Genome Polymorphism Detection Using De Bruijn Graph

1st Dhruvjyoti Swain
dept. of AI
Amrita Vishwavidyapeetham Amritapuri
Amritapuri, India
amenu4aie20023@am.students.amrita.edu

2nd Dhanush Krishna R
dept. of AI
Amrita Vishwavidyapeetham Amritapuri
Amritapuri, India
amenu4aie20021@am.students.amrita.edu

3rd Pavithra P M Nair
dept. of AI
Amrita Vishwavidyapeetham Amritapuri
Amritapuri, India
amenu4aie20055@am.students.amrita.edu

4th Pranav Jayasankar Nair
dept. of AI
Amrita Vishwavidyapeetham Amritapuri
Amritapuri, India
amenu4aie20056@am.students.amrita.edu

***Abstract*—This paper aims to establish an understanding of genome polymorphisms and their detection using De Bruijn Graphs.**

***Index Terms*—Genome, Polymorphism, sequencing, De Bruijn Graph**

I. Introduction

Polymorphism involves one of two or more variants of a particular DNA sequence. The most common type of polymorphism involves variation at a single base pair. Polymorphisms can also be much larger in size and involve long stretches of DNA. Called a single nucleotide polymorphism, or SNP (pronounced snip), scientists are studying how SNPs in the human genome correlate with disease, drug response, and other phenotypes.

Detecting polymorphisms in the genome is an important task for an individual specimen (disease association studies) and for a species as whole (phylogenetic tree reconstruction). Whether it be identifying single nucleotide polymorphisms (SNPs) in an individual compared to a reference genome or comparing different species, identifying polymorphic differences is a difficult task. Methods, however, are usually extremely conservative and only identify simple variation (SNPs, insertions, deletions) leaving more complex variation (translocations, inversions) unexamined.

Genomic variation such as translocations and inversions have been shown to cause many human diseases. Translocations have been shown to be the cause of several different types of cancer, such as Burkitt's lymphoma [1] and acute promyelocytic leukaemia . They have also been shown to be associated with schizophrenia [2] . Studying and identifying different types of genomic polymorphisms could have impact on two very important fields in biology: genome wide association studies as well as phylogenetic tree reconstruction.

A. Genome Wide Association Studies

In genomics, a genome-wide association study (GWA study, or GWAS), also known as whole genome association study (WGA study, or WGAS), is an observational study of a genome-wide set of genetic variants in different individuals to see if any variant is associated with a trait. GWA studies typically focus on associations between single-nucleotide polymorphisms (SNPs) and traits like major human diseases but can equally be applied to any other genetic variants and any other organisms.

In GWAS, next-generation sequence (NGS) reads are mapped to a reference genome. Differences, commonly SNPs and indels, are then identified from the read mapping results. This method has helped identify and associate many mutations with different diseases. Read mapping, however, is a difficult task. More than 10% of reads were unmapped when mapping 12.2 million reads to the human genome using the popular Burrows-Wheeler Aligner . Some of the reads will be left unmapped due to errors generated during sequencing. Other reads are left unmapped for unknown reasons. It may be that some unmapped reads vary significantly from the reference genome making read mapping difficult. Mapped reads represent reads that are similar enough to the reference genome to be mapped with a given set of parameters. Unmapped reads may contain more interesting and novel biological information than mapped reads because these reads diverge enough from the reference genome to remain unmapped. Harnessing unmapped reads enables more thorough analysis of how individuals within a species differ and how genomic rearrangements may affect phenotypes.

B. Phylogenetic Tree Reconstruction

Phylogenetic tree reconstruction is usually done by comparing homologous gene sequences in a group of species of interest. Identifying the homologous genes is a difficult task and is often a conservative process, allowing for only gene sequences that are very similar to be clustered together. This approach is limited because it only allows for comparing gene sequences instead of comparing whole genomes . Comparing the entire genome of one species to another is valuable to see if genomic rearrangements or other structural variations occurred to the genome. Accounting for these genomic variations may serve as a future phylogenetic signal in future phylogenetic tree reconstruction.

II. Methods

For utilizing unmapped reads and to compare whole genomes is to construct a relaxed de Bruijn graph that allows for more complex genomic variation to be observable.

A. Standard De Bruijn Graph

A standard de Bruijn graph is a graph structure that represents the genome of an organism. de Bruijn graphs are commonly used for genome assembly and usually representative of a single species .Beyond genome assembly, they have also been found to increase the percent mapped reads when mapping reads to a de Bruijn graph versus contigs. In a de Bruijn graph, each node represents a unique kmer. Edges in the graph represent kmer overlaps. The graph is usually constructed from NGS reads where reads are broken into kmers and used to populate the graph.

In graph theory, the standard de Bruijn graph is the graph obtained by taking all strings over any finite alphabet of length l as vertices and adding edges between vertices that have an overlap of l-1. In the following, we consider assembly using a slightly modified version of the standard de Bruijn graph from the L-spectrum of a genome.

B. Relaxed De Bruijn Graph

Our relaxed de Bruijn graph differs from a standard de Bruijn Graph is two major ways:

- The graph contains sequence information for multiple species
- Kmers can occur multiple times in the graph

By relaxing these constraints on the de Bruijn graph, we are able to identify interesting genomic variation in a tractable amount of time and space. Conceptually, this method can be thought of as merging two separate deBruijn graphs by exploiting uniquely occurring kmers in one sequence as anchor points to merge the graphs.

Algorithm 1 Creating De Bruijn initially

```
1: function CONSTRUCT($seq, k$)
2:     $occs$ ←occurrences of each‘kmer
3:     $curridx \leftarrow 0$
4:     $g \leftarrow$ empty graph
5:     for each kmer $kmer$ in seq do
6:         $l \leftarrow$ prefix of kmer
7:         $lidx \leftarrow curidx$
8:         $curidx \leftarrow curidx$ + 1
9:         $occs[l] \leftarrow occs[l]$ + 1
10:        $rlookup[l] \leftarrow lidx$
11:        $r \leftarrow$ suffix of $kmer$
12:        $ridx \leftarrow curidx$
13:        $curidx \leftarrow curidx$ + 1
14:        $occs[r] \leftarrow occs[r]$ + 1
15:        $rlookup[r] \leftarrow ridx$
16:        $lPos, rPos \leftarrow reverselookup(lidx, ridx)$
17:        $g.addedge(lPos, rPos)$
18:    end for
19:    return $occs, curidx, g$
20: end function
```

Algorithm 2 Appending New Sequence to existing De Bruijn Graph

```
1: function APPEND($g, occs, curidx, seq, k, rlookup$)
2:     for each kmer $kmer$ in seq do
3:         $l \leftarrow$ prefix of kmer
4:         if $occs[l] > 0$ then
5:             $lidx \leftarrow rlookup[l]$
6:         else
7:             $lidx \leftarrow curidx$
8:             $curidx \leftarrow curidx$ + 1
9:         end if
10:        $r \leftarrow$ suffix of $kmer$
11:        if $occs[r] > 0$ then
12:            $ridx \leftarrow rlookup[r]$
13:        else
14:            $ridx \leftarrow curidx$
15:            $curidx \leftarrow curidx$ + 1
16:        end if
17:        $lPos, rPos \leftarrow reverselookup(lidx, ridx)$
18:        $g.addedge(lPos, rPos)$
19:    end for
20: end function
```

III. Results and Discussion

We have taken sequence ”ACTTACGTACTTG” and created the De Bruijn graph as seen in Fig1. Then we have taken the sequence ”ACTTACTTACT” and appended the new k-1 mers onto the De Bruijn graph as shown in Fig2.

- Graph Construction: Our graph construction algorithm is outlined in Algorithm 1, also see Figure for a visual representation of the graph construction process. After graph construction, we simplify the graph by collapsing neighboring nodes in a graph where that path through the nodes is unambiguous to form unitigs.

- Implementation: The NetworkX python package [4] was used for storing and manipulating de Bruijn graphs,and gvmagic [5] was used for graph visualization.

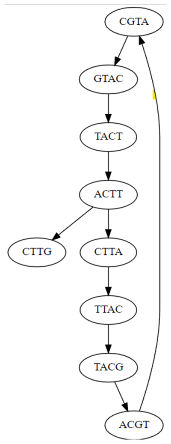

Fig. 1. Initial construction of De Bruijn Graph as demonstrated by Algorithm 1

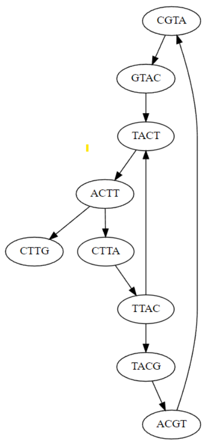

Fig. 2. Graph after appending the second sequences k-1 mers as demonstrated by Algorithm 2

Conclusion

In this work, we have presented an method for constructing a relaxed de Bruijn graph and explained genome polymorphism and phylogenetic Tree reconstruction. We can use the relaxd De Bruijn Graph to identify graph structurs and these may be used for signals for phylogenetic tree reconstruction or we can use them for association studies for phenotypes.

References

[1] R. Hoffman, E. J. Benz Jr, L. E. Silberstein, H. Heslop, J. Anastasi, and J. Weitz, Hematology: basic principles and practice. Elsevier Health Sciences, 2013.

[2] J. E. Eykelenboom, G. J. Briggs, N. J. Bradshaw, D. C. Soares, F. Ogawa, S. Christie, E. L. Malavasi, P. Makedonopoulou, S. Mackie, M. P. Malloy et al., “A t (1; 11) translocation linked to schizophrenia and affective disorders gives rise to aberrant chimeric disc1 transcripts that encode structurally altered, deleterious mitochondrial proteins,” Human molecular genetics, vol. 21, no. 15, pp. 3374–3386, 2012

[3] H. Li and R. Durbin, “Fast and accurate short read alignment with burrows–wheeler transform,” Bioinformatics, vol. 25, no. 14, pp. 1754–1760, 2009.

[4] A. Hagberg, P. Swart, and D. S Chult, “Exploring network structure, dynamics, and function using networkx,” Los Alamos National Laboratory (LANL), Tech. Rep., 2008.

[5] V.N. Kasyanov & E.V. Kasyanova (2013) Information visualisation based on graph models, Enterprise Information Systems, 7:2, 187-197, DOI: 10.1080/17517575.2012.743188

Appendix

```
def prefix(kmer):
    n = len(kmer)
    return kmer[0:n-1]

def suffix(kmer):
    n = len(kmer)
    return kmer[1:n]

def kmer_composition(seq, k):
    kmers = []
    for i in range(len(seq)-k+1):
        kmers.append(seq[i:i+k])
    return kmers

def reverse_lookup(rlookup, lidx, ridx):
    key_list = list(rlookup.keys())
    val_list = list(rlookup.values())
    position = val_list.index(lidx)
    lPos = key_list[position]
    position = val_list.index(ridx)
    rPos = key_list[position]
    return lPos, rPos

def visualize_debruijn(g):
    nodes = set()
    for elem in list(g.nodes(data=True)):
        nodes.add(elem[0])
    edges = list(g.edges())
    dot_str= 'digraph "DeBruijn graph" {\
        n '
    for node in nodes:
        dot_str += '    %s [label="%s"]
            ;\n' %(node, node)
    for src, dst in edges:
        dot_str += '    %s->%s;\n' %(src,
            dst)
    return dot_str + '}\n'

def CONSTRUCT(seq, k):
    occs = {}
    curidx = 0
    g = nx.DiGraph()
    rlookup = {}
    kmers = kmer_composition(seq, k)
    for kmer in kmers:
        l = prefix(kmer)
        if l not in occs.keys():
            occs[l] = 0
            rlookup[l] = 0
        lidx = curidx
        curidx += 1
        occs[l] = occs[l] + 1
        rlookup[l] = lidx
        r = suffix(kmer)
        if r not in occs.keys():
            occs[r] = 0
            rlookup[r] = 0
        ridx = curidx
        curidx += 1
        occs[r] += 1
        rlookup[r] = ridx
        lPos, rPos = reverse_lookup(
            rlookup, lidx, ridx)
        g.add_edge(lPos, rPos)
    return occs, curidx, g, rlookup

def APPEND(g, occs, curidx, seq, k,
    rlookup):
    kmers = kmer_composition(seq, k)
    for kmer in kmers:
        l = prefix(kmer)
        if l not in occs.keys():
            occs[l] = 0
        if occs[l] > 0:
            lidx = rlookup[l]
        else:
            lidx = curidx
            curidx += 1
        r = suffix(kmer)
        if r not in occs.keys():
            occs[r] = 0
        if occs[r] > 0:
            ridx = rlookup[r]
        else:
            ridx = curidx
            curidx += 1
        lPos, rPos = reverse_lookup(
            rlookup, lidx, ridx)
        g.add_edge(lPos, rPos)
```