness is computed. Fitness is the sum of all column scores, minus all gap penalties.

Before column scores and gap penalties can be computed, MSG must convert the linear chromosome to a 2D array of characters. This process would be straightforward, except for the problem of duplicate gap locations. The chromosome and operator definitions allow a gap location to appear multiple times in the chromosome's gap-location block for any sequence. The gap locations must be translated into meaningful information. If a chromosome dictates that a sequence has x gaps at location y, MSG interprets the allele to mean that the sequence contains a run of x gaps, centered on location y. If this run overlaps another gap or run, the gap or run is assimilated so that the merged run has appropriate length.

After gap locations are resolved, all other locations in the alignment must contain amino acid characters. These are assigned linearly, sequence by sequence.

Column scores are computed by pairwise application of the BLOSUM62 matrix. This popular scoring technique [14] reflects the likelihood that any amino acid can be substituted by any other. Substitutions between acids with similar chemical properties are more likely than those between acids with different properties, as the former are less likely to have much effect on the overall shape or charge of the protein, while the latter could alter protein structure enough to destroy function. Species receiving the latter kind of mutation are unlikely to survive to pass on the mutation to offspring. As an example of the BLOSUM-62 matrix, the value for glutamic acid vs. lysine is +1. Both amino acids are hydrophilic. The value for glutamic acid vs. alanine, on the other hand, is -1; alanine is hydrophobic, with significantly different chemical properties from glutamic acid. Other matrices besides BLOSUM62 are available [3]: for example, higher-numbered BLOSUM matrices can be used for highly similar sequences, and vice-versa. However, without prior knowledge of the nature of the sequences, BLOSUM62 is a good general-purpose matrix.

To compute a column score, MSG simply adds the BLOSUM62 scores of all pairs of amino acids in the column. Pairs that include at least 1 gap are not considered. Consequently, columns that contain only gaps do not influence the score.

Gaps are deleterious to an alignment's score in two ways. First, they don't contribute to a column's score. Second, they carry their own penalty. A gap of length g in a sequence is assessed an affine penalty of 11 for the first column of the gap and 1 for each subsequent column. (This formula is the same one applied by Clustal-W.) However, gaps aren't counted if they come from a column that consists entirely of gaps.

The fitness function of a chromosome is easy to compute. Unfortunately, it is $O(n^2m)$, where n is the number of sequences and m is the alignment width. The n-squared complexity derives from the comparison of every non-gap member of a column to every other non-gap member of the same column. With n sequences, a column with no gaps requires (n-1) + (n-2) + … + 1 comparisons, and this introduces the $O(n^2)$ term. The practical consequence is that on my available hardware, data sets of up to 5 sequences were processed efficiently, those with 6 - ~8 sequences were sluggish, and those with > 9 sequences were prohibitively slow. The m complexity term is also significant, as performance degraded severely with alignment widths >~ 60.

Breeding

MSG uses a standard roulette wheel with elitism [19, 23] to determine which members of a generation are allowed to breed.

The elitism policy promotes the fittest member of a generation verbatim into the next generation. MSG actually promotes the 2 fittest members, because this policy leaves an even number of offspring to generate.

Imagine a roulette wheel where the slots subtend differential angles. Slots of larger angle will enjoy higher probability, with probability that is proportional to the angle. This is a metaphor that is frequently seen in genetic algorithms for determining which chromosome in a generation will contribute to the next generation. Since MSG uses elitism to choose 2 members of the next generation, 98 members are to be created by breeding. Each member of the current generation has a chance of breeding that is proportional to its fitness score. If there really were a roulette wheel, each member of the current generation would correspond to a wheel slot whose angle is large or small depending on the member's relative fitness. The metaphorical wheel is spun twice to select two parents. Two offspring are generated by applying the crossover and mutation operators. These offspring are added to the next generation, and the process continues until the next generation contains 100 members.

Each site on the chromosome except the first is eligible to be a crossover site. An eligible site has a certain probability p of actually hosting a crossover event. MSG uses an initial value of p = 7%. As described below in the *Operator Adaptation* section, the crossover probability is varied according to past performance of the algorithm. Initially, one offspring is identical to the "mother" parent while the other offspring is identical to the "father" parent. ("Mother" and "father" are arbitrary designations for distinguishing between the two parents.) At a crossover site, the father's chromosome values are copied into the child that initially resembled the mother, and vice versa. At the next crossover site the sources are switched, and so on.

After crossover, the mutation operator is applied. The initial mutation rate is 20%, meaning that any site in either child has a 20% probability of turning into any feasible value. As with crossover rate, mutation rate is subject to adaptation. The initial rate seems quite high, but is necessary to prevent premature convergence. Lower rates seem to have little effect. This is probably because many mutations are synonymous due to gap overlaps.

Once mutation has occurred, a chromosome is fully “born”. Its fitness function is computed, it is added to the current generation, and the cycle continues.

Operator Adaptation

Operator adaptation [28] has already been described theoretically in the *Literature Review* chapter. I was at first hesitant to implement adaptation, as it seemed to be one more feature in a program that was already feature-rich. However, experience with running early revisions of MSA on various data sets taught me that different data requires different crossover and mutation rates. If the rates are too low, new positive traits are unlikely to appear; if the rates are too high, traits don't persist long enough to become established in the population. Moreover, I had no scientifically-based criterion for choosing a hardcoded rate. I realized that I am better at writing adaptation software than I am at guessing crossover and mutation rates.

I therefore chose initial rates more or less arbitrarily. As described above, the initial crossover rate is 7% and the initial mutation rate is 20%. A population monitors its own top fitness score over time. If this remains constant over 20 generations, both operator probabilities are incremented by 10% of their current value. If the curve rises too steeply (>10% net increase over 20 generations), both probabilities are decremented by 10% of their current value.

The latter adjustment might seem counterintuitive. Why apply the brakes just when the machine is really speeding up? I have no rigorous explanation, but MSG definitely achieves better scores when operator rates can decrease as well as increase. Perhaps it is best to look at adaptation as a kind of thermostat. If rates can only increase, the only feedback in the system is positive. The system eventually becomes unconstrained, and is unable to converge on a good solution. With negative feedback, scores can still improve significantly, but the rate of evolution is slow enough that new helpful traits can become established in the population.

Tribes and Jumpstarting

MSG maintains multiple independent concurrent tribes, as described above in the *Literature Review* chapter and in [4, 30]. The user may select 10 or 20 tribes.

Implementation of this feature was straightforward, and well worth the effort. Tribes execute independently for 1000 generations. The top (100/number-of-tribes) performers of each tribe are then combined to form a merged population which runs for an additional 1000 generations. Naturally, execution slows by a factor of 10x or 20x during the tribal phase. However, this is adequately compensated. Without the 1000-generation merged phase, running once with n independent tribes is equivalent to running n times with a single tribe. I found in my original work in this field that it takes 10-20 single-tribe runs to get a good result. Thus running with multiple tribes saves me the effort of manually restarting the software and keeping track of the best result.

Tribe provided a second benefit. While it was usually the case that the merged phase was not able to improve on the best chromosome's score, in a few cases the post-merge improvement was dramatic.

I have Prof. Khuri to thank for suggesting that I "jumpstart" my population from the alignment produced by Clustal-W. I was initially reluctant, because I was concerned that the GA might have trouble exploring far from the Clustal-W solution. However, the tribes feature provided a perfect way to jumpstart: one tribe out of 10 or 20 is initialized with the Clustal-W solution (1 verbatim chromosome and 99 mutated ones); the other tribes are initialized randomly. Thus MSG is able to leverage the Clustal-W solution while retaining 90 or 95% of its original exploration resources.

Jumpstarting presented an "impedance-mismatch" problem. The GA requires as input the desired alignment width. If this is exactly the width of the Clustal-W alignment, there's no problem. But for other widths, the Clustal-W solution must be widened or narrowed. This cannot be done arbitrarily, as the chromosome for the jumpstarted tribe must be the same length as the chromosome for the other tribes. (Otherwise the crossover operator cannot be applied; the situation is analogous to trying to interbreed two species.) The problem is to produce an alignment that most closely resembles the Clustal-W solution¸ given a specified alignment width (to match the width for which the GA is computing), and a specified number of gaps (to match chromosome size). There are 3 cases:

1) Desired alignment width = Clustal-W alignment width. Nothing needs to be done.
2) Desired alignment width > Clustal-W alignment width. The Clustal-W alignment can be adjusted by adding columns of gaps at the beginning and end of the original alignment.
3) Desired alignment width < Clustal-W alignment width. Now it is necessary to remove the correct number of columns, while simultaneously removing the right number of gaps.

The solution to Case 2 always inserts the correct number of gaps, so chromosome compatibility is automatic.

Case 3 is far more difficult. Prof. Taylor pointed out that the problem is equivalent to the 0-1 Knapsack Problem, where the knapsack must be filled exactly to capacity. (Imagine that as columns are removed, they are put in the knapsack. Knapsack capacity corresponds to the number of gaps to be removed, and columns correspond to indivisible objects.) The problem should therefore yield to a Dynamic Programming solution. However, on further investigation I found that the problem is more complicated than it appears. Not all feasible solutions are equally valuable. I want to jumpstart the special tribe with the best "Clustal-inspired" alignment I can find. I therefore used a randomized algorithm that generates 10,000 sets of columns uniformly at random. The number of columns is an input. Sets whose columns contain the right number of gaps are feasible solutions. The feasible set that produces the highest fitness score is chosen, and those columns are deleted to produce the trimmed-down Clustal-inspired chromosome. That chromosome is added verbatim to the jumpstarted tribe, and then mutated 99 times to fill out the population. If no feasible solution can be found in 10,000 tries, the algorithm abandons jumpstarting, and the 10th/20th tribe is randomized just like the others; this never happened in practice.

My randomized solution for case 3 runs quickly, and is only applied <= 21 times per data set (once for each alignment width), so even if it were inefficient the impact would be light. However, a more robust solution might be required for larger data sets. Please see Chapter 6 (*"Conclusions"*) for further discussion.

Top-Level Flow

MSG's chromosome strategy requires specification of alignment width. My protocol, which is described in detail in Chapter 7, calls for the user to perform an initial Clustal-W alignment. MSG uses the alignment to jumpstart the Clustal-W tribe and to determine the initial width.

At its top level, MSG searches alignment-width space for a feasible width that supports the best score. A width is considered feasible if it is >= the width of the longest sequence in the data set. MSG tries to search a range from (Clustal width – 10) through (Clustal width+10). If (Clustal width – 10) is not feasible, then the range starts at the minimum feasible width. The range is not searched in order. I found that a better intuition for the nature of the search space is obtained when MSG first computes the bottom of the range, then the top, then the Clustal width, and finally all other widths in increasing order.

The overall top-level flow of MSG is given in Figure 4.2:

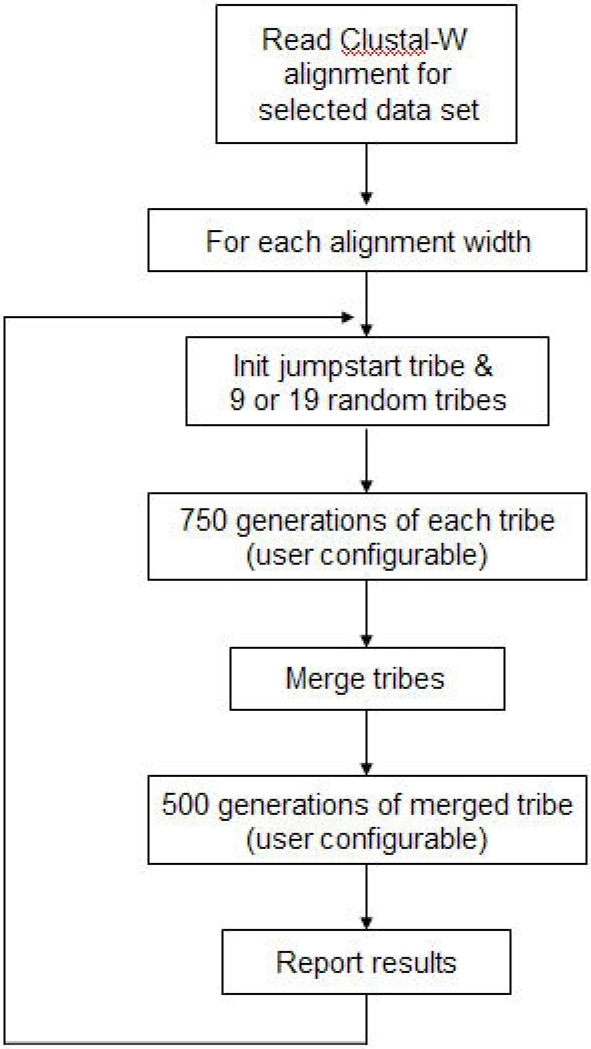

Read Clustal-W alignment for selected data set
For each alignment width
Init jumpstart tribe & 9 or 19 random tribes
750 generations of each tribe (user configurable)
Merge tribes
500 generations of merged tribe (user configurable)
Report results

Figure 4.2: MSG Flowchart

Wider Search Space and the Akaike Criterion

MSG's +/- 10 range was sufficient to find local optima for all the data sets that I tried. That is, there was a score peak somewhere in the interior of the range. Scores distinctly fall off for widths that are far from the optima, suggesting that the optima are not only local but global.

However, the data sets were all trimmed to width ~= 50, so that they would run in reasonable time. A more general algorithm would handle arbitrary width. As sequence width increases, it becomes decreasingly likely that the +/- 10 range will contain the optimum solution. Moreover, it is possible that larger data sets don't present the single-global-optimum profile that I observed with my smaller data sets. In other words, it might be valuable to explore width space far from the Clustal width.

Prof. Bremer suggested a statistically meaningful strategy for searching a broader swath of width space, where exhaustive computation is not feasible. The Akaike Information Criterion (AIC) method [1, 16] is an approach for sampling parameterized space, where the parameter bears a penalty. With MSA, the space parameter is the alignment width. Width is penalized explicitly (compared to narrower solutions) by the inclusion of gap penalties. Moreover, we intuitively feel that, all other things being equal, a narrow alignment is superior to a wide one. The Akaike Criterion formalizes this intuition according to the formula AIC = 2k – 2 ln(L). In our case k is the score obtained by the GA for alignment width L. The width that optimizes AIC is the one to be chosen.

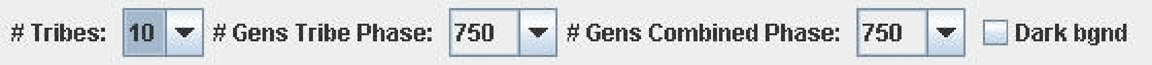
An enhanced version of MSG might search widths up to 2x the Clustal width, which for entire protein alignments could easily exceed 1000. Solving for each possible width is not feasible. Prof. Bremer suggested solving for an evenly-spread sample (perhaps every 10 widths), and choosing the solution with the best AIC score.

As it turned out, there was no opportunity to implement AIC, because on available hardware MSG runs prohibitively slowly with alignment width >~ 50. However, the AIC approach would be an excellent enhancement if MSG were to be ported to hardware that can handle wide inputs.

Graphical User Interface

MSG’s graphical interface provides simple controls and sophisticated visualization for the genetic algorithm. In the terminology of Java’s SWING toolkit [13], the model is complex but only requires simple user interaction; hence the controller can be simple, and is built from standard components. The view, on the other hand, needs to be rich enough to give users an intuition for what the model is doing; hence it is custom built.

MSG’s control panel looks like this:



Tribes: 10 ▾ # Gens Tribe Phase: 750 ▾ # Gens Combined Phase: 750 ▾ □ Dark bgnd

Figure 4.3: MSG Control Panel

The user may choose between 10 or 20 tribes. In practice, 20 proved superfluous: 10 tribes were always enough to exceed the Clustal-W score. The next two drop-down choices allow the user to specify the number of generations before and after tribal merging. 750 was always adequate. The next-lower option, 500, was nearly always adequate, but sometimes more generations were needed for the score curve to flatten out.

The last control is cosmetic. The background of the tribes panel (described below) can be very light blue for printing, or black for easy reading during dramatic projection.

In addition to this control panel, MSG has a very simple File menu via which the user selects the desired data set.

Directly under the contol panel is a progress panel, as shown in Figure 4.4.

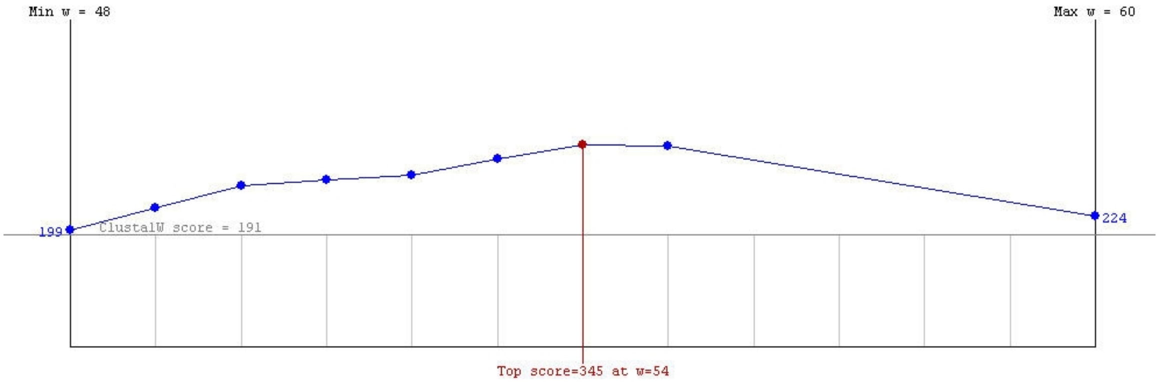
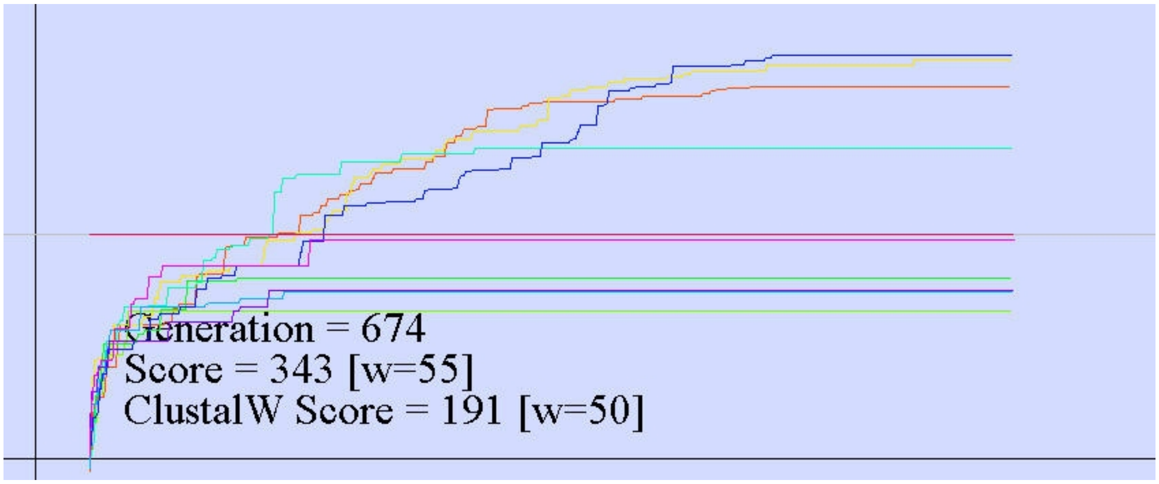


Figure 4.4: MSG Progress Panel

The progress panel can be thought of as a visualization of MSG's main loop, which applies the genetic algorithm on a range of feasible alignment widths. The panel shows MSG's scores for the various alignment widths. The Clustal-W score appears as a light-blue horizontal line. The best MSG score is marked in red. The figure shows the middle of a highly successful run on the BB11001 data set: MSG has exceeded Clustal-W for all alignment widths so far (48-55, and 60). Eventually MSG will compute scores for widths 56-59, and the overall shape of the curve will provide an intuition for the optimal alignment width.

Directly under the progress panel is a tribes panel, as shown in Figure 7.3.



Generation = 674
Score = 343 [w=55]
ClustalW Score = 191 [w=50]

Figure 4.5: MSG Tribes Panel

The tribes panel provides real-time information on the evolution of the tribes. Each tribe appears in a different color. As generations are executed, the colored traces extend to the right. Eventually they all flatten out, indicating that they have found local maxima from which they cannot escape.

The faint horizontal line in the middle of the figure represents the Clustal-W score. Note that one tribe, traced in red, coincides exactly with this line. The red tribe is the one that is "jumpstarted" with the Clustal-W result. With most data sets, the red line remains perfectly flat or shows slight increase, indicating that the Clustal-W solution is a local maximum. The result is not surprising: we expect Clustal-W to optimize, at least locally.

After the merging of the tribes, the 10 colored traces are replaced by a single trace that indicates the evolution of the one merged tribe.

Chapter 5: Data, Protocol, and Results

In this chapter I describe data I used to test MSG, the protocol I used to control the testing process, and the results I obtained.

Data

I initially intended to test MSG on protein sequences taken from the BAliBASE benchmark suite [29]. These are publicly available at http://www-bio3d-igbmc.u-strasbg.fr/balibase. BAliBASE sequences are selected for various properties that stress various aspects of MSA and Phylogenetic Tree creation software. The suite is intended for use by software developers, to benchmark the performance of new alignment programs.

The BAliBASE suite is divided into reference groups, which are numbered 1-5. Each data set has an i.d. whose first digit is the reference group number. Sets in Group 1 are mutually equidistant. Those in Group 2 include a highly divergent orphan sequence. Sets in Group 3 consist of several related subsets, with low identity among subsets. Group 4 contains sets with N/C terminal extensions, and Group 5 sets have internal insertions. Thus the suite exercises a solid range of real-life situations.

Unfortunately I ran into some practical limitations. With >8 sequences, or with sequence length >~ 50 characters, CPU performance dropped off sharply. I suspect that in both situations all available RAM was used, and the system spent too much time page swapping. I therefore used subsets of certain BAliBASE data sets, reducing both sequence count and sequence length to manageable sizes. I cannot assume that the essential nature of the sequences survived the size reduction. For example, in trimming down the sequences from a Group 2 set, it is possible that I removed the most divergent part of the orphan. However, I still had useful data in a useful format. I used one set from each of Groups 1, 3, and 5. The exact sequences that I used are listed in Appendix A.

I supplemented the BAliBASE data with sequences from 4 other sources. I reused the GAG and POL sets on which I tested my original CS223 project. I used some Platelet-Derived Growth Hormone (PDGH) sequences, thanks to Wendy Lee. Lastly, I built a random protein generator and used it to create four data sets with low to high sequence length variance. Table 5.1 summarizes the data sets.

Name	Source	Description	# of Sequences	Longest Sequence
BB11001	BAliBASE	Mutually equidistant sequences	4	48
BB30020	BAliBASE	1 highly divergent orphan	8	41
BB50001	BAliBASE	Internal insertions	8	42
Gag_04	HIV & SIV	Viral CDS	4	47
Gag_08	HIV & SIV	Viral CDS	8	45
Pol_04	HIV & SIV	Viral CDS	4	35
Pol_08	HIV & SIV	Viral CDS	8	35
PDGH_First	Vertebrates	Platelet-Derived Growth Hormone	7	50
PDGH Middle	Vertebrates	Platelet-Derived Growth Hormone	4	50
PDGH Last	Vertebrates	Platelet-Derived Growth Hormone	6	50
Random Uniform	Random generator	All same length	6	50
Random Light Skew	Random generator	Small Δ length	6	50
Random Medium Skew	Random generator	Medium Δ length	6	50
Random Intense Skew	Random generator	Large Δ length	6	50

Table 5.1 – Data sets

Protocol

I used the following protocol:

1) Original sequences were obtained in FASTA format.
2) FASTA data sets were pasted into the EBI Clustal-W web page at http://www.ebi.ac.uk/Tools/clustalw2/index.html, and remotely aligned.
3) I visually inspected the alignments from Step 2 to select interesting regions that were small enough to be analyzed by MSG on my computer.
4) The interesting regions were extracted by software, gaps were removed, and the sequences were converted to FASTA format and stored on my local hard drive.
5) The sequences from Step 4 were once again pasted into Clustal-W for alignment. The aligned sequences were pasted into text files on my local hard drive. These files were the inputs to MSG.

The double Clustal-W alignment steps might seem redundant, but they served very different purposes. The first alignments consisted of the original data sets, which would be too large for MSG. Inspection of these alignments told me which ones could be reasonably retained, and which portions should be retained. My criteria were broad: I just wanted noticeable conservation without full-on identity, and no regions in which a sequence consisted mostly of gaps.

The second alignment of Step 5 ensured that no matter what columns of what sequences I removed, the inputs to MSG would still be aligned as well as possible. This step was necessary because optimality of alignment is not linear: a subset of a good alignment is not necessarily well aligned.

Results

MSG produced consistently good results. I expected my results to be a moderate improvement on my original CS223 project, in which a much simpler genetic algorithm matched or beat Clustal-W's score in 3 out of 6 tests. In fact, MSG exceeded the Clustal-W score for each of 14 data sets. Table 5.2 presents comparative scores for MSG and Clustal-W.

Name	MSG Score OF msg solution	MSG SCORE of clustal-w SOLUTION	clustal-w SCORE of clustal-w SOLUTION *
BB11001	336	191	166
BB30020	4261	4073	4857
BB50001	2906	2702	3150
Gag_04	956	876	974
Gag_08	2986	2689	2679
Pol_04	324	83	53
Pol_08	1206	475	394
PDGH_First	228	139	n/a
PDGH Middle	79	10	n/a
PDGH Last	1980	1915	n/a
Random Uniform	209	-499	n/a
Random Light Skew	275	-132	n/a
Random Medium Skew	269	-109	n/a
Random Intense Skew	-10	-51	n/a

* Where available: Clustal-W does not always report scores for its solutions.

Table 5.2 - Scores

Chapter 6: Conclusions

In this chapter I draw conclusions from the results, interpret the results, evaluate the development process, and propose directions for further research.

General Conclusions

MSG clearly needs further development (see *Future Directions* section below) before it can be considered robust and general, but the early results are quite good. To my knowledge there are only 3 papers that report results in applying Genetic Algorithms to the MSA problem. Two of these [26, 31] do not use a gap-oriented chromosome, and their GAs require extensive pre- and/or post-processing. The one project that does use a gap-oriented chromosome [17] is for nucleic acid sequences rather than amino acid sequences; they were not able to exceed Clustal-W scores.

Apparently MSG is a new idea, combining the power of jumpstarting with the efficiency of a gap-based chromosome representation to produce improved multiple sequence alignments.

Interpreting the Results

As Table 5.2 shows, MSG is consistently and significantly successful with respect to Clustal-W. A responsible sense of skepticism prompts several questions: Is MSG really *that* good? What accounts for MSG's quality? And are the results typical of performance on other data sets?

First, regarding whether MSG is as good as it seems, readers should be warned against reading too much into Table 5.2. In particular, percentage score differences on the same data between MSG and Clustal-W do not represent quality differences between the two algorithms. For example, MSG is not 700% better than Clustal-W on the "PDGH Middle" sequences. The scoring function does not have a zero baseline (for example, both results for "Random Intense Skew" are negative), so percentages or fractional differences are not applicable. Nonetheless, Table 5.2 shows significant score differences, even if the exact degree of significance is hard to measure.

It should also be pointed out that MSG uses its own fitness function, which is not identical to Clustal-W's. Obviously Clustal-W optimizes alignments according to its own function. I suspect that if MSG were able to use the exact Clustal-W function, the comparative scores would be much closer.

This brings us to the matter of explaining the results. First, it is impossible for MSG to do worse than Clustal-With: the preferred Clustal-W alignment width is always tested, and one tribe is always jumpstarted with the Clustal-W result. For sequences for which Clustal-W finds an optimal alignment, MSG can do no better. The fact that MSG actually did do better in all cases indicates that Clustal-W did not find optimal alignments. MSG has a strong advantage in trying out a range of alignment widths. Often MSG's best score came at a width that was several columns wider than Clustal-W's. It seems the extra width allows for the possibility of very good column scores, which are more than enough to outweigh the incremental gap penalties that are the price of incremental width.

On March 13, 2009, I had the opportunity to speak with Prof. Dietland Gerloff, of the Biomolecular Engineering faculty at U.C. Santa Cruz. In the past she has specialized in multiple sequence alignment. I told her about MSG's results, and she said that it is certainly possible to improve on Clustal-W results [34]. Clearly MSG is able to do so. Why?

MSG's virtue lies in the simplicity of its Genetic Algorithm. There are only two operators, crossover and mutation, and these are completely generic. The only place where the GA uses knowledge about the problem domain is the unavoidably domain-specific fitness function. The best use of GAs is when the domain is too complicated for exact, approximate, or randomized algorithms. In such cases, introducing domain-based processing could interfere with the GA's ability to fully explore solution space. MSG imposes no such constraints on the GA, and exploration is quite successful.

Finally, concerning the applicability of MSG to larger data sets, it certainly seems plausible that, given powerful enough hardware, MSG could succeed regardless of input size. Obviously, no matter how big the data is, MSG can still do no worse than Clustal-W. The question is, could MSG's improvement be significant enough that it is worth the effort? The data sets that I used (except for the random sequences) came from real life; although they were truncated, they are still (for their size) more or less typical of real-world inputs. There is nothing inherent in the GA that imposes any size restriction, beyond the possibility that more generations might be required. If we accept the plausibility that MSH could succeed on larger data sets, then MSG could be a valuable addition to our field’s arsenal of bioinformatics tools.

Evaluating the Process

I named MSG before I began the hard work. I was willing to tolerate the headaches and high blood pressure that the name implies. But in the event, the development process was quite smooth. I attribute this mainly to my decision to implement a GUI for monitoring the GA’s progress. It was the visual feedback I got from the Tribes Panel that told me that tribes were all converging much too quickly. Manual tampering with crossover and mutation rates proved futile, since no rates are ideal over all generations of a single tribe, let alone all tribes or all input sets. It was then easy to conclude that my operators needed to be adaptive. As soon as I implemented adaptability, MSG began producing its excellent results.

Since I am likely to implement more GAs in the future, it is important to formalize what I’ve learned from MSG. I have four recommendations to anyone (including “future me”) who contemplates applying GA technology to new problems. First, avoid domain-specific knowledge when designing the chromosome and the operators. Second, use the multi-tribe approach unless memory usage becomes an issue. Third, as soon as the GA is able to loop through generations, implement GUI instrumentation. Fourth, use adaptation for operator rates.

Future Directions

Given unlimited resources, I would want to pursue two lines of inquiry:

Does MSG continue to provide good results with more/wider sequences? The abrupt speed dropoff as data set size increases indicates that some critical resource is completely consumed. I suspect the JVM runs out of RAM. Further investigation would require much more memory (my system is a 1.73 GHz Intel CPU with 1 Gbyte of RAM). MSG would need to implement some kind of intelligent search of alignment width space, since widths would be too large for the current +/- 10 exhaustive search scheme. The most promising approach at present is the Akaike Information Criterion strategy described in Chapter 4. The additional width might also stress the randomized algorithm currently used to create Clustal-inspired chromosomes for widths that are narrower than the Clustal-W solution. Chapter 4 describes Prof. Taylor's knapsack suggestion.

Does MSG continue to provide good results if its fitness function is replaced with an exact duplicate of Clustal-W's scoring function? Recall that MSG applies its own fitness function to the Clustal-W alignment, and uses the result as the value to beat. If MSG used the Clustal-W function instead, then presumably the Clustal-W solution would have a higher score. MSG would still do no worse than this score, but the score difference could be significantly closer. As discussed in Chapter 6, it is likely that MSG would still produce good results.

If both of these lines of inquiry lead to good results, MSG would be a good postprocessor for Clustal-W, in situations where there is value in having the highest-score MSA possible. It would be useful to relieve users of the burden of submitting their sequences to the Clustal-W website, and then pasting the results into MSG. Ideally the two programs could be integrated into a single tool. The fact that they are written in different languages is no obstacle; the two pieces could easily be connected via a TCP/IP socket with a simple custom protocol.

Chapter 7: References

[1] Akaike,H. “A new look at the statistical model identification” *IEEE Transactions on Automatic Control Vol 19, Issue 6, Dec 1974 pp 716-723.*

[2] Altenberg, L. (Duke University, Institute of Statistics and Decision Sciences) *The Schema Theorem and Price’s Theorem.* http://dynamics.org/Altenberg/FILES/LeeSTPT.pdf

[3] Altschul, S., Wootton, J., Gertz, E., Agarwala, R., Morgulis, A., Schäffer, A., & Yu, Y-K. “Protein Database Searches Using Compositionally Adjusted Substitution Matrices”. *The FEBS Journal* 2005 Oct; 272(20):5101-9. (PMID 16218944)

[4] Bar-Cohen, Y. “Biomimetics: Biologically Inspired Technologies” *CRC Press, London, 2006.*

[5] Ben Amor, H. and Rettinger, Achim. “Intelligent Exploration for Genetic Algorithms”. *GECCO ’05 ACM 1-59593-010-8/05/0006*.

[6] Cormen, T., Leiserson, C., Rivert, R., & Stein, C. *Introduction to Algorithms*, 2nd edition. McGraw-Hill, Boston, MA. 2001.

[7] De Jong, K. “Are Genetic Algorithms Function Optimizers?” *Parallel Problem Solving from Nature, 2, Männer, R., and Manderick, B. (editors).*

[8] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. *Biological Sequence Analysis*. Cambridge University Press, Cambridge, England, 2002.

[9] Feng, D.F., Johnson, M.S. & Doolittle, R.F. (1985) “Aligning amino acid sequences: comparison of commonly used methods” *J. Mol. Evol.,* 21**,** 112-125.

[10] Fogel, D. *An introduction to simulated evolutionary optimization*. In *IEEE Transaction on Neural Networks,* 5(1):3-14, 1994

[11] Fonseca, C. & Fleming, P. “Genetic Algorithms for Multiobjective Optimization:

Formulation, Discussion and Generalization” *Proceedings of the Fifth International Conference on Genetic Algorithms.*

[12] Heller, P. & Roberts, S. *Complete Java 2 Certification*, 5th edition. Sybex, Alameda, CA. 2005.

[13] Heller, P. *Ground-Up Java*. Sybex, Alameda, CA. 2004.

[14] Henikoff, S. & Henikoff, J. “Amino acid substitution matrices from protein blocks” *Proc. Natl. Acad. Sci. USA* Vol. 89, pp. 10915-10919, Nov 1992.

[15] Holland, John H. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI. 1975.

[16] Houssain, Z. “Modified Akaike Information Criterion (MAIC) For Statistical Model Selection” *Pak. J. Statist. 2002 Vol. 18(3) pp 383-393*.

[17] Horng, J.T., Wu, L. C., Lin, C.M., & Yang, B.H. *A genetic algorithm for multiple sequence alignment*. In *Soft Computing, 9, 407-420 (2005).*

[18] Julstrom, Bryant A. *Very Greedy Crossover in a Genetic Algorithm for the Traveling Salesman Problem*. In *Proceedings of the 1995 ACM Symposium on Applied Computing*

[19] Khuri, S. *Design and Analysis of Algorithms: Course Notes for Computer Science 255, Spring Semester 2008*. San Jose State University, 2008.

[20] Khuri, S., Back, T., & Heitkotter, J. “The Zero-One Multiple Knapsack Problem and Genetic Algorithms”. *Proceedings of the 1994 ACM Symposium on Applied Computing*.

[21] Lipman, D.J., Altschul, S.F. & Kececioglu, J.D. “A tool for multiple sequence alignment” *Proc Natl Acad Sci USA. 1989; 86:4412-4415*

[22] Lucasius, C.B. & Kateman, G. *Towards Solving Subset Selection Problems with the Aid of the Genetic Algorithm*. In *Parallel Problem Solving from Nature, 2,* Männer, R., and Manderick, B. (editors).

[23] Man, K.F., Tang, K.S., and Kwong, S. *Genetic Algorithms*. Springer-Verlag London Berlin Heidelberg. 1999.

[24] Moore, M. “Teaching Students To Use Genetic Algorithms To Solve Optimization Problems”. *JCSC 16, 3 (March 2001).*

[25] Mühlenbein, H. “How Genetic Algorithms Really Work: Mutation and Hillclimbing” *Parallel Problem Solving from Nature, 2*, Männer, R., and Manderick, B. (editors).

[26] Notredame, C and Higgins, D.G. Sequence Alignment by Genetic Algorithm (SAGA). Nucleic Acid Research, 1996, vol 24, 8, 1515-1524; 1996.

[27] Polani, D. & Uthmann, T. *Adaptation of Kohonen Feature Map Topologies by Genetic Algorithms*. In *Parallel Problem Solving from Nature, 2,* Männer, R., and Manderick, B. (editors).

[28] Smith, J.E. & Fogarty, T.C. *Operator and Parameter Adaptation in Genetic Algorithms*. In *Soft Computing – A Fusion of Foundations, Methodologies, and Applications*. Springer, Berlin/Heidelberg, 1997.

[29] Thompson, J.D., Plewniak, F., & Poch, O. “BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs” *Bioinformatics, Col 15, 87-88*. 1999.

[30] Turner, A., Corne, D., Ritchie, G., & Ross, D. “Obtaining multiple distinct solutions with genetic algorithm niching methods” *Lecture Notes In Computer Science*, *Springer, Berlin/Heidelberg, 1996.*

[31] Zhang, C. and Wong, A. A. A Genetic Algorithm for Multiple Sequence Alignment. Comput. Appl. Bioscience, 13, 565-581; 1997.

[32] Krogh, A., Brown, M., Mian, I., Sjölander, K., & Haussler, D. “Hidden Markov Models in Computational Biology” *J. Mol. Biol. (1994) 235, 1501-1531.*

[33] Mitzenmacher, M. & Upfal, E. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, Cambridge, England. 2005.

[34] Gerloff, D. (U.C. Santa Cruz, Biomolecular Engineering) *Pers. Comm*. March 13, 2009.

[35] Zvelebil, M. & Baum, J. *Understanding Bioinformatics*. Garland Science Textbooks, London. 2007.

Appendix A: Data

This appendix lists all the sequences of all the datasets I used. Format is *name*:*sequence*. Dataset names correspond to labels on menu items in MSG.

BAliBASE data sets:

BB11001

1aab_: GKGDPKKPRGKMSSYAFFVQTSREEHKKKHPDASVNFSEFSKKCSER
1j46_A: MQDRVKRPMNAFIVWSRDQRRKMALENPRMRNSEISKQLGYQ
1k99_A: MKKLKKHPDFPKKPLTPYFRFFMEKRAKYAKLHPEMSNLDLTKILSKK
2lef_A: MHIKKPLNAFMLYMKEMRANVVAESTLKESAAINQILGRR

BB30020

1awd_: TLKTPSGEETIECPEDTYILDAAEEAGLDLPYSCRAGACSS
FER1_DUNSA: TLKTPSGEQKVEVSPDSYILDAAEEAGVDLPYSCRAGSCSS
FER_EUGVI: KLINPDGEVTIECGEDQYILDAAEDAGIDLPYSCRAGACSS
FER_LEUGL: KLLTPDGPKEFECPDDVYILDQAEELGIELPYSCRAGSCSS
FER_MARPO: TLNTPTGQSVIDVEDDEYILDAAEEAGLSLPYSCRAGACSS
FER_PERBI: TLDTPDGKKSFECPGDSYILDKAEEEGLELPYSCRAGSCSS
FER_RHOPL: TLSTPGGVEEIEGDETTYVLDSAEDQGIDLPYSCRAGACST
FER_SOLLY: KLITPEGPVEFNCPDDVYILDSAEENGHDLPYSCRAGACSS

BB50001

SYP_BACSU: MRQSLTLIPTLREVPADAEAKSHQLLLRAGFIRQNT
SYP_ECOLI: MRTSQYLLSTLKETPADAEVISHQLMLRAGMIRKLA
SYP_HAEIN: MRTSQYLFSTLKETPNDAQVVSHQLMLRAGMIRPMA
SYP_MYCTU: MITRMSELFLRTLRDDPADAEVASHKLLIRAGYIRPVA
SYP_RICPR: MLLSKYFLPILKEEPSEAKVISHKLMLRSGMIMKQA
SYP_STRCO: MANAPVQRMSKLMAKTLRDDPADAEVLSHKLLVRAGYVRRTA
SYP_TREPA: MSAFFAPTLRSAPADATIASHQLLMRAGYVRKIA
SYP_ZYMMO: MRLSRYFLPVMKETPADAQIISHKLMLRAGMIRQTA

HIV/SIV data sets:

gag_04

HIV2_D205_gag: APPADPAVEMLKSYMQMGRQQRESRERPYKEVTEDLLHLNSLFGEDQ
HIV2_UCI_gag: APPADPAVEMLKSYMKMGRQQRESRERPYKEVTEDLLHLNSLFGEDQ

SIV_AGM_gag: YDPAKKLLQQYAEKGQRLREEREQTRKQKEKEVEDVSLSSLFGGDQ
SIV_Mm251_gag: PAVDLLKNYMQLGKQQRESREKPYKEVTEDLLHLNSLFGGDQ

gag_08

HIV2_D205_gag: APPADPAVEMLKSYMQMGRQQRESRERPYKEVTEDLLHLNSLFGE
HIV2_ROD_gag: KRQREQRERPYKEVTEDLLHLEQGETPYREPPTEDLLHLNSLFGK
HIV2_ST_gag: GKRQREQRERPYKEVTEDFLQLEKQETPCRETTEDLLHLNSLFGK
HIV2_UCI_gag: APPADPAVEMLKSYMKMGRQQRESRERPYKEVTEDLLHLNSLFGE
SIV_AGM_gag: YDPAKKLLQQYAEKGQRLREEREQTRKQKEKEVEDVSLSSLFGG
SIV_CPZ_gag: PTAPPIESYGYQEEEKSQEKKEGESSLYPPTSLKSLFGS
SIV_Mm251_gag: PAVDLLKNYMQLGKQQRESREKPYKEVTEDLLHLNSLFGG
SIV_SMM_gag: APPEDPAVDLLKNYMKMGRKQRENRERPYKEVTEDLLHLNSLFGE

pol_04

HIV1_BRU_pol: FFREDLAFLQGKAREFSSEQTRANSPTISSEQTRA
HIV1_NDK_pol: FFREDLAFPQGKAGEFSSEQTRANSPTSRELRVWG
SIV_AGM_pol: VRQNWPYGKRLQEWTGKFFRVWPLGRSETKKFCAI
SIV_SMM_pol: KTGGFFRAWPMGKEAPQFPHGPDASGADTNCSPRG

pol_08

HIV1_BRU_pol: FFREDLAFLQGKAREFSSEQTRANSPTISSEQTRA
HIV1_ELI_pol: FFRENLAFPQGKAGELSPKQTRANSPTSRELRVWG
HIV1_MAL_pol: FFRENLAFPQGKAREFPSEQTRANSPTSRELRVWG
HIV1_NDK_pol: FFREDLAFPQGKAGEFSSEQTRANSPTSRELRVWG
SIV_AGM_pol: VRQNWPYGKRLQEWTGKFFRVWPLGRSETKKFCAI
SIV_CPZ_pol: STKKKRLLAVWARGTPNERLHRKTGEFFRERLAFP
SIV_Mm251_pol: VLELWEGGTLCKAMQSPKKTEMLEMWKNGPCYGQM
SIV_SMM_pol: KTGGFFRAWPMGKEAPQFPHGPDASGADTNCSPRG

PDGH data sets:

first_7

gi|148683496|gb|EDL15443.1|: KRG
gi|149412101|ref|XP_001510685.:
MKSGGFSSILQDPGQRLSPEKSGRNVSPADTLQQQRNLPDGNNQRGASPQ
gi|149633265|ref|XP_001509581.: MDTVGYGSSVKEN
gi|149698125|ref|XP_001500695.: MLLFGLLLLTSALAGQRHGTQAE
gi|149716446|ref|XP_001500485.: MHRLILIYTLVCANFCSYRDTSAIPQSASIK
gi|149716448|ref|XP_001500479.: MHRLILIYTLVCANFCSYRDTSAIPQSASIK
gi|159159983|gb|ABW95041.1|: MLLFGFLLLTFALVSQRQGAEAE

last_6

gi|10242385|ref|NP_064355.1|: MLLLGLLLLTSALAGQRTGTRAESNLSSKLQLSSDKEQNGVQDP
gi|13376808|ref|NP_079484.1|:
MHRLIFVYTLICANFCSCRDTSATPQSASIKALRNANLRRDESNHLTDLY
gi|13786136|ref|NP_112607.1|: MLLLGLLLLTSALAGQRTGTRAESNLSSKLQLSSDKEQNGVQDP
gi|8886884|gb|AAF80597.1|AF244:
MSLFGLLLVTSALAGQRRGTQAESNLSSKFQFSSNKEQNGVQDP
gi|9652344|gb|AAF91483.1|AF286:
MLLLGLLLLTSALAGQRTGTRAESNLSSKLQLSSDKEQNGVQDP
gi|9994187|ref|NP_057289.1|: MSLFGLLLLTSALAGQRQGTQAESNLSSKFQFSSNKEQNGVQDP

middle_4

gi|139948855|ref|NP_001077175.: MHRLVLVYTLVCANFCSYRDTSATPQSASIKALRNAN
gi|148683495|gb|EDL15442.1|: KRGER
gi|148683496|gb|EDL15443.1|: KRGER
gi|149412101|ref|XP_001510685.:
MKSGGFSSILQDPGQRLSPEKSGRNVSPADTLQQQRNLPDGNNQRGASPQ

Random data sets:

IntenseSkew

Random_0: VYGELNVLSCLMNAPMTMQVFCVTNMWQMVMMIGSRHCMMLPYLCDVFD
Random_1: FIGDNFATGTQAPGHVIFLVLCNHRMVQHQNDFGCDWCEDWA
Random_2: HRAVEMDNDWWRFCQLQRLYKSDPGFYQYGIEENFT
Random_3: WMYYEVKCPTMMWRDCNHFAYENGAWWCY
Random_4: RYEMTFLKYFYGTLCGWELVFH
Random_5: PPIINCQYAPFQHKD

LightSkew

Random_0: QVYRNCSDYHQQFPYFNTEHFCDREHAPNQLPLNTMGCFHMYCQIIW
Random_1: IPPCIKQCKITEEAVRCKGVRKFCHMQVVQHHSSHGNFKERVQW
Random_2: TSQDQFMAQINDACYRDESGPYMLECYPCNQIRLHNRVEMCYHKNW
Random_3: KTINHYSAAVNSDGNTSYYQTHWPFHTTGAIFYQEYNHGQNW
Random_4: GYDISAGMLRQNKTVEWFDPCGNEARYLNAQGTAQDSCMVME
Random_5: AWCPSVAMCVAPFFELGAKGEYCDHTWPLQVEHDWAHMTC

ModerateSkew

Random_0: TLRSNIHSMWCDGDIIGGDPLHLTCTDLHPNKVRIENVFHDCYAWTAN
Random_1: LQWKRNCFYRNANMHRAIHPQCRYVHRGVSMTAHMSMWNPIECWA
Random_2: SITVWINIMVSFKAVFESCCFDKVPFQSEPNPQMKFHSPC
Random_3: HGWVHQDKGYTMIKHQFMSLLKTTNHQEMTMWSIP

Random_4: QEVDHCGMHQMRNQFEYEHIYHEWCMTLHP
Random_5: AENIHDSSKQPKANMANAPQTVFVN

Uniform

Random_0: AHTEERFGHDYNERPCVWFQMLGNNHPAQKSWGDDCDAPEVRENSA
Random_1: YTYCIIYPHYSWGIQAYYRIMYAGQMYRYCNCMWTIIKMLEYHEL
Random_2: NTGVKYSFFLYQHIKSPMSDLYGNIHQTRGARWGSETYCAFAMAAP
Random_3: HDYKASHQLFQCWPSQKWYDMCCIKVWISKRTLQVQQYQLVVNEDM
Random_4: PWDPLQMLMEGPFIASWWVPELEYTEYPPGRNMEILPGPIDQEV
Random_5: QMTPLRMWPLLLDHAHSVTTEIQESEHCCVCMNKKSYALVAT

Appendix B: Alignments

This appendix presents the alignments that MSG produced for each data set. Accompanying each alignment is the Clustal-W alignment for the same sequences.

BAliBASE data sets:

BB11001

MSG Alignment (score = 336):

```
GKGD-PKKPRGKMSSYA-F---FVQTSREEHKK-KHPDASVNFSEFSKKC-SER
MQDRV-KRP---MNA---FI-VWSRDQRRKMA-LENPRM-RN-SEISKQ-LGYQ
MK-KLKKHPDFPKKPLTPYFRFFM-EKRAKYAKL-HPEMS-NL-DLTK-ILSKK
M--HI-KKP---LNA---FML-YMKEMRANVVA-ES-TLKESAA-I-NQILGRR
```

Clustal-W Alignment (score = 191):

```
GKGDPKK---PRGKMSSYAFFVQTSREEHKKKHPDASVNFSEFSKKCSER
---MQDR---VKRPMNAFIVWSRDQRRKMALENPRMRN--SEISKQLGYQ
MKKLKKHPDFPKKPLTPYFRFFMEKRAKYAKLHPEMSN--LDLTKILSKK
-----MH---IKKPLNAFMLYMKEMRANVVAESTLKES--AAINQILGRR
```

BB30020

MSG Alignment (score = 4261):

```
TLKTPSGE-ETIECP-EDT-YILDAAEE-AGLDLPYSCRAGACSS
TLKTPSGEQK-VEV-SPD-SYILDAAEE-AGVDLPYSCRAGSCSS
KLINPDGEV-TIEC-GED-QYILDAAED-AGIDLPYSCRAGACSS
KLLTPDGP-KEFECP-DDV-YILDQAEE-LGIELPYSCRAGSCSS
TLNTPTGQ-SVIDVE-DD-EYILDAAEE-AGLSLPYSCRAGACSS
TLDTPDGK-KSFECPG-D-SYILDKAEEE-GLELPYSCRAGSCSS
TLSTPGG-VEEIE--GDETTYVLDSAEDQ-GIDLPYSCRAGACST
KLITPEGPVE-FNCP-DDV-YILDSAEEN-GHDLPYSCRAGACSS
```

Clustal-W Alignment (score = 4073):

```
TLKTPSGEETIECPEDTYILDAAEEAGLDLPYSCRAGACSS
TLKTPSGEQKVEVSPDSYILDAAEEAGVDLPYSCRAGSCSS
```

```
KLINPDGEVTIECGEDQYILDAAEDAGIDLPYSCRAGACSS
KLLTPDGPKEFECPDDVYILDQAEELGIELPYSCRAGSCSS
TLNTPTGQSVIDVEDDEYILDAAEEAGLSLPYSCRAGACSS
TLDTPDGKKSFECPGDSYILDKAEEEGLELPYSCRAGSCSS
TLSTPGGVEEIEGDETTYVLDSAEDQGIDLPYSCRAGACST
KLITPEGPVEFNCPDDVYILDSAEENGHDLPYSCRAGACSS
```

BB50001

<u>MSG Alignment (score = 2906):</u>

```
MR-QSLT----L-IPTLR-EVPA-DAEAKSHQLLLRAGFIRQNT
MR--T-SQ--YL-LSTLK-ETPA-DAEVISHQLMLRAGMIRKLA
MR--T-SQ--YLF-STLK-ETP-NDAQVVSHQLMLRAGMIRPMA
M----ITRMSELFLRTLRDD-PA-DAEVASHKLLIRAGYIRPVA
M-L--LSK--Y-FLPILKEE-PS-EAKVISHKLMLRSGMIMKQA
MANAPVQRMSKLMAKTLRDD-PA-DAEVLSHKLLVRAGYVRRTA
M-----S--AF-FAPTLRS-APA-DATIASHQLLMRAGYVRKIA
MR---LSR--Y-FLPVMK-ETPA-DAQIISHKLMLRAGMIRQTA
```

<u>Clustal-W Alignment (score = 2702):</u>

```
------MRQSLTLIPTLREVPADAEAKSHQLLLRAGFIRQNT
------MRTSQYLLSTLKETPADAEVISHQLMLRAGMIRKLA
------MRTSQYLFSTLKETPNDAQVVSHQLMLRAGMIRPMA
----MITRMSELFLRTLRDDPADAEVASHKLLIRAGYIRPVA
------MLLSKYFLPILKEEPSEAKVISHKLMLRSGMIMKQA
MANAPVQRMSKLMAKTLRDDPADAEVLSHKLLVRAGYVRRTA
--------MSAFFAPTLRSAPADATIASHQLLMRAGYVRKIA
------MRLSRYFLPVMKETPADAQIISHKLMLRAGMIRQTA
```

HIV/SIV data sets:

gag_04

<u>MSG Alignment (score = 956):</u>

```
APPADPAVEMLKS-YM-QMGRQQ-RESRE----RPYKEVTEDLLHLNSLFGEDQ
APPADPAVEMLKS-YM-KMGRQQ-RESRE----RPYKEVTEDLLHLNSLFGEDQ
YDPAK---KLLQQ-YAEK-G-QRLREEREQTRKQKEKEV-EDV-SLSSLFGGDQ
--PA---VDLLKN-YM-QLGKQQ-RESRE---K-PYKEVTEDLLHLNSLFGGDQ
```

Clustal-W Alignment (score = 876):

```
APPADPAVEMLKSYMQMGRQQRESRE--RPYKEVTEDLLHLNSLFGEDQ
APPADPAVEMLKSYMKMGRQQRESRE--RPYKEVTEDLLHLNSLFGEDQ
---YDPAKKLLQQYAEKGQRLREEREQTRKQKEKEVEDVSLSSLFGGDQ
-----PAVDLLKNYMQLGKQQRESRE--KPYKEVTEDLLHLNSLFGGDQ
```

gag_08

MSG Alignment (score = 2986):

```
APPADPAV--EMLKSYMQMGR----Q-QRESRERPYKEV-TEDLLHLNSLFGE
KRQRE---QRE--RPYKEV-TEDLLHLE-QGET-PYREPPTEDLLHLNSLFGK
GK-RQRE-QRE--RPYKEV-TEDFLQLEKQ-ET-PCRET-TEDLLHLNSLFGK
APPADPAV--EMLKSYMKMGR----Q-QRESRERPYKEV-TEDLLHLNSLFGE
Y---DPA--KKLLQQYAE--KGQRLREERE-QTRKQKEKEVEDV-SLSSLFGG
-PTA-PPI--ESY-GYQE---EEKSQEKKEGESSLY--PPTS----LKSLFGS
-P----AV--DLLKNYMQLGK----Q-QRESREKPYKEV-TEDLLHLNSLFGG
APPEDPAV--DLLKNYMKMGR--K-Q--RENRERPYKEV-TEDLLHLNSLFGE
```

Clustal-W Alignment (score = 2689):

```
APPADPAVEMLKSYMQMGRQ--QRESRE--RPYKEV-TEDLLHLNSLFGE
---KRQREQRERPYKEVTEDLLHLEQGE--TPYREPPTEDLLHLNSLFGK
--GKRQREQRERPYKEVTEDFLQLEKQE--TPCRET-TEDLLHLNSLFGK
APPADPAVEMLKSYMKMGRQ--QRESRE--RPYKEV-TEDLLHLNSLFGE
---YDPAKKLLQQYAEKGQR--LREEREQTRKQKEKEVEDVS-LSSLFGG
-----PTAPPIESYGYQEEE----KSQE--KKEGESSLYPPTSLKSLFGS
-----PAVDLLKNYMQLGKQ--QRESRE--KPYKEV-TEDLLHLNSLFGG
APPEDPAVDLLKNYMKMGRK--QRENRE--RPYKEV-TEDLLHLNSLFGE
```

pol_04

MSG Alignment (score = 324):

```
FFREDL-AFLQGK-AREFSSEQT-RANSPTIS-SE-Q----T-RA
FFREDL-AFPQGK-AGEFSSEQT-RANSPT-SR-ELRVW-----G
V-RQ--N-WPYGKRLQEWTG-KFFRV-WP-LGRSETKKFC-AI--
KTGGFFRAWPMGKEAPQF--P-----HGPDASGADTN--C-SPRG
```

Clustal-W Alignment (score = 83):

```
---FFREDLAFLQGKAREFSSEQTRANSPTISSEQTRA--
---FFREDLAFPQGKAGEFSSEQTRANSPTSRELRVWG--
```

```
----VRQNWPYGK-RLQEWTGKFFRVWPLGRSETKKFCAI
KTGGFFRAWPMGK-EAPQFPHGPDASGADTNCSPRG----
```

pol_08

MSG Alignment (score = 1206):

```
FFRE-DLAFLQGKA--REF-SS--EQT--RANSPTISSE-QTRA
FFRE-NLAFPQGKA-G-EL--SP-KQT--RANSPT-SRELRVWG
FFRE-NLAFPQGKA--REFP-S--EQT--RANSPT-SRELRVWG
FFRE-DLAFPQGKA-G-EF-SS--EQT--RANSPT-SRELRVWG
V-RQ-N--WPYGKRL-QEWTGKFF-RVWPLGRSET--KKFCAI-
STKKKRL---L--AVWAR--GTPNERLH-RKTGE-FFRERLAFP
VL-ELWEGGTLCKAM-QS-PKK-TEMLEMWKNGPCY-GQM----
KTGGFFRAWPMGKEAP-QFPHGP-D-A-S-G-ADTNCSP-R--G
```

Clustal-W Alignment (score = 475):

```
FFREDLAFLQGKAREFSSEQTRANSPTISSEQTRA--
FFRENLAFPQGKAGELSPKQTRANSPTSRELRVWG--
FFRENLAFPQGKAREFPSEQTRANSPTSRELRVWG--
FFREDLAFPQGKAGEFSSEQTRANSPTSRELRVWG--
-VRQNWPYGK-RLQEWTGKFFRVWPLGRSETKKFCAI
STKKKRLLAVWARGTPNERLHRKTGEFFRERLAFP--
-VLELWEGGTLCKAMQSPKKTEMLEMWKNGP-CYGQM
--KTGGFFRAWPMGKEAPQFPHGPDASGADTNCSPRG
```

PDGH data sets:

first_7

MSG Alignment (score = 228):

```
K-------------------------------------------------RG----
MKSG-GFSSILQDPG-QRLSPEKS-GRNVSPA--DT-LQQQRNLPDGNNQRGASPQ
M--DT-------------------V-G-Y--G--SSVKE-----------N-----
M---LLFGLL-L---LTS---AL-AG----------------------QRHGTQAE
MHR-LIL-----IYTL-V---------CAN--FC-SYR--DTS-AI-P--QSASIK
MH-RLI--L-IY--TLVC-A-------NFCS-YRDT-SA-------IP--QSASIK
M---LLFG-FL-------L--LT----------FALV-------S--Q-RQGAEAE
```

Clustal-W Alignment (score = 139):

```
-------------------------------------------KRG----
MKSGGFSSILQDPGQRLSPEKSGRNVSPADTLQQQRNLPDGNNQRGASPQ
MDTVGYGSSVKEN-------------------------------------
--MLLFGLLLLTSA--LAGQRHGTQAE-----------------------
MHRLILIYTLVCAN--FCSYRDTSAIPQSASIK-----------------
MHRLILIYTLVCAN--FCSYRDTSAIPQSASIK-----------------
--MLLFGFLLLTFA--LVSQRQGAEAE-----------------------
```

last_6

<u>MSG Alignment (score = 1980):</u>

```
M-LLL-GLLLLTSALA-GQRT-GT-RAESN--LSSKLQLSSDKEQNGVQDP
MHRLIFVYTLICANFCSCRDTSATPQSASIKALRNA-NLRRDESNHLTDLY
M-LLL-GLLLLTSALA-GQRT-GT-RAESN--LSSKLQLSSDKEQNGVQDP
MS-L-FGLLLVTSALA-GQR-RGT-QAESN--LSSKFQFSSNKEQNGVQDP
M-LLL-GLLLLTSALA-GQRT-GT-RAESN--LSSKLQLSSDKEQNGVQDP
MS-L-FGLLLLTSALA-GQR-QGT-QAESN--LSSKFQFSSNKEQNGVQDP
```

<u>Clustal-W Alignment (score = 1915):</u>

```
--MLLLGLLLLTSALAGQRT--GTRAESNLSSKLQLSSDKEQ-NGVQDP-
MHRLIFVYTLICANFCSCRDTSATPQSASIKALRNANLRRDESNHLTDLY
--MLLLGLLLLTSALAGQRT--GTRAESNLSSKLQLSSDKEQ-NGVQDP-
--MSLFGLLLVTSALAGQRR--GTQAESNLSSKFQFSSNKEQ-NGVQDP-
--MLLLGLLLLTSALAGQRT--GTRAESNLSSKLQLSSDKEQ-NGVQDP-
--MSLFGLLLLTSALAGQRQ--GTQAESNLSSKFQFSSNKEQ-NGVQDP-
```

middle_4

<u>MSG Alignment (score = 79):</u>

```
M-HR--L--VLVY--TLVCANFCSYRD-TSA-T-PQSASI-K-A-LRNA--N
K---------------------------------------------RG--ER
K---------------------------------------------RG--ER
MK-SGGFSSILQDPGQRLSPE-KSGRNVSPADTLQQQRNLPDGNNQRGASPQ
```

<u>Clustal-W Alignment (score = 10):</u>

```
--MHRLVLVYTLVCANFCSYRDTSATPQSASIKALR----N--AN-----
----------------------------------KR----G--ER-----
----------------------------------KR----G--ER-----
```

```
MKSGGFSSILQDPGQRLSPEKSGRNVSPADTLQQQRNLPDGNNQRGASPQ
```

IntenseSkew

MSG Alignment (score = -10):

```
VYGELNVLSCLMNAPM--TMQ-VFCVTNMWQ-M-VMMIG-SRH---CMML-PYLCD-VFD
FIG-DNFATG-TQAP--GH-V-IF---LVLCN--H-RM-VQHQNDFGC---DW-CED-WA
H--R--A-VEM-D-N-----DWWRFC--QL-Q-R---LYKSDPGF--YQ-Y-G-IEENFT
WM---YY-E-V--K--CPTMMW-RDCN-H-FAY-------ENGA--W-----W-C---Y-
RY------E-M---TF---L-K-YFY--------G----------T-LC--GW--ELVFH
P---P------IIN-C-----Q-------YAPFQH-----KD------------------
```

Clustal-W Alignment (score = -51):

```
VYGELNVLSCLMNAPMTMQVFCVTN-MWQMVMMIGSRHCMMLPYLCDVFD
FIGDNFATGTQAPGHVIFLVLCNHR-MVQHQNDFG---C----DWCEDWA
--------HRAVEMDNDWWRFCQLQRLYKSDPGFYQY------GIEENFT
-W-------------MYYEVKCPTM-MWRDCNHFAYENG---AWWC--Y-
----------------RYEMTFLKY-FYGTL-------C----GWELVFH
----------------PPIINCQYA-PFQHKD------------------
```

LightSkew

MSG Alignment (score = 275):

```
QVYRNC-S--DYHQQFPYFNT-EHFCDREHA-PNQLPLNTMGCFH-MYCQII-W
IP--PCIKQCKITEEAVRCKGVRKFCHMQ-V--VQHH-SSHGNFKE---RVQ-W
TSQDQ-F-MAQINDACYRDES-GPYM-LE-CYPCN-QIRLHNRV-EMCYH-KNW
KTINH-YSAA-VNSD--G-NT-S-Y-YQTH-WP--FH-TTGAIFYQEYNHGQNW
G-YD--IS-AGM---LRQNKTV-EW-F-DPCGNEARYLNAQGTAQDSCM-VME-
A-WCPSVAMC-VAP-FFELGAKGEYC--DHTWP----LQVE---HD-WAH-MTC
```

Clustal-W Alignment (score = -132):

```
QVYRNCSDYHQQFPYFNTEHFCDREHAPNQLPLNTMG--CFHMYCQ-IIW
-IPPCIKQCKITEEAVRCK--GVRKFCHMQVVQHHSS---HGNFKERVQW
---TSQDQFMAQINDACYRD-ESGPYMLECYPCNQIRLHNRVEMCYHKNW
---KTINHYSAAV---NSD--GNTSYYQTHWPFHTTGAIFYQEYNHGQNW
GYDISAGMLRQNKTVEWFD--PCGN--EARY-LNAQG--TAQDSCMVME-
AWCPSVAMCVAPF--FELG--AKGEYCDHTWPLQVEHD-WAHMTC-----
```

ModerateSkew

MSG Alignment (score = 269):

```
TLRSNIHSMWCDG-DIIGGDPLHLTCTDLHPNKVRIENVFHDC-Y-AWT-AN
LQWKR-NCFYRNA-NM--HRAIHPQCRYVHRGVSMTAHMSMWNPIECWA---
SITVWINIMV--SFKAV----FESCCFDKVPFQSE-PNP-Q---MK-FHSPC
HGW--VH-QD-KGYTMIKHQFM-S-LL-----KT-T-N-HQ--EMTMWSIP-
QE-VD-H---C-G---M-HQ-MRNQFEYEH-----IYH--EWC-MT-LH-P-
AE--NIH----DS-S--K-Q----------P-KANMAN----APQTVF-V-N
```

Clustal-W Alignment (score = -109):

```
TLRSNIHSMWCDGDIIGGDPLHLTCTDLHPN--KVRIENVFHDCYAWTAN
-LQWKRNCFYRNANMHR--AIHPQCRYVHRGVSMTAHMSMWNPIECWA--
SITVWINIMVSFKAVFES-CCFDKVPFQSEP----NPQMKFH-SPC----
--HGWVHQDKGYTMIKH--QFMSLLKTTNH-----QEMTMWS-IP-----
-------QEVDHCGMHQ---MRNQFEYEH-------IYHEWC-MTLHP--
-----------AENIHDS-SKQPKANMAN------APQTVF--VN-----
```

Uniform

MSG Alignment (score = 209):

```
AHTE-ERFGH--DY-N-ERPCVWF-QML-GNNHP--AQKSWGDDC-DAPEVRENSA
Y-TYCIIYPH---YS-WGIQ-AYYR-IMYAGQ-MYRYCNCMWT-II-K-MLEYHEL
N-TGVK-YSFFL-YQHIKSPMS---DL-YGNIHQTRGAR-WGSETYCAFAMA-AP-
H-DY-KA-SHQL-FQCWPSQ-KWY-DMCCI-K-VWISKRTLQVQQYQ--LVVNEDM
PWDPLQM---LMEGPFIAS-W-WVPELEY-TEYPP-G-RNM--EILPGP-I-DQEV
QMTPLRMWPLLLDHAH--S-VT--TEIQ-ESEH--C-CVCMNKKSY-A-LVA---T
```

Clustal-W Alignment (score = -499):

```
--AHTEERFGHDYNERPCV-WFQMLGNNHPAQKSWGDDCDAP-EVRENSA
---YTYCIIYPHYS-WGIQAYYRIMYAGQMYRYCNCMWTIIK-MLEYHEL
NTGVKYSFFLYQHIKSPMSDLYGNIHQTRGARWGSETYCAFA----MAAP
---HDYKASHQLFQCWPSQKWYDMCCIKVWISKRTLQVQQYQ-LVVNEDM
----PWDPLQMLMEGPFIASWWVPELEYTEYPPGRNMEILPG--PIDQEV
--------QMTPLRMWPLLLDHAHSVTTEIQESEHCCVCMNKKSYALVAT
```

Appendix C: Source Code

This appendix presents complete Java source code for MSG. Each source module begins on a separate page. The main application class is MSGFrame.

```
package msg;

import java.util.*;
import java.io.*;

class AlignmentScorer
{

/***
 * http://icb.med.cornell.edu/education/courses/introtobio/BLOSUM62
 *
 * Verbatim source is:
 *
 * # Entries for the BLOSUM62 matrix at a scale of ln(2)/2.0.
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  J  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1 -1 -1 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1 -2  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  4 -3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4 -3  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -1 -3 -1 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0 -2  4 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1 -3  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -4 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0 -3  0 -1 -4
I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3  3 -3 -1 -4
L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4  3 -3 -1 -4
K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0 -3  1 -1 -4
M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3  2 -1 -1 -4
F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3  0 -3 -1 -4
P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -3 -1 -1 -4
S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0 -2  0 -1 -4
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1 -1 -1 -4
W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -2 -2 -1 -4
Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -1 -2 -1 -4
V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3  2 -2 -1 -4
B -2 -1  4  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4 -3  0 -1 -4
J -1 -2 -3 -3 -1 -2 -3 -4 -3  3  3 -3  2  0 -3 -2 -1 -2 -1  2 -3  3 -3 -1 -4
Z -1  0  0  1 -3  4  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -2 -2 -2  0 -3  4 -1 -4
X -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -4
* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1

*/

    private final static int[][]        BLOSUM_62_MATRIX;
    private final static String         INDICES = "ARNDCQEGHILKMFPSTWYVBJZX";
    private final static String[]       RAW_ROWS =
    {
        "A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1 -1 -1 -4",
        "R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1 -2  0 -1 -4",
        "N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  4 -3  0 -1 -4",
        "D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4 -3  1 -1 -4",
        "C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -1 -3 -1 -4",
        "Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0 -2  4 -1 -4",
        "E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1 -3  4 -1 -4",
        "G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -4 -2 -1 -4",
        "H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0 -3  0 -1 -4",
        "I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3  3 -3 -1 -4",
        "L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4  3 -3 -1 -4",
        "K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0 -3  1 -1 -4",
        "M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3  2 -1 -1 -4",
        "F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3  0 -3 -1 -4",
        "P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -3 -1 -1 -4",
        "S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0 -2  0 -1 -4",
        "T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1 -1 -1 -4",
        "W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -2 -2 -1 -4",
        "Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -1 -2 -1 -4",
        "V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3  2 -2 -1 -4",
        "B -2 -1  4  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4 -3  0 -1 -4",
        "J -1 -2 -3 -3 -1 -2 -3 -4 -3  3  3 -3  2  0 -3 -2 -1 -2 -1  2 -3  3 -3 -1 -4",
        "Z -1  0  0  1 -3  4  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -2 -2 -2  0 -3  4 -1 -4",
        "X -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -4",
        "* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1"
    };


    static boolean isAa(char ch)
    {
        ch = Character.toUpperCase(ch);
        if (ch < 'A'  ||  ch > 'Z')
            return false;
        return "BJOUXZ".indexOf(ch) < 0;
    }


    static
    {
        // Convert raw string table to a map of maps.
        TreeMap<Character, TreeMap<Character, Integer>> mapOfMaps =
            new TreeMap<Character, TreeMap<Character, Integer>>();
        for (String row: RAW_ROWS)
        {
            String[] rawPieces = row.split(" ");     // produces some length=0 pieces
            Vector<String> pieces = new Vector<String>();
            for (String rawPiece: rawPieces)
                if (rawPiece.trim().length() > 0)
                    pieces.add(rawPiece.trim());
            assert pieces.size() == 26 : pieces.size();
            // (0) = amino acid, (1-25) = vals by INDICES, including 5 weirdos
            // at the end for B/J/Z/X/*.
            Character aa = pieces.remove(0).charAt(0);
            TreeMap<Character, Integer> map = new TreeMap<Character, Integer>();
            mapOfMaps.put(aa, map);
            for (int i=0; i<20; i++)
            {
                Character otherAa = INDICES.charAt(i);
                Integer val = Integer.parseInt(pieces.get(i));
                map.put(otherAa, val);
            }
        }

        // Convert map of maps to 26x26 array. Only entries for the 20 amino
        // acids are valid.
        BLOSUM_62_MATRIX = new int[26][26];
        for (int i=0; i<26; i++)
            for (int j=0; j<26; j++)
                BLOSUM_62_MATRIX[i][j] = Integer.MIN_VALUE;
        String notAas = "BJOUXZ";
        for (char aa: mapOfMaps.keySet())
        {
            if (!isAa(aa))
                continue;
            Map<Character, Integer> map = mapOfMaps.get(aa);
            for (char otherAa: map.keySet())
            {
                if (!isAa(otherAa))
                    continue;
                int val = map.get(otherAa);
                BLOSUM_62_MATRIX[aa-'A'][otherAa-'A'] = val;
            }
        }

        // Misc checks.
        for (char aa='A'; aa<= 'Z'; aa++)
        {
            if (!isAa(aa))
                continue;
            for (char aa1='A'; aa1<= 'Z'; aa1++)
            {
                if (!isAa(aa1))
                    continue;
                assert BLOSUM_62_MATRIX[aa-'A'][aa1-'A'] ==
                       BLOSUM_62_MATRIX[aa1-'A'][aa-'A'];
                assert BLOSUM_62_MATRIX[aa-'A'][aa1-'A'] > Integer.MIN_VALUE;
            }
        }
        assert BLOSUM_62_MATRIX['C'-'A']['Q'-'A'] == -3;    // Spot checks
        assert BLOSUM_62_MATRIX['M'-'A']['M'-'A'] == 5;
        assert BLOSUM_62_MATRIX['A'-'A']['P'-'A'] == -1;
    }


    static int scoreAlignment(Collection<String> alignment)
    {
        // Convert input to char[][].
        char[][] charAlignment = new char[alignment.size()][];
        int n = 0;
        for (String s: alignment)
            charAlignment[n++] = s.toCharArray();

        // Score.
        return scoreAlignment(charAlignment);
    }


    // More efficient, since scoreAlignment(Collection<String>) converts to
    // this format.
    static int scoreAlignment(char[][] alignment)
    {
        // Collect all-gap columns.
        int w = alignment[0].length;
        boolean[] ungapped = new boolean[w];
        for (int seq=0; seq<alignment.length; seq++)
        {
            for (int col=0; col<alignment[0].length; col++)
            {
                if (alignment[seq][col] != '-')
                {
                    ungapped[col] = true;
                    break;
                }
            }
        }

        int score = 0;
        for (int col=0; col<alignment[0].length; col++)
            score += scoreColumn(alignment, col);
        for (int row=0; row<alignment.length; row++)
            score += gapPenaltiesForRow(alignment[row], ungapped);
        return score;
    }


    private static int scoreColumn(char[][] alignment, int col)
    {
        int score = 0;

        try
        {
            for (int row1=0; row1<alignment.length-1; row1++)
            {
                char ch1 = alignment[row1][col];
                if (ch1 == '-')
                    continue;
                for (int row2=row1+1; row2<alignment.length; row2++)
                {
                    char ch2 = alignment[row2][col];
                    if (ch2 == '-')
                        continue;
                    int partialScore = BLOSUM_62_MATRIX[ch1-'A'][ch2-'A'];
                    assert partialScore >= -100  &&  partialScore <= 100  :
                           "bad chars [" + ch1 + "][" + ch2 + "]";
                    score += BLOSUM_62_MATRIX[ch1-'A'][ch2-'A'];
                }
            }
        }
        catch (Exception x)
        {
            sop("?????");
        }

        return score;
    }
```

```
    // Gap-open penalty = 11, gap-extend penalty = 1. Only look at columns
    // that aren't completely gaps (ungappedCols[n] = true).
    private static int gapPenaltiesForRow(char[] row, boolean[] ungappedCols)
    {
        int score = 0;
        char prevChar = '#';
        for (int col=0; col<row.length; col++)
        {
            if (!ungappedCols[col])
                continue;
            char currentChar = row[col];
            if (currentChar == '-')
            {
                if (prevChar == '-')
                    score -= 1;     // extend
                else
                    score -= 11;    // open
            }
            prevChar = currentChar;
        }
        return score;
    }


    static void sop(Object x)               { System.out.println(x); }
}
```

```
package msg;

import java.util.*;


/*
 * A chromosome contains 1 value for each gap to be inserted into any sequence of
 * the dataset. The value is the index (from 0) of the gap. Coincident gaps are
 * dealt with elsewhere. Gap locations (genes of the chromosome) are associated
 * with individual sequences, but not in any way that matters to the G.A. operators,
 * so the gap-to-containing-sequence mapping is supported elsewhere.
 */


public class Chromosome implements Comparable<Chromosome>
{
    private static int      nextSn;

    protected int           maxGapIndex;                // = consensus width - 1
    protected int[]         gapLocations;
    protected int           sn = nextSn++;
    private String          sval;                       // cached by toString()
            int             score = Integer.MIN_VALUE;  // for faster access


    Chromosome()        { }


    Chromosome(int nGaps, int maxGapIndex)
    {
        this.maxGapIndex = maxGapIndex;
        gapLocations = new int[nGaps];
    }


    Chromosome(Chromosome src)
    {
        this.maxGapIndex = src.maxGapIndex;
        this.gapLocations = new int[src.gapLocations.length];
        System.arraycopy(src.gapLocations, 0,
                         this.gapLocations, 0, this.gapLocations.length);
        this.sval = src.sval;  // immutable strings, so no risk
        this.score = score;
    }


    // Arg is a string extracted from database file. Format is e.g. "12_34_56".
    Chromosome(String dbString, int maxGapIndex)
    {
        this.maxGapIndex = maxGapIndex;

        // Major delimiter is '_'
        String[] pieces = dbString.split("_");
        gapLocations = new int[pieces.length];

        // Each piece is a gap location.
        for (int i=0; i<gapLocations.length; i++)
            gapLocations[i] = Integer.parseInt(pieces[i]);
```

```
}


String toStringForDatabase()
{
    String s = "";
    for (int x: gapLocations)
        s += x + "_";
    if (s.length() > 0)
        s = s.substring(0, s.length()-1);
    return s;
}


public String toString()
{
    if (sval == null)
    {
        sval = "Chromosome: /" + gapsToString() +
               " score=" + (isEvaluated() ? (" " + score) : " UNSCORED");
    }
    return sval;
}


String gapsToString()
{
    String s = "";
    for (int gloc: gapLocations)
        s += gloc + "/";
    return s;
}


// Not valid unless chromosome has been evaluated.
public int compareTo(Chromosome that)
{
    assert score != Integer.MIN_VALUE : "Compared before score was set.";
    assert that.score != Integer.MIN_VALUE : "Compared before score was set.";

    if (score != that.score)
        return score - that.score;
    else if (!(this.toString().equals(that.toString())))
        return this.toString().compareTo(that.toString());
    else
        return this.sn - that.sn;
}


public boolean equals(Object x)
{
    Chromosome that = (Chromosome)x;
    return this.toString().equals(that.toString());
}


public int hashCode()
{
        assert score != Integer.MIN_VALUE : "Hashed before score was set.";
        return score;
    }


    private final static int randomFrom0ThruN(int n)
    {
        return (int)((n+1) * Math.random());
    }


    void randomize()
    {
        for (int i=0; i<gapLocations.length; i++)
            gapLocations[i] = (int)((maxGapIndex+1) * Math.random());
    }


    // Tribe of origin is automatically maintained during the exchange, since
    // it's stored in the upper bits of the chromosome's values.
    static Chromosome[] crossover(Chromosome ma, Chromosome pa, float oddsPerSite)
    {
        assert ma.length() == pa.length() :
               "Unequal lengths: " + ma.length() + " != " + pa.length();

        // Build a template.
        boolean[] template = new boolean[ma.length()];
        boolean b = true;
        for (int i=0; i<template.length; i++)
        {
            if (Math.random() <= oddsPerSite)
                b = !b;
            template[i] = b;
        }

        // Produce offspring.
        Chromosome[] kids = new Chromosome[2];
        for (int i=0; i<2; i++)
            kids[i] = new Chromosome(ma.gapLocations.length, ma.maxGapIndex);
        for (int i=0; i<template.length; i++)
        {
            kids[0].gapLocations[i] = template[i] ? ma.gapLocations[i] :
pa.gapLocations[i];
            kids[1].gapLocations[i] = template[i] ? pa.gapLocations[i] :
ma.gapLocations[i];
        }
        return kids;
    }


    // Mutation rate = odds of changing 1 gap.
    void mutate(float oddsPerSite)
    {
        for (int i=0; i<gapLocations.length; i++)
        {
            if (Math.random() < oddsPerSite)
            {
                gapLocations[i] = (int)((1+maxGapIndex)*Math.random());
        }
    }
}


int getGapLocation(int n)
{
    assert n >= 0  &&  n <= gapLocations.length-1  :
           "Bad gap location " + n + " (max = " + (gapLocations.length-1) + ").";
    return gapLocations[n];
}


// Sets score to != Integer.MIN_VALUE.
void evaluate(UngappedSequenceDataset ungapped, int alignmentWidth)
{
    char[][] charArrs = toCharArrays(ungapped, alignmentWidth);
    score = AlignmentScorer.scoreAlignment(charArrs);
}


char[][] toCharArrays(UngappedSequenceDataset ungapped, int alignmentWidth)
{
    // Build empty alignment.
    int nSeqs = ungapped.size();
    char[][] gappedAlignment = new char[nSeqs][alignmentWidth];

    // Place gaps.
    int gapIndexInEntireChromosome = 0;
    int seqNum = 0;
    for (String ungappedSeq: ungapped.values())
    {
        int nGapsThisSeq = alignmentWidth - ungappedSeq.length();
        for (int i=0; i<nGapsThisSeq; i++)
        {
            int gapLocation = gapLocations[gapIndexInEntireChromosome];
            while (gappedAlignment[seqNum][gapLocation] == '-')
                gapLocation = (gapLocation + 1) % alignmentWidth;
            gappedAlignment[seqNum][gapLocation] = '-';
            gapIndexInEntireChromosome++;
        }
        seqNum++;
    }

    // Place chars in non-gap locations in gappedAlignment[][].
    seqNum = 0;
    for (String ungappedSeq: ungapped.values())
    {
        int indexInUngapped = 0;
        for (int col=0; col<gappedAlignment[0].length; col++)
        {
            if (gappedAlignment[seqNum][col] == '-')
                continue;
            gappedAlignment[seqNum][col] = ungappedSeq.charAt(indexInUngapped++);
        }
        seqNum++;
    }
```

```
        return gappedAlignment;
    }


    ArrayList<String> toMSAStrings(UngappedSequenceDataset ungapped, int alignmentWidth)
    {
        char[][] charArrs = toCharArrays(ungapped, alignmentWidth);
        ArrayList<String> ret = new ArrayList<String>();
        for (char[] charArr: charArrs)
            ret.add(new String(charArr));
        return ret;
    }


    static void sop(Object x)               { System.out.println(x);             }
    int length()                            { return gapLocations.length;        }
    int consensusWidth()                    { return maxGapIndex + 1;            }
    void setGapLocation(int index, int loc) { gapLocations[index] = loc;         }
    boolean isEvaluated()                   { return score != Integer.MIN_VALUE; }
}
```

```
package msg;

import java.io.*;
import java.util.*;


//
// Parses an alignment pasted from the ClustalW results panel.
//


class ClustalParser
{
    private final static int    MAX_SEQUENCES       =  8;
    private final static int    MAX_SEQUENCE_LEN    = 50;
    private       static File   file;


    // Returns a map from sequence name to sequence, with gaps inserted by
    // ClustalW. All sequences should be same length.
    static SequenceDataset parseFileToGapped(File f) throws IOException
    {
        file = f;
        FileReader fr = new FileReader(file);
        LineNumberReader lnr = new LineNumberReader(fr);
        String datasetName = f.getName();
        assert datasetName.endsWith(".clw");
        datasetName = datasetName.substring(0, datasetName.length()-4);
        SequenceDataset ret = new SequenceDataset(datasetName);
        String line;
        while ((line = lnr.readLine()) != null)
        {
            line = line.trim();
            if (line.length() == 0)
                continue;
            // Remove consensus value line (:.*)
            if (isConsensusLine(line))
                continue;
            String[] pieces = line.split("\\s+");   // @ least 1 whitespace
            String name = pieces[0];
            String aligned = ret.get(name);
            if (aligned == null)
                ret.put(name, pieces[1]);
            else
                ret.put(name, aligned+pieces[1]);
            if (ret.size() == MAX_SEQUENCES)
                break;
        }
        lnr.close();
        fr.close();
        int len = ret.values().iterator().next().length();
        for (String s: ret.values())
        {
            assert s.length() == len;
            String err = isAAaOrGaps(s);
            assert err == null : err;
        }
        Collection<String> clonedKeys = new HashSet<String>(ret.keySet());
        for (String k: clonedKeys)
        {
            if (ret.get(k).length() > MAX_SEQUENCE_LEN)
            {
                String v = ret.get(k).substring(0, MAX_SEQUENCE_LEN);
                ret.remove(k);
                ret.put(k, v);
            }
        }

        return ret;
    }


    private static boolean isConsensusLine(String s)
    {
        if (s.indexOf('|') >= 0)
            return false;

        return s.indexOf(':') >= 0  ||
               s.indexOf('.') >= 0  ||
               s.indexOf('*') >= 0;
    }


    // Returns null if string only contains amino acid codes or gaps. Otherwise
    // returns an error message.
    private final static String AAS_AND_GAP     = "ACDEFGHIKLMNPQRSTVWY-";
    static String isAAaOrGaps(String s)
    {
        for (int i=0; i<s.length(); i++)
            if (AAS_AND_GAP.indexOf(s.charAt(i)) < 0)
                return "Illegal char in file " + file.getName() + ": |" + s.charAt(i) +
"| in " + s;
        return null;
    }

    static void sop(Object x)          { System.out.println(x); }
}
```

```
package msg;

import java.io.*;
import java.util.*;


class ClustalToFasta
{
    static void convertFile(File dirf, String clwFilename) throws IOException
    {
        File clustalFile = new File(dirf, clwFilename);
        SequenceDataset clustalAlignment = ClustalParser.parseFileToGapped(clustalFile);
        UngappedSequenceDataset ungapped = clustalAlignment.removeGaps();
        String fastaFilename = clwFilename.substring(0, clwFilename.length()-4);
        fastaFilename += ".fasta";
        File fastaFile = new File(dirf, fastaFilename);
        FileWriter fw = new FileWriter(fastaFile);
        PrintWriter pw = new PrintWriter(fw, true);     // true for autoflush
        for (String k: ungapped.keySet())
        {
            pw.write(">" + k + "\r\n");
            pw.write(ungapped.get(k) + "\r\n");
        }
        pw.close();
        fw.close();
    }


    static void convertDir(File dirf) throws IOException
    {
        String[] contents = dirf.list();
        for (String kid: contents)
            if (kid.endsWith(".clw"))
                convertFile(dirf, kid);
    }


    static void sop(Object x)       { System.out.println(x); }


    public static void main(String[] args)
    {
        try
        {
            File dirf = MSGFrame.PDGH_DIRF;
            convertDir(dirf);
            System.out.println("Done");
        }
        catch (IOException x)
        {
            System.out.println("IO Stress: " + x.getMessage());
            x.printStackTrace(System.out);
        }
    }
}
```

```
package msg;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;


class ConsensusDialog extends JDialog implements ActionListener
{
    private ConsensusPanel      conPan;


    ConsensusDialog(ArrayList<String> msgAlignment, Collection<String> clustalAlignment,
                    int msgScore, int clustalScore)
    {
        setModal(false);
        conPan = new ConsensusPanel(msgAlignment, clustalAlignment, msgScore,
clustalScore);
        add(conPan, BorderLayout.CENTER);
        JPanel pan = new JPanel();
        JButton okBtn = new JButton("OK");
        pan.add(okBtn);
        okBtn.addActionListener(this);
        add(pan, BorderLayout.SOUTH);
        pack();
    }


    void setParams(ArrayList<String> msgAlignment, Collection<String> clustalAlignment,
                   int msgScore, int clustalScore)
    {
        conPan.setParams(msgAlignment, clustalAlignment, msgScore, clustalScore);
    }


    public void actionPerformed(ActionEvent e)
    {
        setVisible(false);
    }
}
```

```
package msg;

import java.awt.*;
import javax.swing.*;
import java.util.*;


class ConsensusPanel extends JPanel
{
    private final static Font       MSA_FONT            = new Font("Monospaced",
Font.BOLD, 16);
    private final static Font       SCORE_FONT          = new Font("Serif", Font.BOLD,
26);
    private final static int        TOP_BASELINE        =   25;
    private final static int        LEFT_MARGIN         =   25;
    private final static int        LINE_V_SPACING      =   20;
    private final static int        CONSEN_BOX_W        =   10;
    private final static int        CONSEN_BOX_H        =   16;
    private final static Color[]    CONSEN_COLORS       =
    {
        Color.CYAN, new Color(25, 255, 25), Color.YELLOW, Color.ORANGE, Color.PINK,
Color.RED
    };
    private final static Color[]    CONSEN_COLORS_4     =
    {
        Color.CYAN, Color.YELLOW, Color.ORANGE, Color.RED
    };
    private final static Color[]    CONSEN_COLORS_6     =
    {
        Color.CYAN, new Color(25, 255, 25), Color.YELLOW, Color.ORANGE, Color.PINK,
Color.RED
    };
    private final static Color[]    CONSEN_COLORS_8     =
    {
        Color.CYAN, new Color(25, 255, 25), new Color(25, 255, 25),
        Color.YELLOW, Color.YELLOW, Color.ORANGE, Color.PINK, Color.RED
    };
    private final static Color[]    CONSEN_COLORS_10    =
    {
        Color.CYAN, new Color(25, 255, 25), new Color(25, 255, 25),
        Color.YELLOW, Color.YELLOW, Color.ORANGE, Color.ORANGE,
        Color.PINK, Color.RED, Color.RED
    };
    private final static Color[]    CONSEN_COLORS_12    =
    {
        Color.CYAN, new Color(25, 255, 25), new Color(25, 255, 25),
        Color.YELLOW, Color.YELLOW, Color.ORANGE, Color.ORANGE,
        Color.PINK, Color.PINK, Color.RED, Color.RED, Color.RED
    };
    private final static Map<Integer, Color[]>
                                    DATA_SET_SIZE_TO_CONSEN_COLORS;

    static
    {
        DATA_SET_SIZE_TO_CONSEN_COLORS = new HashMap<Integer, Color[]>();
        DATA_SET_SIZE_TO_CONSEN_COLORS.put(4, CONSEN_COLORS_4);
        DATA_SET_SIZE_TO_CONSEN_COLORS.put(6, CONSEN_COLORS_6);
        DATA_SET_SIZE_TO_CONSEN_COLORS.put(8, CONSEN_COLORS_8);
        DATA_SET_SIZE_TO_CONSEN_COLORS.put(10, CONSEN_COLORS_10);
        DATA_SET_SIZE_TO_CONSEN_COLORS.put(12, CONSEN_COLORS_12);
        for (int i: DATA_SET_SIZE_TO_CONSEN_COLORS.keySet())
            assert DATA_SET_SIZE_TO_CONSEN_COLORS.get(i).length == i;
    }


    private ArrayList<String>       msgAlignment;
    private Collection<String>      clustalAlignment;
    private int                     msgScore;
    private int                     clustalScore;
    private char[]                  gaOrClustalConsensus;           // evil global
    private int[]                   gaOrClustalConsensusDissention; // evil global


    ConsensusPanel(ArrayList<String> msgAlignment, Collection<String> clustalAlignment,
                   int msgScore, int clustalScore)
    {
        setParams(msgAlignment, clustalAlignment, msgScore, clustalScore);
    }


    public Dimension getPreferredSize()
    {
        return new Dimension(1000, 400);
    }


    void setParams(ArrayList<String> msgAlignment, Collection<String> clustalAlignment,
                   int msgScore, int clustalScore)
    {
        this.msgAlignment = msgAlignment;
        this.clustalAlignment = clustalAlignment;
        this.msgScore = msgScore;
        this.clustalScore = clustalScore;
        repaint();
    }


    private void paintConsensus(Graphics g, int y, Collection<String> text)
    {
        g.setFont(MSA_FONT);
        int x = LEFT_MARGIN;
        Color[] bgByNDissenters = DATA_SET_SIZE_TO_CONSEN_COLORS.get(text.size());

        for (int col=0; col<gaOrClustalConsensus.length; col++)
        {
            char ch = gaOrClustalConsensus[col];
            int nDissenters = gaOrClustalConsensusDissention[col];
            Color boxColor = (bgByNDissenters == null) ?
                                 Color.BLACK :
                                 bgByNDissenters[nDissenters];
            g.setColor(boxColor);
            g.fillRect(x, y-CONSEN_BOX_H+8, CONSEN_BOX_W, CONSEN_BOX_H);
            g.setColor(Color.BLACK);
            g.drawString(""+ch, x, y+4);
            x += CONSEN_BOX_W;
        }
```

```
}


public void paintComponent(Graphics g)
{
    g.setColor(Color.WHITE);
    g.fillRect(0, 0, 2000, 2000);

    // G.A. alignment.
    int y = paintMSAWithConsensus(g, TOP_BASELINE, msgAlignment, false);

    // ClustalW alignment.
    paintMSAWithConsensus(g, y+2*LINE_V_SPACING, clustalAlignment, true);
}


// Returns next y.
private int paintMSAWithConsensus(Graphics g, int y,
                                  Collection<String> text,
                                  boolean consensusOnTop)
{
    // Generate gaOrClustalConsensus[] & gaOrClustalConsensusDissention[]
    // for immediate use only.
    computeConsensus(text);

    // Paint consensus if it's on top.
    if (consensusOnTop)
    {
        paintConsensus(g, y, text);
        y += LINE_V_SPACING+3;
    }

    // Paint text.
    g.setFont(MSA_FONT);
    g.setColor(Color.BLACK);
    for (String s: text)
    {
        g.drawString(s, LEFT_MARGIN, y);
        y += LINE_V_SPACING;
    }

    // Paint consensus if it's on bottom.
    if (!consensusOnTop)
    {
        y += 3;
        paintConsensus(g, y, text);
    }

    // Paint score in large font.
    String s = text.iterator().next();
    int sw = g.getFontMetrics().stringWidth(s);
    int x = LEFT_MARGIN + sw + 35;
    boolean msgNotClustal = !consensusOnTop;
    paintScore(g, x, y, msgNotClustal);

    return y;
}
```

```
private void paintScore(Graphics g, int x, int baseline, boolean isMsg)
{
    if (!isMsg)
        baseline -= 22;
    g.setColor(Color.BLACK);
    g.setFont(SCORE_FONT);
    int score = isMsg ? msgScore : clustalScore;
    String s = isMsg ? "Our" : "ClustalW";
    s += " score = " + score;
    g.drawString(s, x, baseline);
}


// Sets global consensus string and dissenter counts.
private void computeConsensus(Collection<String> text)
{
    int nRows = text.size();
    int nCols = text.iterator().next().length();
    gaOrClustalConsensus = new char[nCols];
    gaOrClustalConsensusDissention = new int[nCols];
    for (int col=0; col<nCols; col++)
    {
        int[] counts = new int[26];
        int dashCount = 0;
        int winnerCount = 0;
        char winner = 0;
        for (String s: text)
        {
            char ch = s.charAt(col);
            int newCount = -12345;
            if (ch == '-')
                newCount = ++dashCount;
            else
                newCount = ++counts[ch-'A'];
            if (newCount > winnerCount)
            {
                winnerCount = newCount;
                winner = ch;
            }
        }
        gaOrClustalConsensus[col] = winner;
        gaOrClustalConsensusDissention[col] = nRows - winnerCount;
    }
}


private void paintConsensus(Graphics g, int y)
{
    assert msgAlignment != null  &&  msgAlignment.size() >= 4;

    g.setFont(MSA_FONT);
    int nCols = msgAlignment.get(0).length();
    int x = LEFT_MARGIN;
    for (int col=0; col<nCols; col++)
    {
        // Determine consensus char and degree of conservation.
        char majorityChar = 0;
```

```
            Map<Character, int[]> colCensus = new HashMap<Character, int[]>();
            int bestCount = 0;
            for (String s: msgAlignment)
            {
                char ch = s.charAt(col);
                int[] wrappedCount = colCensus.get(ch);
                if (wrappedCount == null)
                {
                    wrappedCount = new int[]{0};
                    colCensus.put(ch, wrappedCount);
                }
                wrappedCount[0]++;
                if (wrappedCount[0] > bestCount)
                {
                    bestCount = wrappedCount[0];
                    majorityChar = ch;
                }
            }

            // Paint consensus char on bg.
            int badness = msgAlignment.size() - bestCount;
            badness = Math.min(badness, CONSEN_COLORS.length-1);
            if (badness < 0)
            {
                String s = "alignment size = " + msgAlignment.size() +
                           ", best count = " + bestCount;
                assert false : s;
            }
            Color boxColor = CONSEN_COLORS[badness];
            g.setColor(boxColor);
            g.fillRect(x, y-CONSEN_BOX_H+8, CONSEN_BOX_W, CONSEN_BOX_H);
            g.setColor(Color.BLACK);
            g.drawString(""+majorityChar, x, y+4);
            x += CONSEN_BOX_W;
        }
    }
}
```

```
package msg;

import java.util.*;


class ConsensusWidthModel extends TreeMap<Integer, Integer>
{
    private ConsensusWidthPanel     view;
    private Integer                 lastKeyAdded;


    private ConsensusWidthModel(ConsensusWidthModel src)
    {
        super(src);
    }


    ConsensusWidthModel(ConsensusWidthPanel view)
    {
        this.view = view;
    }


    synchronized ConsensusWidthModel xerox()
    {
        return new ConsensusWidthModel(this);
    }


    // Repaints the view.
    public synchronized Integer put(Integer k, Integer v)
    {
        super.put(k, v);
        lastKeyAdded = k;
        view.repaint();
        return 0;
    }


    public String toString()
    {
        String s = "BinarySearchModel";
        for (int key: keySet())
            s += "\n  " + kvToString(key);
        return s;
    }


    private String kvToString(int k)
    {
        return "(" + k + ") = " + get(k);
    }


    int keyOfBestValue()
    {
        int bestVal = Integer.MIN_VALUE;
        int bestKey = Integer.MIN_VALUE;
    for (int key: keySet())
    {
        if (get(key) > bestVal)
        {
            bestKey = key;
            bestVal = get(key);
        }
    }
    return bestKey;
}


private int[] getBracketingXs(int x)
{
    assert containsKey(x);
    assert x > getMinX()  &&  x < getMaxX();
    ArrayList<Integer> keys = new ArrayList<Integer>(keySet());
    int indexOfKey = keys.indexOf(x);
    int prevKey = keys.get(indexOfKey-1);
    int nextKey = keys.get(indexOfKey+1);
    return new int[] {prevKey, nextKey};
}


// Utility for binary searching. Inserts newly computed (x,y). Returns next
// x to be computed, or min int if search has completed. Repaints the view.
int insertAndGetNextXForBinarySearch(int x, int y)
{
    put(x, y);
    int[] bracketingXs = getBracketingXs(x);

    // If yprev > ynext, next search is in [xprev .. x], otherwise next
    // search is in [x .. xnext]. In case of tie, favor xprev because smaller
    // x really means smaller consensus width, which is faster to compute.
    int[] nextRange = new int[2];
    if (get(bracketingXs[0]) > get(bracketingXs[1]))
    {
        nextRange[0] = bracketingXs[0];
        nextRange[1] = x;
    }
    else
    {
        nextRange[0] = x;
        nextRange[1] = bracketingXs[1];
    }

    assert nextRange[1] > nextRange[0];
    if (nextRange[1] - nextRange[0] == 1)
        return Integer.MIN_VALUE;
    else
        return (int)((nextRange[1] + nextRange[0]) / 2);
}


boolean isFinished()
{
    int k = keyOfBestValue();
    return containsKey(k-1) && containsKey(k+1);
    }


    int getMaxX()
    {
        return (Integer)keySet().toArray(new Integer[0])[size()-1];
    }


    int bestValue()             { return get(keyOfBestValue());      }
    int getMinX()               { return keySet().iterator().next(); }
    Integer getLastKeyAdded()   { return lastKeyAdded;               }
    static void sop(Object x)   { System.out.println(x);             }
}
```

```
package msg;

import java.util.*;
import java.awt.*;
import javax.swing.*;


class ConsensusWidthPanel extends JPanel
{
    private final static Font           FONT = new Font("Monospaced", Font.PLAIN, 12);
    private final static Color          BG = Color.WHITE;
    private final static Color          AXIS_COLOR = Color.BLACK;
    private final static Color          AXIS_TEXT_COLOR = Color.BLACK;
    private final static Color          DOT_COLOR = Color.BLUE;
    private final static Color          DOT_JOIN_COLOR = Color.BLUE.darker();
    private final static Color          CURRENT_COLOR = Color.MAGENTA;
    private final static Color          CLUSTAL_COLOR = new Color(128, 128, 128);
    private final static Color          LATEST_COLOR = new Color(105, 150, 255);
    private final static Color          WINNER_COLOR = Color.RED.darker();
    private final static Color          GRID_COLOR = Color.LIGHT_GRAY;
    private final static int            DOT_RADIUS = 4;
    private final static int            H_MARGIN = 60;
    private final static int            TOP_MARGIN = 40;
    private final static int            BOTTOM_TO_H_AXIS_LABEL_BASELINE = 20;
    private final static int            BOTTOM_TO_H_AXIS = 41;
    private final static int            GRAPH_W_PIX_PREF = 600;
    private final static int            GRAPH_H_PIX_PREF = 280;

    private float                       xUnitsPerPixel;
    private float                       yUnitsPerPixel;
    private int                         xCurrent;       // In abstract (non-pixel) units
    private ConsensusWidthModel         model;
    private int                         clustalScore;


    ConsensusWidthPanel()
    {
        setOpaque(true);
        setBackground(BG);
        model = new ConsensusWidthModel(this);
    }


    public Dimension getPreferredSize()
    {
        int wPref = H_MARGIN + GRAPH_W_PIX_PREF + H_MARGIN;
        int hPref = TOP_MARGIN + GRAPH_H_PIX_PREF + BOTTOM_TO_H_AXIS;
        return new Dimension(wPref, hPref);
    }

    public void paintComponent(Graphics g)
    {
        // If only 0 or 1 dots, there's not enough info to set H scale. It will
        // be set to something useless, and should not be used.
        adjustHScale();
        adjustVScale();

        // Clear
```

```
g.setColor(BG);
g.fillRect(0, 0, 2000, 2000);

// Use a threadsafe clone of the model.
ConsensusWidthModel safeModel = (model == null) ? null : model.xerox();

// Vertical grid.
if (safeModel != null  &&  safeModel.size() >= 2  &&  xUnitsPerPixel < 0.2f)
{
    int yTop = yToPix(clustalScore);
    int yBottom = getHeight() - BOTTOM_TO_H_AXIS;
    int xMinUnits = safeModel.getMinX();
    int xMaxUnits = safeModel.getMaxX();
    g.setColor(GRID_COLOR);
    for (int xUnits=xMinUnits; xUnits<=xMaxUnits; xUnits++)
    {
        int xPix = xToPix(xUnits);
        g.drawLine(xPix, yTop, xPix, yBottom);
    }
}

// Axes.
g.setColor(AXIS_COLOR);
int y = getHeight() - BOTTOM_TO_H_AXIS;
int r = getWidth() - H_MARGIN;
g.drawLine(H_MARGIN, y, r, y);
g.drawLine(H_MARGIN, y, H_MARGIN, TOP_MARGIN);
g.drawLine(r, y, r, TOP_MARGIN);

// Label axes if values at both ends have been computed.
g.setFont(FONT);
if (safeModel != null  &&  safeModel.size() >= 2)
{
    int xMinUnits = safeModel.getMinX();
    int xMaxUnits = safeModel.getMaxX();
    int xMinPix = xToPix(xMinUnits);
    int xMaxPix = xToPix(xMaxUnits);
    int[] xPixes = new int[] { xMinPix, xMaxPix };
    String[] ss = new String[2];
    ss[0] = "Min w = " + xMinUnits;
    ss[1] = "Max w = " + xMaxUnits;
    for (int i=0; i<2; i++)
    {
        int sw = g.getFontMetrics().stringWidth(ss[i]);
        int sx = xPixes[i] - sw/2;
        g.drawString(ss[i], sx, TOP_MARGIN-3);
    }
}

// Nothing else to do if no data yet.
if (safeModel == null  ||  safeModel.isEmpty())
    return;

// If multiple dots, join them. Draw these lines first, so they'll
// be overlaid by the dots themselves.
if (safeModel.size() >= 2)
{
    g.setColor(DOT_JOIN_COLOR);
    boolean first = true;
    int x0Pix = 0;
    int y0Pix = 0;
    for (int x1: safeModel.keySet())
    {
        int x1Pix = xToPix(x1);
        int y1Pix = yToPix(safeModel.get(x1));
        if (!first)
            g.drawLine(x0Pix, y0Pix, x1Pix, y1Pix);
        first = false;
        x0Pix = x1Pix;
        y0Pix = y1Pix;
    }
}

// ClustalW score.
if (clustalScore > 0)
{
    g.setColor(CLUSTAL_COLOR);
    y = yToPix(clustalScore);
    g.drawLine(0, y, 2000, y);
    String s = "ClustalW score = " + clustalScore;
    g.drawString(s, H_MARGIN+25, y-2);
}

// Mark most recent score.
Integer xLastUnitsWrapped = safeModel.getLastKeyAdded();
int xBestUnits = safeModel.keyOfBestValue();
int xBestPix = xToPix(xBestUnits);
if (xLastUnitsWrapped != null  &&  xLastUnitsWrapped != xBestUnits)
{
    int xLastUnits = xLastUnitsWrapped;
    int yLastUnits = safeModel.get(xLastUnits);
    int xLastPix = xToPix(xLastUnits);
    int yLastPix = yToPix(yLastUnits);
    g.setColor(LATEST_COLOR);
    g.drawLine(xLastPix, getHeight()-BOTTOM_TO_H_AXIS+25, xLastPix, yLastPix);
    String s = "Last score=" + yLastUnits + " at w=" + xLastUnits;
    int sw = g.getFontMetrics().stringWidth(s);
    int xText = xLastPix - sw/2;
    xText = Math.max(xText, 0);
    xText = Math.min(xText, getWidth()-sw/2);
    g.drawString(s, xText, getHeight()-BOTTOM_TO_H_AXIS+38);
}

// Mark top score so far.
int yBestUnits = safeModel.get(xBestUnits);
int yBestPix = yToPix(yBestUnits);
g.setColor(WINNER_COLOR);
g.drawLine(xBestPix, getHeight()-BOTTOM_TO_H_AXIS+14, xBestPix, yBestPix);
String s = "Top score=" + yBestUnits + " at w=" + xBestUnits;
g.setFont(FONT);
int sw = g.getFontMetrics().stringWidth(s);
int xText = xBestPix - sw/2;
xText = Math.max(xText, 5);
xText = Math.min(xText, getWidth()-sw/2-5);
g.drawString(s, xText, getHeight()-BOTTOM_TO_H_AXIS+25);
```

```
        // All dots. Min & max are labeled. Best is in special color.
        g.setColor(DOT_COLOR);
        FontMetrics fm = g.getFontMetrics();
        int xMinUnits = safeModel.getMinX();
        int xMaxUnits = safeModel.getMaxX();
        int stringW = fm.stringWidth(""+xMinUnits);
        Point textOffsetXMin = new Point(-DOT_RADIUS - stringW - 9, DOT_RADIUS + 2);
        Point textOffsetXMax = new Point(DOT_RADIUS + 2, DOT_RADIUS + 2);
        for (int x: safeModel.keySet())
        {
            Point textOffset = null;
            if (x == xMinUnits)
                textOffset = textOffsetXMin;
            else if (x == xMaxUnits)
                textOffset = textOffsetXMax;
            Color dotColor = DOT_COLOR;
            if (x == xBestUnits)
                dotColor = WINNER_COLOR;
            else if (xLastUnitsWrapped != null  &&  xLastUnitsWrapped != xBestUnits)
                dotColor = LATEST_COLOR;
            g.setColor(dotColor);
            paintDotAndValue(g, x, textOffset);
        }

        // X currently being computed.
        if (xCurrent > 0)
        {
            g.setColor(CURRENT_COLOR);
            int xPix = xToPix(xCurrent);
            g.drawLine(xPix, 0, xPix, 2000);
        }
    }


    private void adjustHScale()
    {
        if (model == null  ||  model.size() < 2)
        {
            xUnitsPerPixel = 12345;     // should never matter
            return;
        }

        float graphWUnits = model.getMaxX() - model.getMinX();
        float graphWPix = getWidth() - 2*H_MARGIN;
        xUnitsPerPixel = graphWUnits / graphWPix;
    }


    private void adjustVScale()
    {
        int bestGAScore = (model == null || model.isEmpty()) ? 1 : model.bestValue();
        float bestScore = Math.max(bestGAScore, clustalScore);
        float fUnitsPerPix = bestScore / GRAPH_H_PIX_PREF;
        yUnitsPerPixel = (float)Math.ceil(fUnitsPerPix);
    }


    // X is in arbitrary units, not pixels. Uses g's current color & font.
```

```
// Labels the point if textOffset is not null.
private void paintDotAndValue(Graphics g, int xUnits, Point textOffset)
{
    assert model.containsKey(xUnits) : "Model has no value for " + xUnits;

    int xPix = xToPix(xUnits);
    // Special case: put max x on right vertical axis. Roundoff would put it
    // a couple of puxels to the left.
    if (model.size() >= 2  &&  xUnits == model.getMaxX())
        xPix = getWidth() - H_MARGIN;
    int yUnits = model.get(xUnits);
    int yPix = yToPix(yUnits);
    g.fillOval(xPix-DOT_RADIUS, yPix-DOT_RADIUS, 2*DOT_RADIUS, 2*DOT_RADIUS);

    if (textOffset != null)
    {
        String s = "" + yUnits;
        g.drawString(s, xPix + textOffset.x, yPix + textOffset.y);
    }
}


private int xToPix(int xUnits)
{
    // Ensure max x is on right-side vertical axis. (Otherwise rounding
    // error makes ugly.)
    if (xUnits == model.getMaxX())
        return getWidth() - H_MARGIN;

    int xMin = model.getMinX();
    return H_MARGIN + Math.round((xUnits-xMin)/xUnitsPerPixel);
}


private int yToPix(int yUnits)
{
    yUnits = Math.max(yUnits, 0);
    return getHeight() - BOTTOM_TO_H_AXIS - Math.round(yUnits/yUnitsPerPixel);
}


void resetForNewDataSet()
{
    model = new ConsensusWidthModel(this);
    clustalScore = 0;
    repaint();
}


void setClustalScore(int c)
{
    clustalScore = c;
    repaint();
}


ConsensusWidthModel getModel()  { return model;                }
boolean isFinished()            { return model.isFinished();   }
```

```
    static void sop(Object x)       { System.out.println(x);       }


    public static void main(String[] args)
    {
        ConsensusWidthPanel cwp = new ConsensusWidthPanel();
        cwp.setClustalScore(350);
        ConsensusWidthModel model = cwp.getModel();
        model.put(10, 100);
        model.put(50, 500);
        model.put(20, 200);
        model.put(30, 300);
        model.put(40, 400);
        JFrame frame = new JFrame();
        frame.add(cwp, BorderLayout.CENTER);
        frame.pack();
        frame.setVisible(true);
    }
}
```

```
package msg;

import java.util.*;


class KickstartedPopulation extends Population
{
    private SequenceDataset          clustalSolution;


    KickstartedPopulation(int nChromosomes, int breedingPoolSize,
                          int nGapsPerChromosome,
                          UngappedSequenceDataset ungappedDataset,
                          SequenceDataset clustalSolution,
                          int consensusWidth, int historySize)
    {
        // Construct.
        this.nChromosomes = nChromosomes;
        this.breedingPoolSize = breedingPoolSize;
        this.ungappedDataset = ungappedDataset;
        this.consensusWidth = consensusWidth;
        this.history = new PopulationHistory(historySize);
        this.clustalSolution = clustalSolution;
        this.chromosomes = new ArrayList<Chromosome>(nChromosomes);

        // Build a chromosome to represent the ClustalW solution, adjusted
        // to the desired consensus width.
        assert clustalSolution.isUniformWidth();
        int deltaWidth = consensusWidth - clustalSolution.widthOfWidestSequence();
        SizeAdjustingChromosome starterChromo =
            new SizeAdjustingChromosome(clustalSolution, deltaWidth, nGapsPerChromosome);
        chromosomes.add(starterChromo);

        // Mutate the starter chromosome enough times to fill out the population.
        while (chromosomes.size() < nChromosomes)
        {
            Chromosome chromo = new Chromosome(starterChromo);
            chromo.mutate((float)Math.random());
            chromo.evaluate(ungappedDataset, consensusWidth);
            chromosomes.add(chromo);
        }

        initOperatorRates();
    }
}
```

```
package msg;

import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;


// Contains a BinarySearchPanel for top-level monitoring. Displays a non-modal
// dialog with a MultiTribePanel for monitoring run with current consensus width/
class MSGFrame extends JFrame implements ActionListener, ItemListener
{
    final static String     SEQUENCES_PARENT_PATH = "..\\..\\Data";
    final static File       SEQUENCES_DIRF = new File(SEQUENCES_PARENT_PATH);
    final static String     BALIBASE_DIR_NAME = "Balibase_bb3_release";
    final static File       BALIBASE_DIRF = new File(SEQUENCES_DIRF, BALIBASE_DIR_NAME);
    final static String     GAG_POL_DIR_NAME = "GagPol";
    final static File       GAG_POL_DIRF = new File(SEQUENCES_DIRF, GAG_POL_DIR_NAME);
    final static String     PDGH_DIR_NAME = "PDGH";
    final static File       PDGH_DIRF = new File(SEQUENCES_DIRF, PDGH_DIR_NAME);
    final static String     RANDOM_DIR_NAME = "Random";
    final static File       RANDOM_DIRF = new File(SEQUENCES_DIRF, RANDOM_DIR_NAME);

    private final static int[]      N_GENS_OPTIONS   = { 3, 100, 200, 300, 500,
                                                         750, 1000, 2000 };
    private final static int[]      N_TRIBES_OPTIONS = { 1, 10, 20 };

    private JMenuItem               quitMI;
    private ConsensusWidthPanel     conWidthPan;
    private MultiTribePanel         multiTribePan;
    private SequenceDataset         clustalGappedDataset;
    private UngappedSequenceDataset ungappedDataset;
    private int                     clustalScore;
    private MSGFrame                outerThis;
    private JComboBox               tribeGenerationsCombo;
    private JComboBox               combinedGenerationsCombo;
    private JComboBox               nTribesCombo;
    private ConsensusDialog         consensusDia;
    private JCheckBox               darkBGCbox;


    public MSGFrame()
    {
        JMenuBar mbar = new JMenuBar();
        try
        {
            mbar.add(buildFileMenu());
        }
        catch (IOException x)
        {
            sop("Can't open dataset dir: " + x.getMessage());
            x.printStackTrace(System.out);
        }
        setJMenuBar(mbar);

        setLayout(new BorderLayout());
        JPanel controlsAndConWidth = new JPanel(new BorderLayout());
        JPanel controls = new JPanel();
        controls.add(new JLabel("# Tribes: "));
        nTribesCombo = buildIntCombo(N_TRIBES_OPTIONS);
        nTribesCombo.setSelectedIndex(1);
        controls.add(nTribesCombo);
        controls.add(new JLabel("# Gens Tribe Phase: "));
        tribeGenerationsCombo = buildNGensCombo();
        tribeGenerationsCombo.setSelectedItem(750);
        controls.add(tribeGenerationsCombo);
        controls.add(new JLabel("# Gens Combined Phase: "));
        combinedGenerationsCombo = buildNGensCombo();
        combinedGenerationsCombo.setSelectedItem(750);
        controls.add(combinedGenerationsCombo);
        darkBGCbox = new JCheckBox("Dark bgnd");
        darkBGCbox.addItemListener(this);
        controls.add(darkBGCbox);
        controlsAndConWidth.add(controls);
        conWidthPan = new ConsensusWidthPanel();
        controlsAndConWidth.add(conWidthPan, BorderLayout.SOUTH);
        add(controlsAndConWidth, BorderLayout.NORTH);
        multiTribePan = new MultiTribePanel(getNTribes());
        multiTribePan.setNGenerationsTribePhase(getNGensTribe());
        multiTribePan.setNGenerationsCombinedPhase(getNGensCombined());
        add(multiTribePan, BorderLayout.SOUTH);

        pack();

        outerThis = this;
    }


    private JComboBox buildNGensCombo()
    {
        return buildIntCombo(N_GENS_OPTIONS);
    }


    private JComboBox buildIntCombo(int[] vals)
    {
        JComboBox combo = new JComboBox();
        for (int i: vals)
            combo.addItem(i);
        combo.addItemListener(this);
        return combo;
    }


    private int getNGensTribe()
    {
        return (Integer)tribeGenerationsCombo.getSelectedItem();
    }


    private int getNGensCombined()
    {
        return (Integer)combinedGenerationsCombo.getSelectedItem();
    }
```

```
private int getNTribes()
{
    return (Integer)nTribesCombo.getSelectedItem();
}


private JMenu buildFileMenu() throws IOException
{
    JMenu menu = new JMenu("File");

    // OPEN submenu.
    JMenu openMenu = new JMenu("Open");
    String[] subsubs = { "Balibase", "Gag/Pol", "PDGH", "Random" };
    File[] dirfs = { BALIBASE_DIRF, GAG_POL_DIRF, PDGH_DIRF, RANDOM_DIRF };
    for (int i=0; i<dirfs.length; i++)
    {
        JMenu subMenu = new JMenu(subsubs[i]);
        File dirf = dirfs[i];
        String[] contents = dirf.list();
        Set<String> clwSorter = new TreeSet<String>();
        for (String kid: contents)
            if (kid.endsWith(".clw"))
                clwSorter.add(kid);
        for (String kid: clwSorter)
        {
            String title = kid.substring(0, kid.length()-4);
            File kidFile = new File(dirf, kid);
            subMenu.add(new DatasetMenuItem(title, kidFile));
        }
        openMenu.add(subMenu);
    }
    menu.add(openMenu);

    // EXIT item.
    menu.addSeparator();
    quitMI = new JMenuItem("Exit");
    quitMI.addActionListener(this);
    menu.add(quitMI);
    return menu;
}


private class DatasetMenuItem extends JMenuItem implements ActionListener
{
    String      displayName;
    File        file;

    DatasetMenuItem(String displayName, File file)
    {
        super(displayName);
        this.displayName = displayName;
        this.file = file;
        addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
            try
            {
                assert file.exists();
                clustalGappedDataset = ClustalParser.parseFileToGapped(file);
                ungappedDataset = clustalGappedDataset.removeGaps();
                clustalScore =
AlignmentScorer.scoreAlignment(clustalGappedDataset.values());
                conWidthPan.resetForNewDataSet();
                conWidthPan.setClustalScore(clustalScore);
                setTitle("Analyzing " + ungappedDataset.getName());
                (new TopLevelThread()).start();
              }
              catch (IOException x)
              {
                  sop("Can't open dataset " + getText() + ": " + x.getMessage());
                  x.printStackTrace(System.out);
              }
          }
    }


    public void actionPerformed(ActionEvent e)
    {
        if (e.getSource() == quitMI)
            System.exit(0);
    }


    public void itemStateChanged(ItemEvent e)
    {
        if (e.getSource() == tribeGenerationsCombo)
        {
            if (multiTribePan != null)
                multiTribePan.setNGenerationsTribePhase(getNGensTribe());
        }

        else if (e.getSource() == tribeGenerationsCombo)
        {
            if (multiTribePan != null)
                multiTribePan.setNGenerationsCombinedPhase(getNGensCombined());
        }

        else if (e.getSource() == darkBGCbox)
        {
            multiTribePan.setDarkBG(darkBGCbox.isSelected());
            multiTribePan.repaint();
        }
    }


    static void snooze(int msecs)
    {
        try
        {
            Thread.sleep(msecs);
        }
        catch (InterruptedException x) { }
    }
    // For consistency checking. Returns true if compo is ultimately contained
    // in this JFrame object.
    private boolean isUltimateChild(Component compo)
    {
        while (compo != null  &&  compo != outerThis)
            compo = compo.getParent();
        return compo == outerThis;
    }


    private Vector<Integer> getWidthProgram()
    {
        int wClustal = clustalGappedDataset.widthOfWidestSequence();
        Stack<Integer> stack = new Stack<Integer>();
        int min = Math.max(wClustal-10, ungappedDataset.widthOfWidestSequence());
        stack.add(min);
        stack.add(wClustal + 10);
        stack.add(wClustal);
        for (int w=min+1; w<wClustal+10; w++)
            if (w != wClustal)
                stack.add(w);
        return stack;
    }


    private class TopLevelThread extends Thread
    {
        public void run()
        {
            assert multiTribePan != null;
            assert isUltimateChild(multiTribePan)  :
                   "multiTribePan isn't ultimately contained by frame";

            setPriority(2); // Don't crowd out the GUI thread or the GC
            Vector<Integer> widths = getWidthProgram();
            ConsensusWidthModel conWidthModel = conWidthPan.getModel();
            Chromosome fittestChromo = null;
            int fittestScore = Integer.MIN_VALUE;
            for (int width: widths)
            {
                Chromosome fittestChromoForWidth = computeGAScoreForWidth(width);
                conWidthModel.put(width, fittestChromoForWidth.score);
                if (fittestChromoForWidth.score > fittestScore)
                {
                    fittestScore = fittestChromoForWidth.score;
                    fittestChromo = fittestChromoForWidth;
                }
            }

            // Display & print our & clustal alignments.
            String dataset = ungappedDataset.getName();
            int nTribes = multiTribePan.getNTribes();
            int width = conWidthModel.keyOfBestValue();
            sop("Finished analysis of " + dataset);
            ArrayList<String> msgAlignment = fittestChromo.toMSAStrings(ungappedDataset,
width);
```

```
            Collection<String> clustalAlignment = clustalGappedDataset.values();
            if (consensusDia == null)
                consensusDia = new ConsensusDialog(msgAlignment, clustalAlignment,
                                                   fittestScore, clustalScore);
            else
                consensusDia.setParams(msgAlignment, clustalAlignment, fittestScore,
clustalScore);
            consensusDia.setVisible(true);
            sop("\nMSG Alignment (score = " + fittestScore + "):");
            for (String s: msgAlignment)
                sop(s);
            sop("\nClustal-W Alignment (score = " + clustalScore + "):");
            for (String s: clustalAlignment)
                sop(s);
        }

        private Chromosome computeGAScoreForWidth(int consensusWidth)
        {
            // Reset the multi-tribe panel for next consensus width.
            multiTribePan.setNTribes(getNTribes());
            multiTribePan.setNGenerationsTribePhase(getNGensTribe());
            multiTribePan.setNGenerationsCombinedPhase(getNGensCombined());
            multiTribePan.setNTribes(getNTribes());
            multiTribePan.reset(ungappedDataset, clustalGappedDataset,
                                clustalScore, consensusWidth);
            return multiTribePan.computeGAScore();
        }
    }


    static void sop(Object x)       { System.out.println(x); }
    static void snooze()            { snooze(1000);          }


    public static void main(String[] args)
    {
        MSGFrame frame = new MSGFrame();
        frame.setLocation(10, 10);
        frame.setVisible(true);
    }
}
```

```
package msg;

import java.awt.*;
import javax.swing.*;
import java.util.*;

class MultiTribePanel extends JPanel
{
    private final static int        PREF_W              = 1000;
    private final static int        PREF_H              =  360;
    private final static int        H_MARGIN            =   75;
    private final static int        GRAPH_W             = PREF_W - 2*H_MARGIN;
    private final static Color      CLUSTAL_COLOR       = Color.LIGHT_GRAY;
    private final static Color      AXIS_COLOR          = Color.BLACK;
    private final static Color      BG                  = new Color(210, 220, 255);
    private final static Color      MERGED_COLOR        = Color.RED.darker();
    private final static int        X_OF_Y_AXIS         =   35;
    private final static int        X_AXIS_TO_BOTTOM    =   25;
    private final static int        MAX_TO_TOP          =   40;
    private final static int        DFLT_N_TRIBES       =   20;
    private final static int        TRIBE_SIZE          =  100;
    private final static int        BREEDING_POOL_SIZE  =   50;
    private final static Font       BIG_FONT            = new Font("Serif", Font.PLAIN,
32);
    private final static int        TEXT_V_SPACING      =   30;
    private final static int        SCORE_PLACEHOLDER   = Integer.MIN_VALUE;

    private Phase                   phase;
    private SequenceDataset         clustalAlignment;
    private int                     clustalScore;
    private int                     consensusWidth;
    private int                     nTribes;
    private Population[]            tribes;
    private Color[]                 tribeColors;
    private Population              mergedTribes;
    private float                   vertPixPerScoreUnit;
    private int                     nGenerationsTribePhase;
    private int                     nGenerationsCombinedPhase;
    private boolean                 darkBG;


    private enum Phase { PRIMORDIAL, TRIBES, MERGED, DONE }


    MultiTribePanel(int nTribes)
    {
        this.nTribes = nTribes;
        phase = Phase.PRIMORDIAL;
    }


    synchronized void reset(UngappedSequenceDataset seqNameToUngapped,
                            SequenceDataset clustalAlignment,
                            int clustalScore, int consensusWidth)
    {
        assert phase == Phase.PRIMORDIAL  ||  phase == Phase.DONE;
        phase = Phase.TRIBES;
        this.clustalAlignment = clustalAlignment;
        this.clustalScore = clustalScore;
        this.consensusWidth = consensusWidth;

        // All but the last tribe are ordinary.
        tribes = new Population[nTribes];
        mergedTribes = null;
        for (int i=0; i<nTribes-1; i++)
        {
            tribes[i] = new Population(TRIBE_SIZE,
                                       BREEDING_POOL_SIZE,
                                       seqNameToUngapped,
                                       consensusWidth,
                                       GRAPH_W);
        }
        int nGapsPerChromosome = tribes[0].getChromosomeLength();

        // Build 1 "kickstarted" population, initialized from the Clustal
        // alignment. If the desired width is << the clustal width, a kickstarted
        // population might be impossible; in this case the last tribe defaults
        // to an ordinary random population as above.
        tribes[nTribes-1] = new KickstartedPopulation(TRIBE_SIZE,
                                              BREEDING_POOL_SIZE,
                                              nGapsPerChromosome,
                                              seqNameToUngapped,
                                              clustalAlignment,
                                              consensusWidth,
                                              GRAPH_W);

        if (tribeColors == null)
        {
            tribeColors = new Color[nTribes];
            float deltaHue = 1f / nTribes;
            float hue = 0.05f;
            for (int i=0; i<nTribes; i++)
            {
                int hsb = Color.HSBtoRGB(hue, 1, 1);
                tribeColors[i] = new Color(hsb);
                hue += deltaHue;
            }
        }

        repaint();
    }


    public Dimension getPreferredSize()
    {
        return new Dimension(PREF_W, PREF_H);
    }


    public void paintComponent(Graphics g)
    {
        g.setColor(darkBG ? Color.BLACK : BG);
        g.fillRect(0, 0, 2000, 2000);

        switch (phase)
    {
        case PRIMORDIAL:
            return;
        case TRIBES:
            paintTribes(g);
            break;
        case MERGED:
            paintMerged(g);
            break;
        case DONE:
            paintDone(g);
            break;
    }
}


private void paintTribes(Graphics g)
{
    assert phase == Phase.TRIBES;

    computeVScaleAndPaintNonHistoric(g);

    // Internal consistency is iffy during tribes->merged phase transition.
    // If tribes becomes null, it's not really serious.
    try
    {
        paintGenerationAndScore(g);
        for (int i=0; i<tribes.length; i++)
        {
            Population tribe = tribes[i];
            g.setColor(tribeColors[i]);
            PopulationHistory safeHistory = tribe.getHistory().xeroxThreadsafe();
            paintHistory(g, safeHistory);
        }
    }
    catch (NullPointerException x) { }
}


private void paintMerged(Graphics g)
{
    try
    {
        computeVScaleAndPaintNonHistoric(g);
        paintGenerationAndScore(g);
        g.setColor(MERGED_COLOR);
        paintHistory(g, mergedTribes.getHistory().xeroxThreadsafe());
    }
    catch (NullPointerException x) { }
}


// Just paint final merged history. Won't be visible long (unless this is
// the last consensus width we try) because thread in frame is about to
// reset us.
private void paintDone(Graphics g)
{
    paintMerged(g);
```

```
}


// Caller should set color prior.
private void paintHistory(Graphics g, PopulationHistory history)
{
    if (history.size() < 2)
        return;
    int x0 = H_MARGIN;
    int y0 = scoreToPix(history.get(0));
    for (int j=1; j<history.size(); j++)
    {
        int x1 = H_MARGIN + j;
        int score = history.get(j);
        if (score == SCORE_PLACEHOLDER)
            continue;
        int y1 = scoreToPix(score);
        g.drawLine(x0, y0, x1, y1);
        x0 = x1;
        y0 = y1;
    }
}


private void computeVScaleAndPaintNonHistoric(Graphics g)
{
    // Axes.
    g.setColor(AXIS_COLOR);
    Dimension size = getSize();
    g.drawLine(0, size.height-X_AXIS_TO_BOTTOM, 2000, size.height-X_AXIS_TO_BOTTOM);
    g.drawLine(X_OF_Y_AXIS, 0, X_OF_Y_AXIS, 2000);

    // Compute vertical scale.
    int maxScore = Math.max(clustalScore, bestGAScore());
    float vertPixelsForGraphing = size.height - MAX_TO_TOP - X_AXIS_TO_BOTTOM;
    vertPixPerScoreUnit = vertPixelsForGraphing / maxScore;

    // ClustalScore
    g.setColor(CLUSTAL_COLOR);
    int clustalPix = scoreToPix(clustalScore);
    g.drawLine(0, clustalPix, 2000, clustalPix);
}


private void paintGenerationAndScore(Graphics g)
{
    int bestGAScore = bestGAScore();
    if (bestGAScore == Integer.MIN_VALUE)
        return;

    g.setFont(BIG_FONT);
    g.setColor(darkBG ? Color.YELLOW : Color.BLACK);
    int generation = getGeneration();
    String s = "Generation = " + generation;
    int y = 250;
    if (generation != -1)
        g.drawString(s, 100, y);
    y += TEXT_V_SPACING;
```

```
        s = "Score = " + bestGAScore + " [w=" + consensusWidth + "]";
        g.drawString(s, 100, y);
        y += TEXT_V_SPACING;
        s = "ClustalW Score = " + clustalScore +
            " [w=" + clustalAlignment.widthOfWidestSequence() + "]";
        g.drawString(s, 100, y);
    }


    // Returns -1 if internally inconsistent. This can happen during switchover
    // from tribes to merged phase.
    private int getGeneration()
    {
        try
        {
            switch (phase)
            {
                case PRIMORDIAL:
                    assert false;
                    throw new IllegalStateException();
                case TRIBES:
                    return tribes[0].getHistory().size();
                default:
                    return nGenerationsTribePhase + mergedTribes.getHistory().size();
            }
        }
        catch (NullPointerException x)
        {
            return -1;
        }
    }


    // Non-negative.
    private synchronized int bestGAScore()
    {
        int best = Integer.MIN_VALUE;
        switch (phase)
        {
            case PRIMORDIAL:
                throw new IllegalStateException("Called bestGAScore() in PRIMORDIAL
phase.");
            case TRIBES:
                assert tribes != null  :  "Unexpected null tribes[]";
                for (Population tribe: tribes)
                {
                    assert tribe != null  :  "Null tribe";
                    assert tribe.getHistory() != null  :  "Tribe has null history";
                    best = Math.max(best, tribe.getMaxScore());
                }
                break;
            default:
                best = mergedTribes.getMaxScore();
                break;
        }
        return best;
    }
```

```
private int scoreToPix(int score)
{
    int relPix = (int)(score*vertPixPerScoreUnit);
    return getHeight() - X_AXIS_TO_BOTTOM - relPix;
}


private void computeMultiTribePhase()
{
    assert phase == Phase.TRIBES;       // Set by reset()

    for (int i=0; i<nGenerationsTribePhase; i++)
    {
        for (Population tribe: tribes)
            tribe.step1Generation();    // records best score into history
        repaint();
        Thread.yield();
    }
    nGenerationsTribePhase = tribes[0].getHistory().size();
}


private void computeMergedPhase()
{
    // Merge all tribes into a single diverse population. From now on,
    // tribes[] is untouchable. Add placeholders to merged population's
    // history; this will position its entries to the right of the tribes
    // when the graph is drawn.
    assert tribes != null  :  "Null tribes[] in computeMergedPhase().";
    mergedTribes = new Population(tribes);
    tribes = null;
    phase = Phase.MERGED;

    // Step the merged population.
    for (int i=0; i<nGenerationsCombinedPhase; i++)
    {
        mergedTribes.step1Generation();    // records best score into history
        repaint();
        Thread.yield();
    }

    phase = Phase.DONE;
    repaint();
}


// Must be called from a safe thread. Returns chromosome with best score.
// The score is stored in the chromosome.
Chromosome computeGAScore()
{
    assert phase == Phase.TRIBES;
    assert consensusWidth > 0;
    computeMultiTribePhase();
    computeMergedPhase();
    return mergedTribes.getFittest();
}
```

```
    static void sop(Object x)                   { System.out.println(x);          }
    int getNTribes()                            { return nTribes;                 }
    void setNTribes(int n)                      { nTribes = n;                    }
    void setNGenerationsTribePhase(int n)       { nGenerationsTribePhase = n;     }
    void setNGenerationsCombinedPhase(int n)    { nGenerationsCombinedPhase = n; }
    void setDarkBG(boolean b)                   { darkBG = b;                     }
}
```

```
package msg;

import java.util.*;


class Population
{
    private final static float          DFLT_CROSSOVER_RATE     = .07f;
    private final static float          DFLT_MUTATION_RATE      = .20f;
    private final static int            ACTIVITY_CHECKIN_PERIOD = 20;
    private final static float          OPERATOR_SPEEDUP        = 1.1f;

    protected int                       nChromosomes;
    protected ArrayList<Chromosome>     chromosomes;
    protected UngappedSequenceDataset   ungappedDataset;
    protected int                       consensusWidth;
    protected int                       breedingPoolSize;
    protected PopulationHistory         history;
    private float                       mutationRate;
    private float                       crossoverRate;


    Population() { }


    // Randomizes. Default crossover & mutation rates.
    Population(int nChromosomes, int breedingPoolSize,
               UngappedSequenceDataset ungappedDataset,
               int consensusWidth, int historySize)
    {
        this.nChromosomes = nChromosomes;
        this.breedingPoolSize = breedingPoolSize;
        this.ungappedDataset = ungappedDataset;
        this.consensusWidth = consensusWidth;
        chromosomes = new ArrayList<Chromosome>(nChromosomes);

        history = new PopulationHistory(historySize);

        // Build, evaluate, and add randomized chromosomes.
        int nSeqs = ungappedDataset.size();
        int nGapsTotal = consensusWidth*nSeqs - ungappedDataset.nCharsOverall();
        for (int i=0; i<nChromosomes; i++)
        {
            Chromosome chr = new Chromosome(nGapsTotal, consensusWidth-1);
            chr.randomize();
            chr.evaluate(ungappedDataset, consensusWidth);
            chromosomes.add(chr);
        }

        // Record best score into history.
        history.add(getFittest().score);

        initOperatorRates();
    }


    // Merges top performers in tribes[] into a single diverse population.
    Population(Population[] tribes)
    {
        assert tribes != null  :  "Null tribes[] in Population ctor.";
        for (Population tribe: tribes)
            assert tribe != null : "Null tribe in Population ctor.";

        // Copy instance vars from one of the tribes.
        this.nChromosomes = tribes[0].nChromosomes;
        this.ungappedDataset = tribes[0].ungappedDataset;
        this.consensusWidth = tribes[0].consensusWidth;
        this.breedingPoolSize = tribes[0].breedingPoolSize;

        // Collect top members of each tribe. Cache single best member in each
        // tribe, in case we need filler.
        int nRepresentativesPerTribe = nChromosomes / tribes.length;
        chromosomes = new ArrayList<Chromosome>(nChromosomes);
        Vector<Chromosome> filler = new Vector<Chromosome>();
        for (Population tribe: tribes)
        {
            List<Chromosome> topN = tribe.topNChromosomes(nRepresentativesPerTribe);
            chromosomes.addAll(topN);
            filler.add(topN.get(0));
        }

        // Might need a few more chromosomes, due to rounding error in
nRepresentativesPerTribe.
        int fillerIndex = 0;
        while (chromosomes.size() < nChromosomes)
        {
            chromosomes.add(filler.get(fillerIndex));
            fillerIndex = (fillerIndex + 1) % filler.size();
        }

        // Record best score into history.
        history = new PopulationHistory(tribes[0].getHistory().getMaxSize());
        history.add(getFittest().score);

        initOperatorRates();
    }


    protected void initOperatorRates()
    {
        mutationRate = DFLT_MUTATION_RATE;
        crossoverRate = DFLT_CROSSOVER_RATE;
    }


    protected Chromosome[] sortDescending()
    {
        TreeSet<Chromosome> ascending = new TreeSet<Chromosome>(chromosomes);
        Chromosome[] descending = new Chromosome[ascending.size()];
        int n = descending.length - 1;
        for (Chromosome chro: ascending)
            descending[n--] = chro;
        return descending;
    }
    // Records top score in history.
    void step1Generation()
    {
        // Collect chromosomes in descending fitness order (best is at [0]).
        Chromosome[] sortedDescending = sortDescending();
        assert sortedDescending.length == chromosomes.size() :
               "Unexpected length of sorted = " + sortedDescending.length +
               " != chromosomes.size() = " + chromosomes.size();

        // Clear out main collection (safe, because all members are in
sortedDescending[]).
        chromosomes.clear();

        // Elitism: best 2 members bypass breeding pool and go directly into
        // next generation. They may also breed if the wheel choses them.
        chromosomes.add(sortedDescending[0]);
        chromosomes.add(sortedDescending[1]);

        // Spin roulette wheel to determine indices of mating pairs.
        int[] randomizedBreederIndices = RouletteWheel.spin(nChromosomes);

        // Breed.
        int n = 0;
        while (chromosomes.size() < nChromosomes)
        {
            // Get parents.
            int maIndex = randomizedBreederIndices[n++];
            Chromosome ma = sortedDescending[maIndex];
            int paIndex = randomizedBreederIndices[n++];
            Chromosome pa = sortedDescending[paIndex];
            // Crossover.
            Chromosome[] kids = Chromosome.crossover(ma, pa, crossoverRate);
            // Mutate.
            kids[0].mutate(mutationRate);
            kids[1].mutate(mutationRate);
            // Evaluate.
            kids[0].evaluate(ungappedDataset, consensusWidth);
            kids[1].evaluate(ungappedDataset, consensusWidth);
            // Collect.
            chromosomes.add(kids[0]);
            chromosomes.add(kids[1]);
        }

        // Record best score into history.
        int currentScore = getFittest().score;
        history.add(currentScore);

        // Adjust operator rates.
        int histoSize = history.size();
        if (histoSize >= ACTIVITY_CHECKIN_PERIOD  &&
            histoSize % ACTIVITY_CHECKIN_PERIOD  == 0)
        {
            int earlierScore = history.get(histoSize-ACTIVITY_CHECKIN_PERIOD);
            if (earlierScore == currentScore)
            {
                // Flatlined => increase operator likelihoods.
                mutationRate *= OPERATOR_SPEEDUP;
                crossoverRate *= OPERATOR_SPEEDUP;
            }
            else if (currentScore-earlierScore > OPERATOR_SPEEDUP)
            {
                // Don't sustain high operator rates for too long.
                mutationRate /= OPERATOR_SPEEDUP;
                crossoverRate /= OPERATOR_SPEEDUP;
            }

        }
    }


    List<Chromosome> topNChromosomes(int n)
    {
        assert chromosomes.size() >= n;
        Chromosome[] allSorted = sortDescending();
        assert allSorted.length >= n;
        ArrayList<Chromosome> ret = new ArrayList<Chromosome>(n);
        for (int i=0; i<n; i++)
            ret.add(allSorted[i]);
        return ret;
    }


    Chromosome getFittest()
    {
        return sortDescending()[0];
    }


    int getChromosomeLengthThrowIfNotUniform() throws IllegalStateException
    {
        int w = chromosomes.iterator().next().length();
        for (Chromosome chromo: chromosomes)
            if (chromo.length() != w)
                throw new IllegalStateException(chromo.length() + " != " + w);
        return w;
    }


    int getChromosomeLength()
    {
        return chromosomes.get(0).length();
    }


    PopulationHistory getHistory()        { return history;                }
    int getMaxScore()                     { return history.getMaxScore(); }
    static void sop(Object x)             { System.out.println(x);        }
}
```

```
package msg;

import java.util.*;


class PopulationHistory extends ArrayList<Integer>
{
    private int             maxSize;


    PopulationHistory(int maxSize)            { this.maxSize = maxSize; }
    PopulationHistory(PopulationHistory src)  { super(src);             }
    int getMaxSize()                          { return maxSize;         }


    public synchronized boolean add(Integer addMe)
    {
        // If full, remove all odd-indexed elements.
        if (size() == maxSize)
            for (int n=size()-1; n>0; n-=2)
                remove(n);

        return super.add(addMe);
    }


    public synchronized PopulationHistory xeroxThreadsafe()
    {
        return new PopulationHistory(this);
    }


    synchronized int getMaxScore()
    {
        int score = Integer.MIN_VALUE;
        for (int i: this)
            score = Math.max(i, score);
        return score;
    }
}
```

```
package msg;

import java.io.*;


class RandomFastaGenerator
{
    private final static int[]      SKEWS       = { 0, 2, 5, 7 };
    private final static String[]   FNAMES      = { "Uniform", "LightSkew",
                                                    "ModerateSkew", "IntenseSkew" };


    private static void generate(File dirf) throws IOException
    {
        assert dirf.exists() : "No such dir: " + dirf.getAbsolutePath();
        for (int i=0; i<SKEWS.length; i++)
            generate(new File(dirf, FNAMES[i]+".fasta"), 6, SKEWS[i]);
    }


    private static void generate(File file, int nSeqs, int skew) throws IOException
    {
        FileWriter fw = new FileWriter(file);
        PrintWriter pw = new PrintWriter(fw, true);     // true for autoflush

        for (int i=0; i<nSeqs; i++)
        {
            String name = "Random_" + i;
            pw.write(">" + name + "\r\n");
            int len = 50 - skew*i;
            assert len > 0;
            StringBuilder sb = new StringBuilder();
            for (int j=0; j<len; j++)
                sb.append(randomAA());
            pw.print(sb + "\r\n");
        }

        pw.close();
        fw.close();
    }


    private final static String AA_CHARS = "ACDEFGHIKLMNPQRSTVWY";


    private static char randomAA()
    {
        int index = (int)(AA_CHARS.length() * Math.random());
        return AA_CHARS.charAt(index);
    }


    public static void main(String[] args)
    {
        File dirf = MSGFrame.RANDOM_DIRF;
        try
        {
            generate(dirf);
            System.out.println("Done");
        }
        catch (IOException x)
        {
            System.out.println("IO Stress: " + x.getMessage());
            x.printStackTrace(System.out);
        }
    }
}
```

```
package msg;

class RouletteWheel
{
    // Assumes generation size = 100. Index is rank (0 is best).
    private final static int[]      RELATIVE_ODDS =
        { 5, 5, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
};

    static int[]                    THE_WHEEL;

    static
    {
        THE_WHEEL = new int[50];
        int n = 0;
        for (int winnerIndex=0; winnerIndex<RELATIVE_ODDS.length; winnerIndex++)
            for (int i=0; i<RELATIVE_ODDS[winnerIndex]; i++)
                THE_WHEEL[n++] = winnerIndex;
        assert n == THE_WHEEL.length;
    }


    static int[] spin(int matingPoolSize)
    {
        int[] winners = new int[matingPoolSize];
        for (int i=0; i<matingPoolSize; i++)
        {
            int randy = (int)(THE_WHEEL.length * Math.random());
            winners[i] = THE_WHEEL[randy];
        }
        return winners;
    }


    public static void main(String[] args)
    {
        int[] winners = spin(50);
        String s = "";
        for (int i: winners)
            s += i + ",";
        System.out.println(s);
    }
}
```

```
package msg;

import java.util.*;


// Extends a map from sequence name to sequence.
class SequenceDataset extends TreeMap<String, String>
{
    private String      name;


    SequenceDataset(String name)        { this.name = name; }


    UngappedSequenceDataset removeGaps()
    {
        UngappedSequenceDataset ret = new UngappedSequenceDataset(name);
        for (String k: keySet())
        {
            String gapped = get(k);
            StringBuilder sb = new StringBuilder();
            for (int i=0; i<gapped.length(); i++)
                if (gapped.charAt(i) != '-')
                    sb.append(gapped.charAt(i));
            ret.put(k, sb.toString());
        }
        return ret;
    }


    public String toString()
    {
        String s = "Sequence Dataset " + name;
        int longestKeyLen = -1;
        for (String k: keySet())
            longestKeyLen= Math.max(longestKeyLen, k.length());
        for (String k: keySet())
        {
            int nSpaces = longestKeyLen - k.length();
            s += "\n" + k + ": ";
            for (int i=0; i<nSpaces; i++)
                s += " ";
            s += get(k);
        }
        return s;
    }


    // With a proper global alignment all sequences are the same width, but
    // you never can tell.
    int widthOfWidestSequence()
    {
        int w = -1;
        for (String seq: values())
            w = Math.max(w, seq.length());
        return w;
    }
    int countGaps()
    {
        int n = 0;
        for (String s: values())
        {
            for (int i=0; i<s.length(); i++)
            {
                if (s.charAt(i) == '-')
                    n++;
            }
        }
        return n;
    }


    boolean isUniformWidth()
    {
        String s = values().iterator().next();
        int w = s.length();
        for (String s1: values())
            if (s1.length() != w)
                return false;
        return true;
    }


    String getName()        { return name; }
}
```

```
package msg;

import java.util.*;
import java.io.*;


class SizeAdjustingChromosome extends Chromosome
{
    // Constructs a chromosome that represents the result of adjusting the
    // width of src by deltaWidth. If widening, columns of gaps are added
    // fore & aft. If narrowing, columns are deleted.
    SizeAdjustingChromosome(SequenceDataset src, int deltaWidth, int nGapsAfter)
    {
        assert src.isUniformWidth();

        this.maxGapIndex = nGapsAfter - 1;

        if (deltaWidth >= 0)
        {
            // Convert source to array of StringBuilder!s.
            StringBuilder[] sbs = new StringBuilder[src.size()];
            int n = 0;
            for (String s: src.values())
                sbs[n++] = new StringBuilder(s);

            // Add columns.
            bookendNGapCols(sbs, deltaWidth);

            // Compute gap locations.
            this.maxGapIndex = sbs[0].length() - 1;
            int nGaps = 0;
            for (StringBuilder sb: sbs)
                for (int i=0; i<sb.length(); i++)
                    if (sb.charAt(i) == '-')
                        nGaps++;
            assert nGaps == nGapsAfter;
            this.gapLocations = new int[nGaps];
            n = 0;
            for (StringBuilder sb: sbs)
                for (int i=0; i<sb.length(); i++)
                    if (sb.charAt(i) == '-')
                        gapLocations[n++] = i;
            assert n == nGapsAfter;

            // Compute score.
            UngappedSequenceDataset ungapped = src.removeGaps();
            evaluate(ungapped, sbs[0].length());
        }

        else
        {
            // Try to slenderize. If slenderizer can't find a solution,
            // revert to random.
            int nColsBefore = src.widthOfWidestSequence();
            deltaWidth = Math.abs(deltaWidth);
            int nColsAfter = nColsBefore - deltaWidth;
            assert nColsAfter > 0;
            Slenderizer slen = new Slenderizer(src.values());
        this.gapLocations = new int[nGapsAfter];
        this.maxGapIndex = nColsAfter - 1;
        if (slen.slenderize(nColsAfter, nGapsAfter) != null)
        {
            // Slenderizer found a solution.
            char[][] reducedAlignment = slen.getAlignment();
            int n = 0;
            for (char[] seq: reducedAlignment)
                for (int col=0; col<seq.length; col++)
                    if (seq[col] == '-')
                        gapLocations[n++] = col;
            assert n == nGapsAfter;
            UngappedSequenceDataset ungapped = src.removeGaps();
            evaluate(ungapped, nColsAfter);
        }
        else
        {
            // Slenderizer couldn't find a solution. Revert to random.
            randomize();
            UngappedSequenceDataset ungapped = src.removeGaps();
            evaluate(ungapped, nColsAfter);
        }
    }
}


private void adjustSBs(StringBuilder[] sbs, int deltaWidth)
{
    if (deltaWidth == 0)
        return;
    else if (deltaWidth > 0)
        bookendNGapCols(sbs, deltaWidth);
    else
        deleteNGapCols(sbs, deltaWidth);
}


private void bookendNGapCols(StringBuilder[] sbs, int deltaWidth)
{
    boolean atEnd = true;

    while (deltaWidth-- > 0)
    {
        if (atEnd)
        {
            for (StringBuilder sb: sbs)
                sb.append('-');
        }
        else
        {
            for (StringBuilder sb: sbs)
                sb.insert(0, '-');
        }
        atEnd = !atEnd;
    }
}
```

```
    // Delete columns with most gaps.
    private void deleteNGapCols(StringBuilder[] sbs, int deltaWidth)
    {
        // Collect column #s by gap count.
        Map<Integer, Vector<Integer>> gapCountToColNums =
            new TreeMap<Integer, Vector<Integer>>();
        for (int colNum=0; colNum<sbs[0].length(); colNum++)
        {
            int gapCount = nGapsInCol(sbs, colNum);
            Vector<Integer> colsForGapCount = gapCountToColNums.get(gapCount);
            if (colsForGapCount == null)
            {
                colsForGapCount = new Vector<Integer>();
                gapCountToColNums.put(gapCount, colsForGapCount);
            }
            colsForGapCount.add(colNum);
        }

        // Collect columns to be deleted. When the time comes, these need to
        // be deleted in descending order, so as to preserve integrity of col #s.
        Set<Integer> deleteUs = new TreeSet<Integer>();
        Vector<Integer> sortedGapCounts = new
Vector<Integer>(gapCountToColNums.keySet());
        outer: for (int i=sortedGapCounts.size()-1; i>=0; i--)
        {
            Integer gapCount = sortedGapCounts.get(i);
            Vector<Integer> colsMatchingGapCount = gapCountToColNums.get(gapCount);
            for (Integer col: colsMatchingGapCount)
            {
                deleteUs.add(col);
                if (deleteUs.size() == Math.abs(deltaWidth))
                    break outer;
            }
        }
        Vector<Integer> vec = new Vector<Integer>(deleteUs);

        // Delete columns in reverse order of appearance in vec.
        for (int i=vec.size()-1; i>=0; i--)
        {
            int col = vec.get(i);
            for (StringBuilder sb: sbs)
                sb.deleteCharAt(col);
        }
    }


    private int nGapsInCol(StringBuilder[] sbs, int col)
    {
        int n = 0;
        for (StringBuilder sb: sbs)
            if (sb.charAt(col) == '-')
                n++;
        return n;
    }


    public static void main(String[] args)
    {
        try
        {
            File dirf = MSGFrame.GAG_POL_DIRF;
            File file = new File(dirf, "gag_04.clw");
            SequenceDataset gag4 = ClustalParser.parseFileToGapped(file);
            SizeAdjustingChromosome saChro = new SizeAdjustingChromosome(gag4, -2, 8);
            sop(saChro);
        }
        catch (IOException x)
        {
            sop("IO Stress: " + x.getMessage());
            x.printStackTrace(System.out);
        }
    }
}
```

```
package msg;

import java.util.*;
import java.io.*;


/*
 * Here's the puzzle: how do you reduce the width of a gapful alignment by
 * n columns, while removing exactly g gaps? The solution here is probabilistic.
 * Each pass randomly removes n columns; the pass is feasible if the n columns
 * contained g gaps. Feasible solutions with higher scores are preferred.
 */

class Slenderizer
{
    private final static int        N_PASSES = 10000;

    private Collection<String>      originalSeqs;
    private int                     nColsBefore;
    private int                     nColsAfter;
    private int                     nGapsBefore;
    private int                     nGapsAfter;
    private int                     nColsToRemove;
    private int                     nGapsToRemove;
    private int                     scoreOfBestSolution;
    private int[]                   gapsByCol;
    private char[][]                slenderAlignment;

    Slenderizer(Collection<String> originalSeqs)
    {
        this.originalSeqs = originalSeqs;
    }


    // Returns null if no solution can be found. Otherwise returns best solution
    // found by randomization.
    int[] slenderize(int nColsAfter, int nGapsAfter)
    {
        this.nColsAfter = nColsAfter;
        this.nGapsAfter = nGapsAfter;

        // Compute # cols & gaps to remove.
        nColsBefore = originalSeqs.iterator().next().length();
        nColsToRemove = nColsBefore - nColsAfter;
        assert nColsToRemove > 0 :
               "nColsBefore=" + nColsBefore + ", nColsAfter=" + nColsAfter;
        nGapsBefore = 0;
        for (String seq: originalSeqs)
            for (int i=0; i<seq.length(); i++)
                if (seq.charAt(i) == '-')
                    nGapsBefore++;
        nGapsToRemove = nGapsBefore - nGapsAfter;
        assert nGapsToRemove > 0;

        // Count gaps per col.
        gapsByCol = new int[nColsBefore];
        for (int col=0; col<gapsByCol.length; col++)
            for (String seq: originalSeqs)
            if (seq.charAt(col) == '-')
                gapsByCol[col]++;

    // Randomly generate columns.
    int[] bestSolution = null;
    int scoreOfBestSolution = Integer.MIN_VALUE;
    for (int i=0; i<N_PASSES; i++)
    {
        int[] candidate = buildRandomCandidate();
        if (!isFeasible(candidate))
            continue;
        char[][] charArr = delColsToAlignment(candidate);
        int candidateScore = AlignmentScorer.scoreAlignment(charArr);
        if (candidateScore > scoreOfBestSolution)
        {
            scoreOfBestSolution = candidateScore;
            bestSolution = candidate;
            slenderAlignment = charArr;
        }
    }

    return bestSolution;
}


private int[] buildRandomCandidate()
{
    int[] ret = new int[nColsToRemove];
    ArrayList unusedCols = new ArrayList<Integer>(nColsBefore);
    for (int i=0; i<nColsBefore; i++)
        unusedCols.add(i);
    int n = 0;
    while (n < ret.length)
    {
        int colIndex =  (int)(unusedCols.size() * Math.random());
        ret[n++] = (Integer)unusedCols.remove(colIndex);
    }
    return ret;
}


// A candidate is feasible if the columns it describes contain a total
// of nGapsToRemove gaps.
private boolean isFeasible(int[] candidate)
{
    int nGaps = 0;
    for (int col: candidate)
        nGaps += gapsByCol[col];
    return nGaps == nGapsToRemove;
}


private char[][] delColsToAlignment(int[] candidate)
{
    // Collect delete columns into a set for easier membership check.
    Set<Integer> deleteCols = new HashSet<Integer>(candidate.length);
    for (int col: candidate)
        deleteCols.add(col);
        // Retain chars from non-deleted cols.
        char[][] ret = new char[originalSeqs.size()][nColsAfter];
        int seqNum = 0;
        for (String seq: originalSeqs)
        {
            int destCol = 0;
            for (int srcCol=0; srcCol<seq.length(); srcCol++)
            {
                if (deleteCols.contains(srcCol))
                    continue;
                ret[seqNum][destCol] = seq.charAt(srcCol);
                destCol++;
            }
            assert destCol == nColsAfter;
            seqNum++;
        }

        return ret;
    }


    char[][] getAlignment()         { return slenderAlignment;    }
    int getWinnerScore()            { return scoreOfBestSolution; }
    static void sop(Object x)       { System.out.println(x);      }


    public static void main(String[] args)
    {
        try
        {
            File dirf = MSGFrame.GAG_POL_DIRF;
            File file = new File(dirf, "gag_04.clw");
            SequenceDataset gag4 = ClustalParser.parseFileToGapped(file);
            Slenderizer that = new Slenderizer(gag4.values());
            int[] winner = that.slenderize(47, 8);
            if (winner == null)
                sop("No solution found.");
            else
            {
                String s = "Delete these columns: ";
                for (int col: winner)
                    s += col + " ";
                sop(s);
            }
        }
        catch (IOException x)
        {
            sop("IO Stress: " + x.getMessage());
            x.printStackTrace(System.out);
        }
    }
}
```

```
package msg;

import java.util.*;


class UngappedSequenceDataset extends SequenceDataset
{
    UngappedSequenceDataset(String name)        { super(name); }


    public String put(String k, String v)
    {
        assert v.indexOf('-') < 0;
        return super.put(k, v);
    }


    int nCharsOverall()
    {
        int n = 0;
        for (String s: values())
            n += s.length();
        return n;
    }


    int[] getSequenceLengths()
    {
        int[] ret = new int[size()];
        int n = 0;
        for (String seq: values())
            ret[n++] = seq.length();
        return ret;
    }
}
```