Developing Algorithms for Computational Comparative Diachronic Historical Linguistics

Dan Wright, Computer Systems Lab Research Project, 2005

May 24, 2005

Abstract

The purpose of my research was to design algorithms and techniques which would aid in using computers to deal with languages, their relationships to each other, and their changes over time. My method of research is mostly thinking of ideas as to how to organize linguistic data or how to use it to build information about the subject, for whatever purpose anybody who might use my algorithms might have. My results are several algorithms, data storage methods, and general insights reached about the difficulties involved in dealing with historical linguistics on an algorithmic basis. My algorithm development culminated in a complete and functional algorithm for separating out related languages, which had some interesting results regarding different aspects of phonemes.

1 Introduction

Historical Linguistics is a relatively new study. It only achieved an even slightly scientific status in the 19th century, and its methods are typically unsystematic, and often reliant on intuition. If it is to be used with any great degree of accuracy and reliability, a systematic approach must be taken to historical linguistics analysis, and the best way to do this is to develop algorithms for it. The development of algorithms not only allows computers to do much of the rote labor which historical linguistic analysis has much of, but also leads to a greater understanding of the methods used.

I am aiming to develop algorithms for diachronic historical linguistics, meaning that it will reconstruct the changes that have occurred in languages at various points in time. This has no material benefit to anybody, but perhaps leads us to a better understanding of history, literature, culture, and to some degree language itself. I cannot predict what the algorithms I am attempting to create might lead to, but my purpose is simply to allow computational analysis of language change.

My specific and imminent material aim is to create a program that will, given a group of languages in some form reflected nothing beyond their phonetic data (with semantic connections to organize), form a chronological and familial hierarchy among them, discovering which grew out of which, and their relations to each other. Ideally, this would also develop hypothetical ancestor languages for each of these, placing these within the temporal hierarchy. Once the hierarchy is in place, the ancestor languages can be honed to be more and more likely and rigorously derived from their descendents, hopefully without modification of the hierarchy already created. I plan to program in C, without any object-oriented or input/output extravagancies.

2 Background Information and Theory

2.1 Phoneme Representation

Creating a phonetic representation system does not require a great deal of foresight as to use, and does not affect the algorithms that use it much by its design, other than in the realm of efficiency. A few integers will describe any phoneme, and there are many phoneme description schemata to choose from. I personally chose the most standard, that of the International Phonetic Alphabet. Once one has categorized and schematized phonemes, they can be arrayed into words. See Appendix A for a description of the phonetic categorization system I used.

2.2 Language Representation

Once one can store words, one can define a language. Morris Swadesh developed a procedure of generating lists of words that will not be borrowed, and will remain in a language only changed by phonetic phenomena. These are simple words, used frequently in daily life. If one stores a phonetic description of a Swadesh List of words, this defines a language at one point in time. This snapshot of a language can be used to directly relate any language to any other on purely phonetic grounds, escaping the traps inherent in keeping any semantic basis. This also avoids using any connection to grammar, which is a far more complicated subject and does not follow direct phonemic changes, unaffected by other things.

2.3 Language over Time

If you can define a language at one point in time, you can define various languages at various points in time and link them, to create a fully temporal as well as phonetically spatial language. It is feasible to represent dialect in this manner, and make the language geographically spatial as well, but this is outside of my intention.

It has been shown, beginning with people such as Jacob Grimm in the 19th century, that languages, on some level, change by regular phonetic rules. These rules are unaffected by semantics or other languages, and function randomly. Their randomness is probably not pure, but the multitude of factors affecting phonemic change are so complex that the result appears random, and can be treated as such in analysis.

Most approaches to this have represented the various states of the language in a tree. This does a good job of showing which languages have relationships to each other, but does nothing to represent the nature of the relationships themselves. My goal is to discover the actual sound changes which occur between language states from the raw data, and using this information better determine unknown states of the language.

2.4 Problems and Simplifications

There are various problems which make determination of sound changes difficult, and simplifications which remove these problems, though also lowering the accuracy of the conclusions.

2.4.1 Borrowing

I aim to analyze the regular phonetic changes between states of a language. However, not all language change occurs because of regular phonetic changes. Therefore, I must only use word that are not subject to borrowing, and only subject to regular phonetic change. There are lists of words in a language known as Swadesh lists which are never borrowed, due to their fundamental and common use. In only using these words, I can avoid the problem of borrowing, with no significant loss of accuracy in conclusions drawn.

2.4.2 Polymorphism

In many languages, there are various phonetic forms of the same semantic form, i.e. synonyms. With these, typically only one will be phonetically related to those in previous languages. When assembling the list of words to use, it then becomes necessary to choose the word form cognate with the other word forms I am using. I can see no way to get around manually checking every list of words for non-cognate synonyms. This can be circumvented, but at a cost in speed.

2.4.3 Dependent Evolution

The base assumption of my model of regular phonetic change is that sound changes are regular and independent. A problem is that occasionally sound changes, by changing the allocation of mouth space for various phonemes, can cause other sound changes. Initially, I will simply assume all sound changes are random. I may work in frequently-occuring dependencies in making my

algorithms more efficient. If possible, I will use dependency in determining the likelihood of various sound changes, and therefore most likely past language states.

2.4.4 Homoplasy

When a phoneme undergoes a sound change, in some cases it will be changed into a phoneme that already exists. E.g., /b/ is devoiced to /p/, but the /p/ phoneme already exists and is unchanged. I do not plan to eliminate homoplasy, and will definitely keep it as a possibility when analyzing sound changes from raw data. I will also attempt to predict homoplasy when formulating past states.

3 Design Criteria

I approached the problem of designing the complete system from both ends.

My first work involved finding ways to categorize phonemes, and store them, then work up from this to storage of words in Swadesh lists. This would be required for all work dealing with languages, whether over time or simply synchronic.

My nextt attempts were to develop algorithms to, given phonetic data of languages, find the connections between them, and the regular sound changes. This was successful in one way, but led to a dead end in another. I was able to develop algorithms to find the connections, but I found these connections useless. Sound changes do not follow linearly in the paths of those preceding them, and there is no such things as phonetic momentum. Simple similarity judgments proved more useful than sound changes. However, these algorithms may serve a purpose in some other aspect of historical linguistics, so my working on them was not a complete waste, they simply ended up being something of a tangent to my main effort.

My third, and most actually significant effort, were my attempts to work from the top down. I would simply, given languages, form a hierarchical web. I began by simply organizing them by similarity, and discovered that organizational processes, if handled correctly, would actually serve to find hypothetical ancestor languages.

3.1 Phoneme Storage

I designed a phoneme storage system described in Appendix A. To store a single phoneme, a 5-dimensional array of integers is needed. I basically used the guidelines of the International Phonetic Alphabet, but found a way to not separate vowels from consonants. I did not include clicks or other weird sounds, but my method of storage could easily be adapted to them. Essentially, there is a series of arrays within arrays describing the various dimensions through

which phonemes can be classified. However, each level means a different thing depending on which section of the array you are in. In the consonants, the second dimension is voice, but in vowels this is not needed, so rounding is used there. This provides for a maximum efficiency of space, and an ease of access, balancing well memory and processing usage. This phoneme storage system can be applied to anything.

Word storage involves simply making an array, or linked list, or some other linear structure of phonemes. A Swadesh list can simply be an array of words in which each position in the array corresponds to a single meaning.

The phonetic categorization and storage system was my most concrete achievement.

3.2 Correspondence

3.2.1 Brute Correspondence Algorithm

This algorithm, the only one I have fully implemented, is very simple and ineffecient, but is a beginning. I go through two word-lists representing states of a language and find every case in which one phoneme in one corresponds to another in the other. When a correspondence occurs in all cases, it is recorded. These correspondences are the most simply sound changes, and can be used to analyze other sound changes. This algorithm is a necessary first step to analyzing the differences between two languages, though horribly inefficient. It is unpleasant, but cannot be avoided.

3.2.2 Limited Correspondence Algorithm

This is a refinement of the Brute Correspondence Algorithm. In this, I first limit my list of phonemes to go through. I can do it either for those that occur in the first list, in both lists, or to some pre-existent list of phonemes for the language. This is somewhat dangerous, as it brings about a possibility of missing really peculiar changes, but it is much, much faster than just going through everything.

3.2.3 Abstract from Correspondences Algorithm

In this, I go through the sound changes discovered by either of the above algorithms, and by a brute search find regularities, constant shifts between phoneme types. Again, this is brute force, but refinements can be added later to make it more efficient.

3.3 Web Formation

3.3.1 Web Formation Algorithm

The Web Formation Algorithm will act on a "Family." The first Family will be formed of all of the languages and a proto-language formed from all of them via

an Ancestration Function. A Family has a parent, the proto-language, and any number of children, which are either languages or Families. At the beginning of the WFA, the Family has no proto-language, and the WFA forms it through an Ancestration Algorithm. The Web Formation Algorithm will take a random language, and find all of the languages which it is closer to (by some Distance Function, which I will discuss later) than it is to the proto-language. These are put into their own family, which is made a subfamily of the original family. This is done until either every language is closer to the proto-language than anything else, or all language are in Families. Then, the Web Formation Algorithm is applied to every Family within this Family. Eventually, there will be a web of connected families, culminating in leaf families whose only member is a single language.

The code for my full web formation program (with equal aspect weighting) can be found in Appendix C.

3.3.2 Ancestration Algorithm

This creates a proto-language from all of the languages which descend from it. For the abstract number languages, I've simply used averaging. This would be where the phonetic data comes in, and this would be one of the only places where abstract number-languages differ from actual phonetic data. It is hypothetically possible to only need a distance function and not a separate ancestraction function by having the ancestration function act by creating random languages and slowly working them closer in distance to the original languages, but this would be monstrously inefficient and could be greatly helped by use of phonetic knowledge.

3.3.3 Distance Function

The Distance Function is used within the WFA. It is what makes use of the information built up through the Correspondence Algorithms. Simply by changing the Distance Function, the Web Formation process I describe here can be applied to many things. For testing, I mostly used abstract number-lists, and applied it to languages at the end.

4 Procedure

I ran several tests of my algorithms on various words in various languages. The exact results of my test can be found in Appendix B. I did two tests on random sets of words, when I was unable to find a reliable and complete source of phonetic data. Once my planned source was available, I did several tests on word banks in six languages: English, German, Portuguese, Dutch, Romanian, Danish, Scots Gaelic, Polish. Polish and Scots were great outliers in many of the cases, so I removed Polish in all but one and Scots in one. They were difficult

to phonetically map, and mapping in general showed itself to perhaps be the greatest difficulty in my endeavor.

I ran my web-formation algorithms with two different distance functions, one in which all phoneme aspects were weighted equally and one with stronger weighting on more important aspects of a phoneme. For each test, I ran a single iteration on the group of words, and recorded which languages were separated out from each other for each word. This kept it simple, so problems could be easily diagnosed. The complete results of my testing can be found in Appendix B. The "Word Bank" words are reliable enough data for analysis.

Each word was converted into a series of integers using my phonetic classification system (see Appendix A). Then, the web-formation algorithm was run on the lists of integers.

5 Discussion

I did not succeed in unifying all of my efforts, due to the overwhelmingly large scope of historical linguistic issues, but I did make significant progress in from both ends of the general problem. My phonetic classification system can be applied, and something like it will be necessary, in any effort actually using linguistic data in any way. Simply put, traditional alphabetic systems will not function, and an organized multi-dimensional method is far more efficient, in space and time, than a linear storage. My correspondence algorithms were generally uninteresting and on the whole not very useful. These were certainly the weak point of my work this year. My web formation algorithms were probably the strong point this year. I examined how to deal with languages (through family relations) and developed methods that will, with further application, yield actual information about language change. By experimenting with various distance algorithms within my framework, and testing the accuracy of these with regard to actual historical data gleaned through writings and extensive manual language derivation that has been done in the past, one could see how these models would correspond to the actuality of language change. My web-formation algorithms are a useful framework for historical linguistic experimentation and theorization, as the field has in the past been limited to purely historical studies.

6 Results

With the equally-weighted distance function, approximately half of the tests were failures, not separating out any languages whatsoever. There was one complete absolute success, in which the Germanic languages for "Horn" were separated from the rest. With Sun, there was a general success, but not perfection.

The adjusted weight distance function always separated out the extreme

outliers, and had one perfect separation (the Germanic languages for "come") and one near-perfect separation (Portuguese and Romanian for "liver").

Neither ever gave any grossly wrong answers, irreparably separating closely grouped languages. Both weighting systems were equally effective. The Germanic languages were far more coherent than the Romance, and the outliers never cohered to each other.

7 Conclusion

My web-formation algorithm provides a good, somewhat reliable and effective tool to separate out related languages from the unrelated. Applying several distance functions with different weighting systems, and combining the results of all of these results in increasing degrees of correctness. Further iterations are as successful as earlier. My web-formation algorithms, using my phoneme classification system, can separate languages given words. By combining more and more results, on different words and with different weights, an extremely accurate portrait of a language-web can be formed. An interesting result I discovered is that language change does not regularly affect any aspect of a phoneme more than any other.

8 References

Sedgewick, Robert. Algorithms in C, Part 5: Graph Algorithms. 2002. Addison-Wesley. Boston, Massachusetts.

Kanna, S., Warnow, T. A Fast Algorithm for the Computation and Dnumeration of Perfect Phylogenies. 1996.

Warnow, T., Nakhleh, L., Ringe, D., Evans, S. A Comparison of Phylogenetic Reconstruction Methods on an IE Dataset.

Warnow, T., Nakhleh, L., Ringe, D., Evans, S. Stochastic Models of Language Evolution and an Application to the Indo-European Family of Languages.

9 Appendix A

My phoneme categorization system uses five integers to define a phoneme. Each phoneme is stored as an array of five integers, but could easily be converted into a more compact binary format, due to the limits on each integer. The first integer can only be within a range of 0 to 1, the second is also 0 to 1, the third is 0 to 11, the fourth is 0 to 7, and the fifth is once more 0 to 1, only used for vowels.

The first integer declares whether the phoneme is a consonant or a vowel. A value of 0 is a consonant, a value of 1 is a vowel.

The four other integers are differently used for consonants and vowels. The consonant system uses only three of the integers, and the vowel system has smaller ranges for the four integers it uses.

The consonant system begins with a 0 or 1 for voice. If 1, the phoneme is voiced. The third integer defines Place of Articulation. 0 is bilabial, 1 labio-dental, 2 spans dental, alveolar and postalveolar in situations where they are undifferentiated, 3 is dental, 4 alveolar, 5 postalveolar, 6 retroflex, 7 palatal, 8 velar, 9 uvular, 10 pharyngeal, and 11 glottal. The fourth integer defines Method of Articulation. 0 is plosive, 1 nasal, 2 trill, 3 tap (or flap), 4 fricative, 5 lateral fricative, 6 approximant, 7 lateral approximant.

The vowel system also begins with a 0 or 1, but for roundedness rather than voice. The second integer defines openness of the vowel: 0 is close, 1 close-mid, 2 open-mid, and 3 open. The third integer defines Frontness: 0 is back, 1 central, 2 front. The fourth integer defines offset, which is for phonemes very close to another phoneme and differentiated in various directions, but not so different in any so as to be classified differently. The phoneme which less completely fits the phonemic classification is offset.

10 Appendix B

Testing Data:

Fly

Languages: English, German, Romanian, Latin, French, Greek Fly, fliegen, bura, volar, voler, peto,

Separated out Greek, German, and Romanian from English, Latin, and French The l was present in the separated languages, which seemed the key factor. The gapping of the lack of vowel in English and German also seemed key. With English and German consonant-expanded rather than vowel-expanded, Greek and Romanian only were separated out. This is probably better.

Hand

Languages: English, German, Italian, French, Greek, Dutch Hand, Hand, mano, mano, cheri, hand Did not separate at all.

Word Bank Languages: English, German,Portuguese, Dutch, Romanian, Danish, Scots Gaelic, Polish. Polish was usually not phonetically mappable to other languages, so it was removed in almost all cases.

Words: Horn, Mouth, Liver, Come, Sun, Moon

Horn

Equal Weight: English, German, Dutch and Danish were kept in, and the rest were separated out. Worked perfectly.

Adjusted Weight: Separated out Polish alone.

Mouth

Equal Weight: All of the languages were separated out as aberrant and unrelated. A failure.

Adjusted Weight: Separated out English alone.

Liver

Equal Weight: All of the languages were separated out as aberrant and unrelated. A failure.

Adjusted Weight: Separated out Portuguese and Romanian. Very effective, very correct.

Come

Equal Weight: Separated out the Scots from the rest, which though not terribly informative is correct.

Adjusted Weight: Separated out English, German, Dutch, and Danish. Absolute perfection.

Sun

Equal Weight: Separated out Portuguese, Romanian, and Danish. Once again, not perfect, but of good quality.

Adjusted Weight: Separated out Scots and Dutch. Imperfect, completely acceptable quality.

Moon

Equal Weight: Separated out Scots, and separated out Portuguese with Scots removed. Scots was removed because it was an intense outlier in this case, not at all phonetically mappable.

Adjusted Weight: Separated out Romanian with Scots removed. Not especially informative.

11 Appendix C: Web Formation Code

```
#include<stdio.h>
#include<malloc.h>

void printlist(int* list);
void familyoutput(int** family);
int** newfamblank();
int** newfam(int* firstlist);
int** addtofam(int** family, int* list);
int** addfamtofam(int** family, int** newFam);
int* ancestor(int* lista, int* listb, int weighta, int weightb);

int** familiate(int** family)
{
int n = family[0][0];
if(n < 2)
return family;
int x = 0;
int y = 0;
```

```
int newfirst = 0;
int newfamcount = 0;
int* toswitch = (int*)malloc(sizeof(int) * (n+1));
for(x = 0; x <= n; x++)
toswitch[x] = 0;
for(x = 2; x <= n; x++)
if(family[0][x] == 0)
newfirst = x;
if(newfirst == 0)
return family;
//printf("Length: %d, %d\n", family[0][0], n);
//printf("Newfirst: %d\n", newfirst);
//familyoutput(family);

int* newanc = (int*)malloc(sizeof(int) * 11);
newanc[0] = 11;
for(x = 1; x <= n; x++)
{
if(family[0][x] == 0)
newanc = family[x];
}

int newedin = 0;
for(x = 1; x <= n; x++)
{
//printf("'Bout to ancestrate, %d\n", family[0][x]);
if(family[0][x] == 0)
{
newanc = ancestor(newanc,family[x],newedin,1);
newedin++;
}
//printf("Ancestrated.\n");
}
int newnewfirst = newfirst;
for(x = 1; x <= n; x++)
if(family[0][x] == 0){
if(distance(newanc,family[x])
>= distance(family[x],family[newfirst])){
toswitch[x] = 1;
newnewfirst = x;
newfamcount++;
//printf("%d is within distance.\n", x);
}
else { //printf("%d is not within distance.\n", x);
```

```
//printf("Distance from anc: %d\n", distance(newanc,family[x]));
//printf("Distance from new: %d\n",
//distance(family[x],family[newfirst]));
}
}


int** newFam = newfamblank();
//printf("The first.\n");
//familyoutput(newFam);
//Create new subfamily if there is one, put it in
if(newfamcount > 0){
int** subFam = newfam(family[newnewfirst]);
for(x = 1; x <= n; x++)
if(toswitch[x] == 1  && x != newnewfirst)
subFam = addtofam(subFam,family[x]);
//printf("The subfam.\n");
//familyoutput(subFam);
newFam = addfamtofam(newFam,subFam);
//familyoutput(newFam);
}

//Add in unswitched lists
for(x = 1; x <= n; x++)
{
if(family[0][x] == 0){
if(toswitch[x] == 0){
newFam = addtofam(newFam,family[x]);
//printf("Added a list.\n");
}
}
//else
//newFam = addfamtofam(newFam,(int**)family[x]);
}
//familyoutput(newFam);
return newFam;
}

int** newfamblank()
{
int** toRet = (int**)malloc(sizeof(int*) * 1);
toRet[0] = (int*)malloc(sizeof(int) * 1);
toRet[0][0] = 0;
return toRet;
```

```
}

int** newfam(int* firstlist)
{
int** toRet = (int**)malloc(sizeof(int*) * 2);
toRet[0] = (int*)malloc(sizeof(int) * 2);
toRet[0][0] = 1;
toRet[0][1] = 0;
toRet[1] = firstlist;
return toRet;
}

int** addtofam(int** family, int* list)
{
int curLen = family[0][0];
int** newFam = (int**)malloc(sizeof(int*) * (curLen + 2));
int* newIndex = (int*)malloc(sizeof(int) * (curLen + 2));
newIndex[0] = curLen + 1;
int x;
for(x = 1; x <= curLen; x++)
{
newIndex[x] = family[0][x];
newFam[x] = family[x];
}
newFam[curLen + 1] = list;
newIndex[curLen + 1] = 0;
newFam[0] = newIndex;
return newFam;
}

int** addfamtofam(int** family, int** newFam)
{
int curLen = family[0][0];
int** toRet = (int**)malloc(sizeof(int*) * (curLen + 2));
int* newIndex = (int*)malloc(sizeof(int) * (curLen + 2));
newIndex[0] = curLen + 1;
int x;
for(x = 1; x <= curLen; x++)
{
newIndex[x] = family[0][x];
toRet[x] = family[x];
}
toRet[curLen + 1] = (int*)newFam;
newIndex[curLen + 1] = 1;
```

```
toRet[0] = newIndex;
return toRet;
}

//A simple family will consist entirely of languages.
void familyoutput(int** family)
{
//printf("I've been called to output!\n");
printf("<<<<<<<<\n");
int len = family[0][0];
printf("%d\n", len);
int x = 0;
for(x = 1; x <= len; x++)
{
if(family[0][x] == 1){
//printf("Subfamily:\n");
familyoutput((int**)family[x]);
}
else
printlist(family[x]);
}
printf(">>>>>>>>\n");
}

int* inlist(FILE* inFile)
{
int newlen, x;
fscanf(inFile, "%d", &newlen);
//printf("Firstlen: %d\n", newlen);
int* toRet = (int*)malloc(sizeof(int) * (newlen + 1));
for(x = 0; x < newlen; x++){
fscanf(inFile, "%d", &toRet[x+1]);
//printf("%d\n", toRet[x+1]);
}
toRet[0] = newlen;
return toRet;
}

void printlist(int* list)
{
int x = 0;
printf("%d: ", list[0]);
for(x = 1; x <= list[0]; x++)
printf("%d ", list[x]);
```

```
printf("\n");
}

int distance(int* lista, int* listb)
{
//printf("About to Distance!\n");
int len = lista[0];
//printf("List A doesn't explode!\n");
if(listb[0] < len)
len = listb[0];
//printf("List B doesn't explode!\n");
int x = 0;
int dif = 0;
double avg = 0;
for(x = 1; x <= len; x++)
{
dif = listb[x] - lista[x];
if(dif < 0)
dif *= -1;
avg += (double)dif/len;
}
//printf("Distanced!\n");
return (int)avg;
}

int* ancestor(int* lista, int* listb, int weighta, int weightb)
{
int lenn = lista[0];
if(listb[0] > lenn)
lenn = listb[0];
//printf("Len: %d\n", lenn);
int* listn = (int*)malloc(sizeof(int) * (lenn + 1));

int x = 0;
int newval = 0, tempa = 0, tempb = 0;
for(x = 1; x <= lenn; x++)
{
newval = 0;
tempa = weighta;
tempb = weightb;
if(lista[0] >= x)
newval += lista[x] * tempa;
else
tempa = 0;
```

```
if(listb[0] >= x)
newval += listb[x] * tempb;
else
tempb = 0;
newval = newval/(tempa + tempb);
listn[x] = newval;
//printf("Newval %d: %d\n", x, newval);
}
listn[0] = lenn;
return listn;
}

int** newinput(FILE* inFile)
{
int newlen, x;
fscanf(inFile, "%d", &newlen);
int** toRet = (int**)malloc(sizeof(int*) * (newlen + 1));
toRet[0] = (int*)malloc(sizeof(int) * (newlen + 1));
for(x = 1; x <= newlen; x++){
toRet[x] = inlist(inFile);
toRet[0][x] = 0;
}
toRet[0][0] = newlen;
return toRet;
}

int main()
{
FILE* inFile = fopen("temp.web", "r");
//FILE* inFileA = fopen("a.web", "r");
//FILE* inFileB = fopen("test.web", "r");
//int* lista = inlist(inFileA);
//int* listb = inlist(inFileB);
//printf("A:\n");
//printlist(lista);
//printf("B:\n");
//printlist(listb);
//int* listc = ancestor(lista, listb, 1, 1);
//printf("C:\n");
//printlist(listc);
//printf("Distance A-B: %d\n", distance(lista,listb));
//printf("Distance A-C: %d\n", distance(lista,listc));
//printf("Distance B-C: %d\n", distance(listb,listc));
//Dealing with families now.
```

```
int x;
/*int** family = (int**)malloc(sizeof(int*) * 4);
family[0] = (int*)malloc(sizeof(int) * 4);
family[0][0] = 3;
for(x = 1; x <= family[0][0]; x++)
family[0][x] = 0;
family[1] = listc;
family[2] = lista;
family[3] = listb;
familyoutput(family);
familiate(family);
//familyoutput(family);*/
//int** family = newfam(listc);
//family = addtofam(family,lista);
//family = addtofam(family,listb);
//printf("About to familiate.\n");
int** family = newinput(inFile);
family = familiate(family);

//family = familiate(family);
//printf("OHMYGODITWORKS\n");
familyoutput(family);
return 0;
}
```