Devan Bicher

Project 1: Implementation Report

3/2/13

For my implementation project I chose to write and implement my algorithm in the language python.

I started out simply but reading in the file that contained all of the reads in it in fasta format, this file is supplied by the user when the program is initially run. The first thing that the program then does is it reads in the file, it does this by reading in lines in the file and storing those lines in an array. This array is then passed to another method which parses the fasta file and puts into a list that is readable for my project. It reads in the header, which is indicated by a ‘<’ and then reads in all of the text after that as my read. After putting all of the reads into a list my program then moves on the next step: creating the hash table of all of my prefixes and suffixes.

Before the hash table can be created I have to convert the prefix and suffix for each read into the hash code that will serve as the index for each node. So I split each read into its appropriate 15 nucleotide sequence and convert it to code with the equation given in class (n0\*4^0 + n1\*4^1 + … n14\*4^14) where A is 0, C is 1, G is 2, and T is 3. Once the hash codes are created I store them into a dictionary, which I called outerSet, by first placing the prefix of the read into the first level of the outerSet table, or dictionary as it is called in python. If the prefix already exists I store the suffix in the dictionary for that prefix, if not I create a new entry in the dictionary for that prefix. If the suffix also exists for that prefix I simply place the read into the edge bag list for that prefix/suffix combination, if it does not exist than I create a new suffix entry under that prefix and store the read in the newly created edge bag. After doing this for a read I store or update the ‘inDegree’ and ‘outDegree’, which is stored in a separate table, for both the prefix and suffix of that read. This is done by either creating the entry for the prefix and adding 1 for the outDegree of that prefix, because clearly this prefix points to the corresponding suffix; if there is an entry for this prefix in the table I increment the outDegree up by one. The same process is done for the suffix of the same read only inDegree is updated instead of outDegree. Then this process of storing data in both tables is repeated for every single read in the fasta file.

Now that both tables are created I need to start generating my paths, or contigs, by walking along the edges, or reads, between prefixes and suffixes. I started at the first entry in my hash table and looked at my degree table to see if the outDegree for that node was greater than the in degree. From there I chose the first read for the first suffix for that prefix, I stored that read into a list of used reads and used that suffix as the next prefix and repeated that cycle until I reached a prefix for which all available suffixes were either not prefixes themselves or had no edges that were not already used. After reaching the end of that particular path I started again at that prefix that was the beginning of that path and if that same prefix still had an outDegree greater than inDegree than I continued that same cycle of finding a path out from that prefix. From there I continued along the list of prefixes, starting a path at each prefix that had outDegree greater than inDegree, along the way I de-incremented the indegree from each suffix I used to and the outDegree for each prefix that I used. I did this cycle until no more prefixes had outDegree greater than inDegree.

Once all of the cycles were completed the next step to take was to perform path compression in order to take the small paths, or contigs and combined them into a list of contigs for which each one is as long as possible. The first step that I took in doing this was to eliminate loops in the contigs. That is contigs which have the same prefix and suffix. This was a simple matter to program and if I found any loops I deleted them from the final list of contigs. After removing loops I moved on to check for the case where one contig was completely contained within another contig. If this was the case than the smaller contig, which was contained in the larger contig, was deleted. After I removed redundant contigs in this way I moved on the most complicated path compression which removed duplicate paths and which assembled overlapping regions, or frameshifts, into one longer contig.

In order to perform this check I looped over every entry in my contig list and for each contig I looped over the whole list again to compare each contig all of the others. The first check in my loop was for duplicate contigs; this was an easy check to see if they were equal, and if one was found it was deleted. Next is the loop I checked if the prefix of the first contig was in the second and simultaneously if the suffix of the second contig was in the first contig. From there I check every occurrence of the prefix of the second contig in the first contig, at each occurrence I compared the value one node away, or in the first case the second node, from the prefix to the node directly after the matching node in the second contig. If these two nodes matched I moved on to the node after that, the third node, and so on, fourth, fifth etc, until I either came to the suffix or until they did not match. If there was a mismatch then I simply continued on in the loop. If there was a match then the second contig was concatenated onto the end of the first contig at the point directly after the node matching the suffix of the first contig.

After this loop was performed for the list of all of the nodes I converted the entire list of contigs in node form into a list of the same contigs in letter form, or proper nucleotide sequence form. This algorithm was slightly complicated, in order to decode the node indexes I started at the last letter, which has to be a multiple of 4^14, or if not it is an A, this was the case when the remainder was the same as the value of the index; and if was an even multiple I set the letter equal to its corresponding multiple. Than after each iteration the new code became the old remainder until the remainder was 0 in which case I again set the letter corresponding to the multiple. Then finally if the remaining code was equal to zero I set every letter after that to A for the length of the remaining code.

Finally after converting to letters I performed the frameshift check again only this time on the letters. I reused the same frameshift checking algorithm as described above for this check. Then finally I printed out all of the remaining contigs. I did this because while some of the remaining contigs might be nonsense it is more likely that these are contigs separated by some random region of DNA that my algorithm had a hard time lining up, or this might come also from the randomization of the vast amount of DNA data. While in my testing I have not come across multiple contigs at the end of my run I do not want to exclude this possibility from the final output.

One of the weaknesses of my algorithm is that in each and every case I have cut off a few nucleotides at the very beginning and ends of my sample genome, no more than around a dozen from what I have found. This is a weakness that arises from the randomization of the data supplied to my algorithm. However this type of error will not have an effect on my algorithms performance in a real world setting because both the end and beginning of real genomes, in eukaryotic organisms, are capped with long repeating regions, so it will not matter if my algorithm cuts off a few nucleotides. Another major weakness of my algorithm is that its performance is rather slow. Despite my best efforts to speed it up I simply cannot make it move very quickly for the largest of the genomes that was supplied to us for testing our program, running my program took over an hour and that was only when walking the paths. This is certainly a draw back and I believe it is a complication of my choice to use python, which may not be as speedy as using something more fundamental like C++.

Some of the difficult regions to code for are cycles, both long and short repeating sequences. Cycles are difficult to find computationally because they are paths that begin and end at the same node and so it is hard to find these because the inDegree of this type of node equals the outDegree. So when searching it is possible that nodes in the path of a cycle cannot be reached by any other edge, and since in a cycle inDegree equals outDegree the check in my, or another implementation, of outDegree being greater then inDegree will not be met and so this cycle cannot be found. Another difficult region to identify is repeating regions, they are regions of short sequences of nucleotides repeated over and over again, they can either be very large or somewhat smaller. These regions are hard to decipher computationally because they are long repeats of the same sequence of DNA and so it presents a similar problem as cycles and it also presents the problem of walking over the same path repeatedly which is hard for a program to decide how many times the series of nucleotides should be repeated. In genetics these regions are useful for watching how genes both evolve and are different for different organisms, they are also useful for determining a person’s identity from their DNA, like in crime shows, and as genetic markers for studying genes, and the diseases and health concerns associated with them, throughout populations. Biologically these regions can represent alleles, for example a really long segment might make expression of the phenotype for the proteins it codes for stronger. These regions are also typically a source of mutation due to slippage by the proteins that latch on at or near these sites.

If I had technology with far more advanced computing power I would certainly strive to fix these errors. One way in which I would do this would be to run the same program on the same set of data in parallel. Only I would start my path walking at different nodes in the set of all nodes so that each parallel assembly will create different initial paths which through repetition will catch any errors that might arise from cycles or repeats. Ideally through using this type of parallel alignment we could also take a consensus at the end of the variable length repeating regions so that we could have a better shot at picking a sequence with the correct number of repeats. Another way in which greater technology could improve upon sequencing would be to go back, after all the contigs were created and check for unused edges, and for each unused edge create a path to search for cycles. Then the program would search through the sequence for a place where the same start/stop node would fit into the sequence, and therefore the cycle would also fit into the sequence. This would hopefully place the cycle in the correct location and so would effectively encompass the entirety of the reads submitted to the program. However depending on the amount of unused reads this could take a while and as the number of initial reads goes up so to would the number of unused reads at the end. In fact one of the drawbacks of my program is that I do not account for cycles. Finally to further increase the accuracy of my sequencer I would cross check the results from my final sequence or sequences with known data for the same species and organism through some reliable database on the internet. This would validate my data and confirm the accuracy of my assembler, and it would have the added advantage of running population analysis on the same sequence at the same time that it is checking the validity of the sequence. Since this is the point in today’s society, to check a person’s DNA versus that of the overall population for both personal and statistical analysis, it would be nice to have this integrated into the algorithm.

A DNA array is a large sample of small portions of DNA, which have been collected, purified and amplified from a living source, attached to a solid surface. This technology is widely used for DNA sequencing and other genetic based analyses. The point of this technology is that small “starter” portions of DNA are immobilized, by being strongly bound, to some spot on a surface of a small chip. Then DNA is washed over this chip so that the DNA from the host will bind to the immobilized portion of DNA already on the chip. This technology is both highly accurate and specific so that an exact amount of DNA, say 30 nucleotides like in our project, can be immobilized on a surface. Then through fluorescence the exact sequence of each segment of DNA can be rapidly deduced, that is through the measurement of the intensity of light given off by each of the nucleotides in the small sequence. These chips are organized in a grid-like setup with numerous spots for the DNA to bind so that this technology is not only high-throughput but fast and efficient as well (3). In fact modern systems like illumina use this approach to high-throughput sequencing.

Gene-expression profiling rather than trying to discover what types of genes are present in a particular cell or organism, which is the primary aim of gene sequencing, is instead interested in the activity of gene expression. More specifically it aims to uncover what types of genes are being expressed, which is different from what types of genes are contained within a genome, how much and how quickly they are expressing their genes and all of this information is analyzed for different cells in different organisms at different times and under different conditions. The point of this technology is to not simply say whether or not a person has a gene or certain type of gene but rather to indicate how a cell is using its genes, in what ways, at what times and under what conditions. This is also important knowledge to obtain because it can indicate the importance of such genes and can also shed light on what mechanisms and proteins the body uses under various conditions. Gene expression profiling can, in addition to DNA sequencing, can also take advantage of the high accuracy and speed of DNA arrays through the correlation between gene expression and RNA. Gene expression is ultimately carried out in the body by the various enzymes and proteins that DNA codes for, but since merely knowing the DNA sequence does not shed light on expression levels another source must be investigated. This source is RNA, RNA can be analyzed through the same methods as described above for DNA. RNA is a good indication of gene expression because RNA must be produced in order for proteins to be produced. So by taking a cell sample RNA can be purified out and analyzed using an array which will indicate the quantities and type of genes being expressed through the quantities and types of RNA that are found in the sample. (1)

Another application for DNA arrays is in SNP detection. A SNP is a single nucleotide polymorphism, which is just the substitution of one nucleotide for another in different organisms. A single substitution might not seem important in a gene consisting of thousands of base pairs, let alone a genome of billions. However a single substitution is not only the most common mutation in the genome it can be the difference between a positively charged amino acid and a negatively charged one, which in enzyme functioning is most crucial. Furthermore it is known that certain SNPs can lead to serious health problems, even including birth defects and genetic disorders, and they are also implicated in various forms and stages of cancer. SNPs are also important for studying cross-species correlations because in some cases it is a change in a single nucleotide that can be the cause for some major difference. With the importance of SNPs and the knowledge of DNA arrays it is easy to see how the speed and accuracy of the DNA array can process numerous pieces of DNA from various organisms to analyze SNPs. DNA arrays can be tailored with certain docking nucleotide sequences so that if we cut a whole bunch of DNA apart, with specific restriction enzymes and are looking at only a specific gene, for which we want to analyze the different SNPs present, we can plant a sequence on the array that is specific to the sequence we want to analyze and in this way we can perform a SNP detection analysis (2).

Works Cited

Gene Expression Profiling by DNA Microarray Technology. Keisuke Iida, Ichiro Nishimura. Critical Reviews in Oral Biology & Medicine January 2002 13: 35-50.

Genome-wide Copy Number Profiling on High-density Bacterial Artificial Chromosomes, Single-nucleotide Polymorphisms, and Oligonucleotide Microarrays: A Platform Comparison based on Statistical Power Analysis. Jayne Y. Hehir-Kwa, Michael Egmont-Petersen, Irene M. Janssen, Dominique Smeets, Ad Geurts van Kessel, Joris A. Veltman. DNA Res (2007) 14(1): 1-11 first published online March 15, 2007.

Light-generated oligonucleotide arrays for rapid DNA sequence analysis.

A C Pease, D Solas, E J Sullivan, M T Cronin, C P Holmes, S P Fodor

Proc Natl Acad Sci U S A. 1994 May 24; 91(11): 5022–5026