**Connor Elkhill**

**Module 1 Day 3 Stub Report**

**Introduction**

The Human Genome Project has been touted as influencing the following now two decades of genetic research, and for good reason. Officially starting in 1990, the Human Genome Project had the panache and spirit of the famed “space race” completed just a few decades before, with far less of the political underpinning. Comparisons aside, the Human Genome Project undoubtedly spurred genomic research at the time, setting it on the path to reach where it is today, with genomic research in near ubiquity in academic institutions throughout the world and revolutionary genetic technology being developed in the private sector on a regular basis1.

The sequencing technology used for the Human Genome Project was called “shotgun sequencing”. This method involves making many copies of the input DNA and then splitting it at random locations in the sequence, resulting in numerous small fragments of varying sizes and locations within the original contiguous sequence, similar to the spread of a shotgun. These fragments are then each sequenced and reported as reads, which can be in the counts of millions depending on the input sequence and the read length. These reads then need to be put back together into the original DNA sequence in a step called assembly. This advancement in DNA sequencing technology has been called next-generation sequencing, marking a large improvement in speed and scalability from the technological improvements2.

The *de novo* assembly of a DNA sequence from a next-generation sequencing experiment has long been the study of bioinformatics scholars and computationally minded researchers. Reconstructing the sequence of a fragmented string interestingly has many parallels to now seemingly ancient mathematical study from Hamilton, whose namesake operator plagued my undergraduate physics studies, and Euler, who did much the same for my math homework. The focus of assemblers has since turned from the former mathematician’s work to the latter, as was argued by Pevzner et al in 20013. Constructing a graph-based representation of a large genome can lead to an excess of nodes and edges that make computation time a significant consideration. The Hamiltonian path problem, whose solution has been proven NP-complete, has yet to have an efficient solution presented that makes that graph layout a viable option. Instead, the de Bruijn graph has been largely preferred. The solutions to de Bruijn graphs are Eulerian Paths, and instead of visiting each node once in the graph the goal is to visit each edge. Most importantly, there exist algorithms that can find a Eulerian path in a de Bruijn graph in linear time O(E) (assuming that the graph *has* a Eulerian path, see the famous Bridges of Konigsberg problem).

The specific problem presented in this exercise is as follows: *given a set of next-generation sequencing reads and a* *query sequence, find the largest contig that contains the query sequence.* The applications of finding the genetic context around a query sequence are rather wide—it could be used to verify a target vector was incorporated into the right location in the genome after CRISPR/Cas9 editing, it could be used to search for contamination sequences to indicate experimental errors, et cetera.

To solve this problem, I constructed a de Bruijn graph from the input next-generation sequencing reads and corrected errors where appropriate based on a simple degree threshold algorithm. Since the direction of next-generation sequencing reads are ambiguous the reverse complement sequence will also be used to construct the graph, allowing for the solution of said graph to include both the forwards and the reverse complement of the input sequences. Once the graph is constructed, the solution is seeded by generating an arc that consists of the query sequence. The solution is extended in the forwards and then reverse directions using a non-exhaustive depth-first search algorithm based on Fleury’s algorithm4, resulting in a contig of some length L that contains the query sequence. Since this algorithm is non-exhaustive, meaning it will only construct one path and will not find all paths, the algorithm will be repeated N times and the largest path found will be deemed the winner. Once the winning solution is found, the indices of the read sequences that contribute to the winning sequence are reported in tabular format.

**Input**

This program takes two inputs. The first is a FASTA file containing the reads collected from the next-generation sequencing run. The start of each read is indicated by a line starting with “>” called a defline which describes the read with a sequence identifier and the next line starts the sequence data itself. An example of a FASTA file (taken from the ncbi website link [here](https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=BlastHelp)):

>P01013 GENE X PROTEIN (OVALBUMIN-RELATED)

QIKDLLVSSSTDLDTTLVLVNAIYFKGMWKTAFNAEDTREMPFHVTKQESKPVQMMCMNNSFNVATLPAE

KMKILELPFASGDLSMLVLLPDEVSDLERIEKTINFEKLTEWTNPNTMEKRRVKVYLPQMKIEEKYNLTS VLMALGMTDLFIPSANLTGISSAESLKISQAVHGAFMELSEDGIEMAGSTGVIEDIKHSPESEQFRADHP

FLFLIKHNPTNTIVYFGRYWSP

The second input is also a FASTA file, but this time containing the sequence that will be queried. The data presented in this exercise are named READS.fasta (next-generation sequencing reads) and QUERY.fasta (query sequence). There are 124,520 reads within the READS.fasta file, each of varying sequence lengths. The QUERY.fasta file contains only one sequence with a length of 648.

**Code Arguments**

There are two required arguments to this code, the reads FASTA file and the query FASTA file. The most simple command line call of the program goes as follows:

python main.py -reads\_fasta ../data/READS.fasta -query\_fasta ../data/QUERY.fasta

More advanced command line options and parameters can be found in the README.md file attached to this submission.

**Output**

There are three default outputs to this program. The first is the constructed de Bruijn Graph saved in JSON format (citation) which can be reused to quickly repeat analysis if desired. The second is a FASTA format file (see above for information about the format) detailing the longest contig that contains the input query sequence. This file is titled ***ALLELES.FASTA***. The third default output file will be a tab-delimited format named ***ALLELES.aln***. This file will resemble the following example table below (reproduced from the Day 3 instructions document):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| sseqid | qseqid | sstart | send | qstart | qend |
| 2S43D:08461:04180 | contig1 | 13 | 40 | 1 | 64 |
| 2S43D:07701:07310 | contig1 | 20 | 112 | 240 | 332 |
| 2S43D:07489:10315 | contig1 | *123* | *90* | 20 | 53 |
| 2S43D:04035:14719 | contig1 | *105* | *41* | 10 | 74 |

The columns are as follows (reproduced from the Day 3 instructions document):

|  |  |
| --- | --- |
| sseqid | : name of sequencing read (from READS.fastq.gz) |
| qseqid | : name of contig matched (from ALLELES.fasta) |
| sstart | : starting coordinate in sequencing read sseqid that matches qseq |
| send | : ending coordinate in sequencing read sseqid that matches qseq |
| qstart | : starting coordinate in contig that matches sseq |
| qend | : ending coordinate in contig that matches sseq |

In addition to the two default output files, there are a few optional output files that are as follows:

1. ***kmer\_table\_(un)corrected.dat:*** A serialized pickle file (citation) containing the kmer table used to construct the de Bruijn Graph. Will contain “corrected” or “uncorrected” depending if error correction was performed.

2. ***KmerDegreeHistogram.csv:*** A CSV file containing the tabulated histogram summarizing the degree of each unique kmer found in the kmer table. The sum of the frequency of the kmers will be equal to the number of edges in the graph.

3. ***ReadLengthHistogram.csv:*** A CSV file containing the tabulated histogram summarizing the length of reach input read.

**Implementation**

This program is implemented in Python 3.8.3 and runs via the command line. The time to run this program on a laptop with a kmer size of 22 was 23.6 minutes using the provided READS.fasta and QUERY.fasta files. More information about program specifications can be found in the README.md file provided with this report.

**Algorithms**

This program contains 5 main algorithms that perform the main functionality needed (aside from reading/parsing or writing files). The algorithms are as follows:

**Algorithm 1: *Parse all kmers and correct errors.***

The first step in the program is to parse the input reads and generate a set of kmers of size K as specified by the user. For each read, we split the read into N kmers of size K, dependent on the length L of the read itself. The kmers for each read are stored in a large list and are not unique (there will be multiples of the same kmer within the list). Then, the unique kmers within the list are determined and the frequency of each unique kmer is calculated. Kmers with a frequency less than or equal to the threshold T (default of 2) are suspected to be containing at least one sequencing error within its length. To correct the error, the algorithm searches for kmers within a Hamming distance of 2 that have a frequency greater than the threshold T and upon finding a kmer that meets that criterion the degree of the errored kmer is added to the degree of the corrected kmer. This algorithm takes an aggressive approach to graph pruning and the errored kmer is deleted regardless of if a correction is found.

TODO: Make kmer histograms showing before and after for an example dataset.

This approach is simplistic and makes numerous assumptions about error correction. Further directions and more robust implementations are discussed in the *Future Directions* section of this report.

**Algorithm 2:*****Create directed de Bruijn multigraph.***

After kmer creation and error correction, the directed de Bruijn multigraph (DBG) is created. To construct the DBG each kmer is split into its prefix and suffix, called a k-1 mer. A new node in the graph is created for each k-1 mer and an edge is drawn between the prefix k-1 mer and suffix k-1 mer nodes with a degree equal to the degree of the kmer as determined by **Algorithm 1**. In this algorithm each edge is actually drawn as two edges, one edge pointing forwards from the prefix k-1 mer to the suffix k-1 mer and another edge pointing backwards from the suffix k-1 mer to the prefix k-1 mer, creating a bidirectional linked list. This implementation removes the need to rebuild the graph when creating a contig in **Algorithm 4.** An example of the construction of a de Bruijn graph using a simple string can be seen below (graphic credits to Dr. Ben Langmead5):

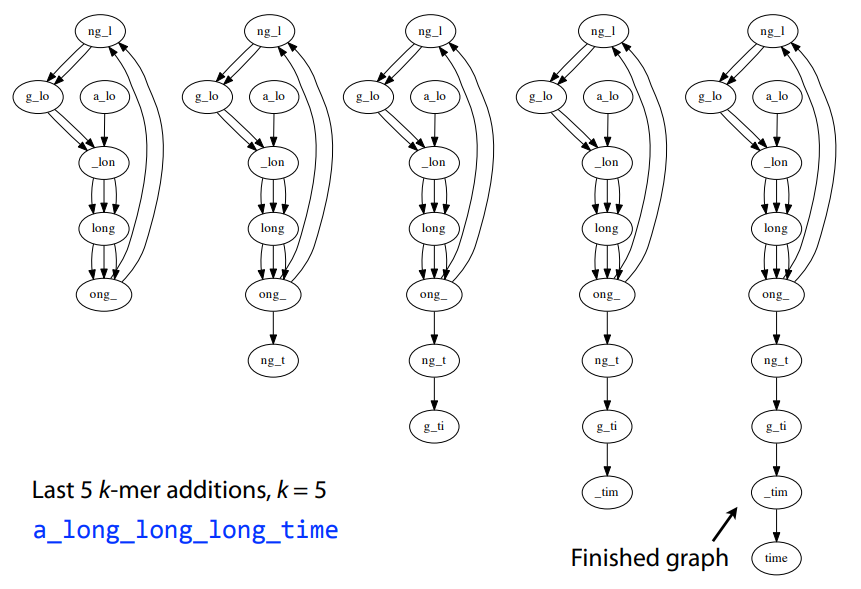


Figure 1: Construction of a de Bruijn graph reproduced from slides created by Dr. Ben Langmead5.

NOTE: A crucial property of de Bruijn graphs is that they are *multigraphs*—meaning that each node can have redundant edges to an adjacent node. This property is one reason why this graph format is able to handle repeats within the sequence reads. We store exactly how many times the prefix k-1 mer connects to the suffix k-1 mer6.

Each node will be an object storing the node sequence and the list of adjacent nodes that it connects to with an edge. An example of the node structure is as follows (example kmersize = 8):

class **GraphNode** {

seq: “GCATCATA”,

next: [“A”,”A”,”A”,”C”,”C”,”T”],

previous: [“C,”C”,”C”,”T”,”T”]

}

In order to account for the ambiguity of the reads and allow the input read sequence to be either “forwards” or “backwards” in the final contig, we will also construct the reverse complement node for each node. The number of edges within our DBG is determined by the following formula:

Where N is the total length of all reads, n is the number of reads, and k is the kmersize. At most, the total number of nodes within the graph can be twice the number of edges, however the true number of nodes in the graph is always much smaller due to repeated k-1 mers within the sequencing reads. For example, with a kmersize of 22 there are E edges within the resulting DBG and 602,446 unique nodes after performing algorithms 1 and 2.

***Algorithm 3: Search for an arc containing our query sequence.***

Before the final solution contig can be found, we need to seed our solution with our query sequence. This will ensure that all of our solution contigs contain our query sequence. To see our solution, we read in our query sequence and split that sequence into k-1 mers. Starting at the first k-1 mer in the query sequence, we identify the node associated with that sequence. We continue along each k-1 mer in our query sequence, adding the nodes to our solution list. Each time we add a node to our solution list, we remove the edge from the node.next list and we also remove the reverse complement edge from the node.next list within the reverse complement node. Additionally, we remove the previous edge from the next k-1 mer and its reverse complement. As we move along, we check if the node associated with the next k-1 mer in the sequence exists in the node.next attribute of the current node. If the next k-1 mer is not in this list or if at any point a node is not found for a k-1 mer in our query sequence, then our graph does not contain a solution with our query sequence; the program is immediately stopped, and the user is notified.

***Algorithm 4: Extending our solution with Fleury’s Algorithm and depth first search.***

Once our solution is seeded, it is time to extend our solution in both the “forwards” and “reverse” directions. We start with the “forwards” direction, where our starting node is the final node found in the seeded solution arc that was created in Algorithm 3. We continue forwards applying the simple principle from Fleury’s algorithm to make the decision: we chose the next node in the graph that will not create a bridge—that is, the next node has to have greater than one edge to another node. If there is no next node in the graph with degree greater than one, the node with the greatest degree is selected.

As we select the next node and traverse the graph, we again remove the edge between the current node and next node. The algorithm halts once the length of next node.next is equal to zero. Then, the exact same algorithm is repeated in the “reverse” direction. Because we have a bidirectional linked list, the implementation of this reversal of direction simply requires us to change “node.next” to “node.previous”. The same portion of Fleury’s algorithm is used to check for bridges before deciding on the next node. Once the reverse direction is exhausted as the forward direction was (that is, the length of next\_node.previous is zero) the algorithm stops and the contig is reported as a possible solution.

This algorithm is a form of non-exhaustive depth first search, meaning it only finds one path through the graph, and does not traverse every single path. It would be computationally prohibitive to search for every possible path given the size of the graph, the reality of possible errors that went uncorrected, and potential gaps between the reads. Therefore, this algorithm is repeated N times in order to create N paths that are all possible contigs. The longest contig from the list of N is the “winning” contig and will be reported.

***Algorithm 5: Find where our winning contig matches our reads.***

The final step in this program is to find the indices of where each read contributes to the final winning contig and generate the ALLELES.aln file. If a read does not contribute to the final contig, it is not reported in the output ALLELE.aln file. To find where our reads contribute to the winning contig, we loop through each read and search for the read string within our contig, which is identical to searching for a substring within a string. We also search for the reverse of the read within the winning contig. Since we appropriately removed edges from the reverse compliment node of each node while traversing our graph, only the “forwards” or the “reverse” direction of the read will match the winning contig. Additionally, the contig will only match completely in one location within the winning contig. The indices of the winning contig containing a given read will be added to our output table, along with the indices pertaining to the query sequence of those respective nucleotides.

**Assumptions and limitations**

De Bruijn graphs are often used by next-generation sequencing assembly tools, such as SPAdes7. However, the actual implementation deviates quite a bit from the theory they are built upon. With perfect coverage and no sequencing errors de Bruijn graphs contain Eulerian paths as their solutions. Unfortunately, neither of these cases are true as there will never be perfect coverage of each location within the genome due to inherent biases in the technology, and the sequencer is going to make errors regardless of the exact sequencing machine that is used (CITATION Ross et al). Therefore, many assemblers create contigs, which are not a Eulerian path through the de Bruijn graph, but instead a portion of the graph that is comprised of a contiguous sequence. While these assemblers go a step further and use them as scaffolds to reconstruct the whole dataset, creating one contig is a perfect solution for the proposed problem, as we are only interested in the contig that contains our query sequence.

While de Bruijn Graphs fit our problem perfectly with their ability to quickly generate contigs, a large limitation of de Bruijn Graphs is the inefficiency of memory used by this method. The nodes in our graph are overlapping k-1 mers that contain k-2 of redundant information, causing the current implementation of this program to create graphs that are gigabytes in size when loaded into memory. While modern computing hardware makes this feasible for running on a laptop, there are numerous ways to optimize the storage solution so that memory can be saved wherever possible. One such example is the creation of a multisized de Bruijn graph, as is implemented in Spades (citation). This approach attempts to collect the advantages of small kmer sizes, that is represent more data with fewer nodes, as well as the distinct advantage of larger kmer sizes to resolve long, repeating portions of the DNA. Another example is called graph simplification and is implemented in another popular assembler named Velvet (citation). Simplification identifies linearly connected subgraphs within the larger graph and collapse them into one node, making what was many nodes into one. While both of these algorithms are wise choices to make the implementation more efficient, the development time to implement such features are outside the scope of this problem. More information is discussed in the *Future Directions* section below.

One final note of limitation for this program is the actual programming language chosen itself, Python. While Python is a wonderful language that is quick to develop and debug, and therefore invaluable in a context where a program needs to be made as robust as possible with little time, it is incredibly slow and lacks granularity and fine control. A program such as this would greatly benefit from composition in a language that allows the user to control memory operations and does not have a built-in garbage collector like Python. This would allow us to create an efficient hashing function to store each nucleotide with only two bits, as is commonly implemented in widely popular assemblers. Efficient, fast operation is a hallmark feature of other popular languages such as C++ or Rust and I would implement my solution in one of these languages (Rust seems great!) were I to release it to the larger scientific community.

One of the largest assumptions we are making in this program is that the query sequence exists within a possible contig we can generate given the next-generation sequencing reads. In practice, it would be more than reasonable to assume our query sequence *could not* *perfectly* match any contig we can generate from the reads. A more robust implementation would be to allow for small substitutions or deletions between the query sequence and our reads. We would have to run some sort of alignment algorithm to find the “best match” of each k-1 mer in our query sequence to the k-1 mer nodes within our graph. A “best match” arc could then be constructed using a dynamic programming algorithm, similar to how BLAST finds the “best match” in a query. This is especially important for ambiguously read nucleotides, denoted with an “N” by the sequencer. Currently, this program does not consider “N” and will not run with “N” in the reads, nor will it find a match to an “N” in a query sequence. This assumption is further discussed in the *Future Directions* section below.

One final assumption within this program is the scope of the problem these algorithms are designed to solve. Nearly all of the algorithms implemented take advantage of the fact that we only need one single contig from a field of possibly many that could be reconstructed from the input reads. This means that most reads will not contribute to our output contig, which is in fact what we see in the output of a solution based on kmer size of 22, where only 136 of all input reads comprise our solution sequence. This fact is taken advantage of during our error correction, where we eliminate possibly correct nodes that have low degree in order to aggressively reduce graph size, as well as our final algorithm, where we use a simple loop to examine the contribution of each read to the final output sequence. Were the sequence longer or were there multiple output contigs that needed to be considered, looping through each read for each contig would not be an efficient solution, as there would be more matches found overall and nested for loop structures are notoriously inefficient.

**Future Directions**

After spending numerous hours reading literature, writing down algorithms in pseudocode, and developing the Python tools, I have been able to build a program that I am proud of that answers the proposed question appropriately. That being said, I have found numerous ways in which this program can be improved, expanded upon, and re written. I would like to discuss a few of the key choices of the current implementation and ways the program could be improved with future work.

The most glaring concession that was made during development was the simplicity of the error correction algorithm (algorithm 1) that was implemented. More robust methods such as Velvet’s correction of topological features of tips, blocks, and erroneous connections are all effective correction methodologies that would go a long way to decreasing graph size and improving performance. The method implemented by algorithm 1 does not address any of these higher order features and merely thresholds based on node degree, likely resulting in erroneous removal of nodes that are not actually errors but instead areas of lower coverage. In the context of a small contig within our graph that we are searching for with the query sequence, removing a small number of correct nodes should not adversely affect our solution. However, were the problem to create an ensemble of contigs in attempt to perform *de novo* assembly, this would not be a valid solution. Only in the context of this smaller problem does the current error correction make sense.

**Future Directions**

There are numerous different areas that could be improved upon in this program based on the limitations, assumptions, and design choices that were made. Perhaps the most pertinent to the accuracy of the program lies within the error correction of the kmers, as seen in Algorithm 1. The approach this program takes is quite simple: define a threshold and remove any kmer that is below that threshold, correcting wherever possible to another kmer within a Hamming distance of 2 that is above the threshold. This strategy effectively prunes the graph quite aggressively, saving on computational time and space for further algorithms. Furthermore, it is reasonable to implement such an aggressive strategy in this case as our problem does not require full assembly, only the construction of 1 contig that contains our query sequence. In the case of a kmer size of 22, there are 602,446 nodes in our graph. The final longest contig assembled from that graph containing the query sequence was only TODO: 907 base pairs long, meaning that at most only 0.15% of the nodes in the graph are actually used to find the final solution. Therefore, this tradeoff by reducing robustness of the error correction algorithm in favor of greater computational efficiency of the downstream steps is appropriate for this specific problem.

**References**

1. Green, E. D. Strategies for the systematic sequencing of complex genomes. *Nat Rev Genet* **2**, 573–583 (2001).

2. Behjati, S. & Tarpey, P. S. What is next generation sequencing? *Arch Dis Child Educ Pract Ed* **98**, 236–238 (2013).

3. Pevzner, P. A., Tang, H. & Waterman, M. S. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences* **98**, 9748–9753 (2001).

4. Fleury, P.-H. Deux problèmes de Géométrie de situation. *Journal de mathématiques élémentaires, 2nd ser. (in French)* **2**, 257–261 (1883).

5. Langmead, B. De Bruijn Graph assembly.

6. Compeau, P. E. C., Pevzner, P. A. & Tesler, G. How to apply de Bruijn graphs to genome assembly. *Nat Biotechnol* **29**, 987–991 (2011).

7. Bankevich, A. *et al.* SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *J Comput Biol* **19**, 455–477 (2012).