Chuong Vu

COMP 5800

Final Project

Prof. Byung Kim

FINAL PROJECT - VELVET PROGRAM

Velvet is an algorithm package that has been designed to deal with de novo genome assembly and short read sequencing alignments. This is achieved through the manipulation of de Bruijn graphs for genomic sequence assembly via the removal of errors and the simplification of repeated regions. Velvet has also been implemented inside of commercial packages, such as Sequencher, Geneious, MacVector and BioNumerics.

Overall structure of the program, the program is broken into two separate executive program (velveth and velvetg). Both processes used hash bucket structure to process the data. This would save a lot of time if the DNA analysis is failed in the middle of the process.

For the first part, velveth. It read the original DNA sequences from the file (in this report I used the read1.fa file) and analyze it to build the new modified DNA sequence file and the Roadmaps file based on the user input K-mer/hashLength. The k-mer value must be odd in order to avoid k-mer length contigs (if the input is even, it will subtract by 1 to make it odd) and it used to build the overlapping map of double-strand nucleotide, codon, and anticodon. The technique of this velvet assembler is use binary bit to represent the DNA nucleotide during the building the overlapping map that make it save about ¼ of memory storage usage. This process is save the k-mer length contigs in an array of unsigned long long integer. This also depends on how many memory system has to determine what would be the best use. Also, the overlap roadmap is saved into a file for a future backup and use. This will save a lot of time in the future when the second process is failed during the analyzing or needed to rerun.

The second part, velvetg. This use the two files that created by velveth (Sequence and Roadmaps) to build a graph and analysis the graph to create new DNA sequences. This called memory-intensive process to speed up the analysis. In this process, it used hash table with quicksort binary tree to build up the overlapping map from the Roadmaps file. This increased the search process when building the graph. The program also implements the de Burijin graph which is an efficient way to represent a sequence in terms of its k-mer components. For each read from the Sequence file, a node is created. Each node has a series of overlapping k-mers and adjacent k-mers overlap by k-1 nucleotides. Each node is also contain its own reverse complement creates bi-graph. Path is built from the graph, they connected together through arcs so the last k-mers of previous node overlaps the first k-mers of next node. The program implements concatenation to shorten the graph and save memory. It also uses “error correction” method to simplify the graph when it analyzing the sequences data in the graph.

The data structure use for this program is hash bucket structure. Without any option from the input command line, the program will use FASTA format file as a default to read. Each read from the original data files is performed in 4096 bytes buffer size until it reach the end of the file. Then the reformat sequence name line is recognized by ‘>” at the beginning of the line then it add the sequence index and category type to the line. Sequence line will be in 35 characters long. It puts all the modified sequence name line and DNA sequence line to a file name Sequences in the output directory specified by end user.

Next, the code will read the new sequence file and put all these sequences in order of array index into an array of the ReadSet struct. Each of index hold 35 character long sequence. Each codon in the sequence in this array is converted to nucleotide in number representing by 2 bits value from 0 to 3 (in tightString.c), A = 0, C = 1, G = 2, T = 3 and if it see N in the input sequence, it will replace it with A. For every 4 characters of the sequence are converted to nucleotide number and saved into 8-bit unsigned character. The position of each character in 4-character string is critical to where it placed in 8-bit char. The codon in positon 0 is placed in the first two bit 0-1, position 1 in bit 2-3, position 3 in placed 4-5, and position 3 in bit 6-7. Anti-codon or double-strand is created from the sequences of nucleotide number when the double strand option is ‘on’ and by default, it set to true.

The hashLength that specified by the end user from the binary velveth is considered as k-mer number in the code. The code will checks and makes sure it is not exceeded the MAXKMERLENGTH value (set as 31 in Makefile). As the conditions are met, k-mer sequence will saved in 64-bit long long variable. The first k-mer length\*2 bits are used and the rest of the bits are blanked out with value 0. K-mer length of codon sequence is extracted from the beginning of the codon sequence in an array of ReadSet struct. K-mer length of anti-codon sequence fragment is also extracted from the beginning of the anti-codon sequence. The values of the k-mer codon and anti-codon sequence fragments will be compared to each other. The smaller value between of the two will determined which sequence fragments will be used to save in the hash table. New k-mer sequence will be checked in the hash table for duplication, and if it is not duplicate then it will put into the hash table. Each index in the hash table is constructed in binary tree storage. The tree will be reconstructed based on the value of new k-mer sequence. If the new k-mer sequence fragments is found in the table, the overlapping information of the sequence fragment such as sequence index that have overlapped with, start position of a new fragment start to overlapping, start and finish position of overlap fragment that are overlapping with new fragments is put in a filename named RoadMaps in same directory where the filename Sequence is in.

The de Bruijn graph is started with the function importRoadmapArray (roadMap.c). The file RoadMaps will be read. The read data will be used to construct the annotation table in the roadmap table which is in hash table of the binary search tree. This overlapping information records position of the sequence in the node that starts overlap with start and finish positon of the sequence next node. After that, it will constructs the pregraph using the function newPreGraph\_pg (preGraphConstruction.c) based on the roadmap table. In this function, the function setInsertionMarkers is called to build the marker table in array of linked list based on the annotation table and sequence fragments. The marker table is in sorted alphabetical order, so again, it using annotation table, prenode list is created with arc. Pregraph is concatenated all the prenode in a single chain into a node using function concatenatePreGraph\_pg (concatenatedPreGraph.c) and exported into a filename PreGraph. File of Pregraph is used in the function importPreGraph (graphReConstruction.c). This then creates reference mapping and k-mer table along with the reads from PreGraph file to build the graph.

“Error correction” and concatenation is implemented in the function correctGraph (locallyCorrectedGraph.c). It use the clipTipsVeryHardLocally function to remove all the node that not in the longest paths. Bubbles also is called after the tips is removed from the graph. It use to remove all the nodes that start and end at the same nodes in the path and that is taken care by the function tourBus\_local (locallyCorrectedGraph.c). After all then it export to a file called Graph for the future building new DNA sequence. Graph is future cleaned out by using the function removeLowCoverageReferenceNodes (graphStats.c) and saved into filename contigs.fa. This file contains all the possible new DNA sequences from the original DNA sequences.

In conclusion, the velvet program is coded in the very well way to handle the memory allocation and cleanup. Also, the Bruijin graph is the most interesting of the program that make it save memory costs when it merges all the nodes together but does not affect the path generated in the graph. With the error removal method, the program give out the result in such a very fast way that 50,000 lines of DNA (from test file) can be analyzed with in just few seconds.