Bioinformatics
doi: 10.1093/oxford-bioinformatics/
Advance Access Publication Date: DD Month 2017
Manuscript Category

Subject Section

Read Mapping and Variant Detection in PanTools

Adzkia Salima Nindyantoro[1]

[1]Department of Bioinformatics, Wageningen University

Supervisor: Dick de Ridder, Siavash Sheikhizadeh Anari, Department of Bioinformatics, Wageningen University

Abstract

Motivation: Determining sequence variation is fundamental in genetic research. The most common approach is aligning short sequence reads to a reference genome and detecting polymorphisms. In order to get all variants, multiple reference genome can be used in read mapping step. However repeatedly aligning reads to different genomes is not efficient.

Results: We present a pan-genome approach to read mapping using PanTools as well as an algorithm to detect variation. Read mapping is performed using the pan-genome construction algorithm in PanTools. It produces a compressed De Bruijn graph stored in a Neo4J graph database. Variant detection is then performed using an algorithm implemented in Java API of Neo4j.

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 INTRODUCTION

Next Generation Sequencing (NGS) technologies have drastically improved our ability to sequence genomes, by producing millions of reads quickly and cheaply. As a result, re-sequencing has become the most popular genome analysis workflow to determine genetic variatons of a sample. The most common approach to discover sequence variation is by aligning sequence reads to a reference genome and looking for polymorphisms.

Using a single reference genome for read mapping has some disadvantages. First, due to limitations of both the NGS machine and the aligner, an assembled genome may not perfectly reflect the true genome sequence. Second, the reference genomes of individuals within species could be very divergent, such as in Arabidopsis thaliana (Clark *et al.*, 2007). Third, as the number of available sequenced genome increases, detecting variants with respect to single reference genome becomes rather arbitrary. But, aligning reads to a collections of reference genome one by one is time consuming and impractical. For this reason, effort has been made to allow read mapping to be performed on multiple genomes using a single data structure known as a pan-genome.

A pan-genome is defined as a collection of genomic sequences to be analysed jointly or to be used as a reference (Marschall *et al.*, 2016). It can be represented as simply a set of aligned sequences or as a graph structure. To be used in comparative genomics analysis, the pan-genome representation should allow a long-term storage. One way to achieve this is by building a De Bruijn graph (DBG) using an online algorithm and storing it in Neo4j graph database, as proposed by Sheikhizadeh *et al.* (2016) in their PanTools project. A De Bruijn graph consist of nodes corresponding *k*-mer that are connected by edges when *k*-1 character overlap. When a path in a DBG is non-branching, its nodes can be joined to form a compressed DBG thus reduce its space complexity. PanTools has an “add genome” functionality to align one genome sequence to the whole pan-genome. In this work, as a preliminary approach, we used this function to align reads to the pan-genome.

Subsequent variant calling can be done by detecting bubble structures in the pan-genome graph. A bubble is a subgraph with a closing bifurcation caused by a single nucleotide polymorphism. Casani (2015) developed a Cypher query to mine the structural variations in Neo4j graph database, but the query only works for two sequences. This Cypher query implementation is also theoritecaly slower compared to the java implementation. To detect bubble structure in multiple sequences, tools such as Cortex (Iqbal *et al.,* 2012) and 2k + 2 (Younsi, MacLean 2014) have been developed. These two algorithms detect bubble by looking for a *source* and *sink* node. A *source* is the node where the sequence from two or more genomes diverge. From the source, all the child nodes are traversed using a breadth first search to find the sink. In this work, we used this graph traversal to detect variation in PanTools using Java.

2 MATERIALS AND METHODS

Our experiments were conducted on a Linux server (Ubuntu14.04) with an Intel® Xeon® X5660@2.8GHz, with 24 logical cores, 64GB RAM and a 32GB RAM disk.

2.1 Data

To build the pan-genome, we used the genomes of *Escherichia coli* O157 strain 180-PT54 and *Escherichia coli* O157:H7 str. Sakai. Both were downloaded from NCBI on 17/8/2017. We also used publicly available Illumina MiSeq read data genomic of *Escherichia coli* O157: SRR6207503, downloaded from SRA archive NCBI. The reads are 220 bp long in average, with 34-fold coverage. *E. coli* is suitable for this experiment because it has a small genome size (5Mbp) and it is a well-studied organism.

2.2 Read mapping

2.2.1 Preprocessing the reads

We trimmed the adapter by cutting the first and the last ten bases of each read using the sed command. We also cut the bases with quality lower 30 using a sliding window method with Sickle (Joshi, 2012). We reduced the number of reads to obtain a 6-fold coverage to avoid long running times.

2.2.2 PanTools

To map a read to a pan-genome, we used the build genomes function in PanTools (Sheikhizadeh *et al,* 2016). PanTools is a software package that provides functionality to build, store and analyses a pan-genome. The functionalities are divided into two types; 1. sequence layer and 2. annotation layer. In the first type, PanTools has some functions to construct a pan-genome, add sequences/genomes, reconstruct genomes, compare and query pan-genomes. In the second layer, its features are annotating pan-genomes, grouping genes, retrieving gene sequences or genomic regions. We only used the sequence layer in this work, so from now on we will just focus on this layer.

PanTools uses Neo4j, a graph database that stores its data instances in the nodes and its relationships as edges between these nodes. It has a different structure in the different layers. In the sequence layer, the graph has a structure as shown in Figure 1. The nodes have 4 types of labels, each with different properties;

a) pangenomes: contains statistics about the whole graphs (*k*-mer size, number of nodes, number of edges);
b) genome: contains IDs and number of sequences;
c) sequence: In genome, a sequence is regarded as a contig, with properties contig's name and contig's length;
d) node: We store DBG in this label. The properties are nucleotide sequence, last *k*-mer and first *k*-mer.

The nodes are connected with two types of relationship; "has" and "coordinate". The "Has" relationship type does not have any properties. It simply links item eg. "pan-genome has genome" and "genome has sequence". The coordinate relationship has four label: FF, FR, RF, and RR. It describes the orientation (F: forward, R: reverse) of the *k*-mer on its source and destination nodes. The coordinates of the *k*-mer are stored as a property in the incoming edge of the node containing the *k*-mer. For example: in Figure 1, the first three nucleotides of genome 1 and contigs 1 is CTT. The incoming relationship for the node CTT has property a1_1: 0, indicating the nucleotides located in the first sequence of the first genome.

We provide the fasta file of the SRA reads and a fasta file of the genome as an input. The "construct pan-genome" algorithm works in three steps. First, it creates an index database for the *k*-mer of the genomes and the reads. Second, it constructs a pan-genome using an online algorithm to create a compressed DBG. Third, it localizes the *k*-mer in the CDBG by adding coordinates in incoming relationships of the nodes.

Since we only used build pan-genome function, PanTools does not differentiate between the genome and the read sequences. For that reason, we considered the last number of the sequence as the reads file. The reads are treated in the same way as contigs in a genome. Thus, the property of coordinate relationship a3_2:0 means that the node, that relationship points to is aligned with the second read in the file at the first (0) position.

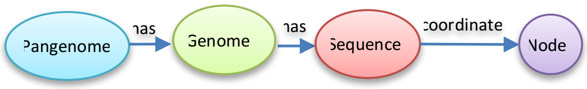
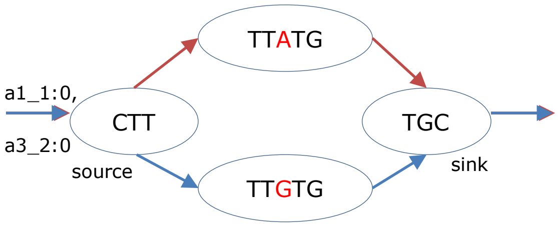


Figure 1 Graph structure in sequence layer



Position	0	1	2	3	4	5	6
Genome1	C	T	T	A	T	G	C
Read2	C	T	T	G	T	G	C

Figure 2 SNP in read that maps in forward direction

2.3 Variant calling

2.3.1 Single Nucleotide Polymorphism

A SNP is defined as a single base variation (A, C, T or G) at the same position between aligned DNA. Each variation is present to some degree within a population. But in our work, we do not take the SNP frequency into account, because as a preliminary work, we want to detect all polymorphic base. We consider all alternative base in reads as a SNP.

2.3.2 Bubble

A SNP is represented by a bubble structure in the graph as shown in Figure 2. A bubble structure is a sub-structure that consists of a source node, a sink node and all nodes that are traversed on the path between that source and sink node. A read with the same sequence as the genome will be mapped to the same node; if at the next location it has a different nucleotide, the node will split and create two nodes. The starting node where the *k*-mer diverge is what we call a source node and the node where the sequences reconverge into a single node is a sink node.

2.3.3 Ne04j

Neo4j is a NoSQL graph database that is open source and highly scalable. It uses a property graph model. Nodes are connected by relationships. Both nodes and relationships can have labels and attributes (key-value-pair). A relationship can also have a direction, a start node and an end node.

There are two primary interfaces for working with Neo4j; Cypher queries and Java API. Cypher is a declarative graph query language (like SQL) that is simple and easy to understand but is relatively slower compared to the Java API. The Java API on the other hand is faster and more flexible to use. The version of Neo4j we used is 3.1.5., the same as the current release of PanTools.

2.3.4 SNP detection algorithm

The algorithm was implemented in Java(TM) SE Runtime Environment (build 1.8.0_45-b14). The program is written as a part of the sequence layer class in PanTools, using Neo4j's Embedded Java API. We developed an algorithm to find the variant by traversing the nodes. We compare it with all variants found in the read mapping to each single genome in the pan-genome.

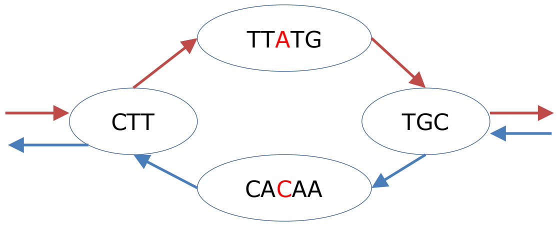
SNP bubbles can be formed by the reads that map in forward direction (Figure 2) and reverse direction (Figure 3). Our SNP detection algorithm basically comprised of finding all potential source node and traverse its child to find the sink node. Potential source node for forward mapping is when the two outgoing edges, one from read and the other from genome, has the same side, eg: F_ and F_ or R_ and R_. For the reverse mapping case, the source candidate has two outgoing edges with different side, eg: F_ & R_. To be precise we created a list of genome-reads pair. For forward mapping, the pairs are [FF-FF, FF-FR, RR-RR, RR-RF]. For reverse mapping, the pairs are [FF-RR, FR-RF, FF-RF, FR-RR]. Since the genome and the read in reverse mapping going to different direction, to make sure that it is a SNP, the start node of the genome incoming edge should be different from the end node of sequence outgoing edge.

After we find the source node we do breadth-first search to find the sink nodes as described in Algorithm 2. Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (in this case, the source node) and explores the neighbouring nodes first, before moving to the next level neighbours. We use a queue to list all nodes to be visited next. After we visit a node, we directly delete it from the queue.

For each child, we check if the node is a sink or not. In forward mapping case, the genome and the reads have the same distance from the source. In the reverse mapping case, we check that the outgoing edge has a genome and compare the distance to the read.

Algorithm 1. Pseudocode of SNP detection algorithm.

```
Data: pan-genome graph produced by construct pnagenome
      algorithm in PanTools. The graph includes one or more
      genomes each containing one or more sequences, and
      one archive containing reads
Result: locations of SNP in each Genome
for node = 1 ... number of sequence node in graph do
    out_or = get_all_outgoing_edge(node);
    check_is_source_forward(edge_pair_forward);
    check_is_source_reverse(edge_pair_reverse);
    if the node is source in reverse direction then
        find_sink_forward(edge_pair_forward);
    elseif the node is sink in backward direction then
        find_sink_reverse(edge_pair_reverse);
    else
        continue
end
```



Genome: CTTATGC
: CTTGTGC (reverse complement of read)
Read : GCACAAG

Figure 3 SNP in read that maps in reverse direction

Algorithm 2. pseudocode of BFS search to find sink

```
Data: edge_pair_forward = two outgoing edges containing reads
      & genome
Result: locations of SNP in each Genome

Initalize queue, visited_node, depth, next_coordinate,

queue <-- children node of edge_pair_forward

from_source_dict <-- {read_edgeId : coordinate}
//dictionary of sequences in source node, with read id and edge
id as a key and coordinate as a value

next_coor = calculateNextCoordinate(coordinate)
next_coordinate <-- { read_edgeId : nextcoor}
//dictionary of next coordinate of the sequence in the node

While not queue.isEmpty() && depth < k graph do
    node = queue.removeFirst()
    for incoming relationship in node do
        for read in relationship do
            if read_location in node - read_location in source
            == genome_location in node – genome_location in
            source
                checkException(read, genome)
            end
            if checkException true
                return snp_list.add {genome: coordinate}
            else
                calculateNextCoordinate(coordinate)
            end
        end
    end
    for outgoing relationship in node do
        for read in relationship do
            if read_coordinate is in next_coordinate do
                child = node.getChild()
                queue.add(child)
            end
        end
    end
end
```

2.3.5 Exceptions in bubble structures

Based on our observations, bubble structure are not always caused by SNPs. For example, in Figure 4 the bubble structure is formed by the genome's repetitive sequence in the ranges [0-7] and [8-16]. The two regions differ by one nucleotide in position 3 and 10 respectively, making the path diverge and reconverge around the polymorphic location. When the read1 (in blue edge path) maps to genome in coordinate [8-16], the read at position 3 is regarded as SNP by the genome in coordinate [0-7]. We prevent that from becoming a false positive by adding a condition: if a read maps to another part of the genome, it is not a SNP, as explained in Algorithm 3.

The function checkException() in Algorithm 3 is called at a node where the incoming edges contains genome and reads that have the same distance. We created a list of all incoming sequences with the same distance. For example in Figure 4. We have [[G1, a, b],[G1, c, d]] and [R1, c, d]. The sink pair therefore is [G1, a, b] and [R1, c, d]. For each of the two sequences (G1 and R1), find the counterpart that has the same edge id for the incoming and the outgoing edge. This is the case for G1 and R1 with edges id c and d. So, the node is not a sink and there is no SNP in that bubble.

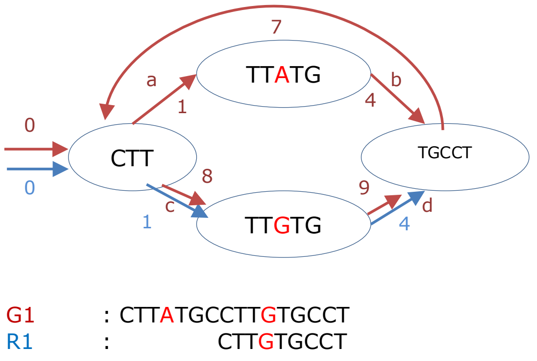


Figure 4. A Bubble structure formed by a repeat with polymorphism. (*k*=3)

```
Algorithm 3. pseudocode of checkException
Data: read_list: [[read, rel_id_out_source, rel_id_in_sink]] and
      genome_list: [genome, rel_id_out_source, rel_id_in_sink]
      that has the same distance from source to sink
Result: boolean, true = if there is no other genome map perfectly
for read, genome in read_list, genome_list do
      if [rel_id_out_source, rel_id_in_sink] is not equal
            edge_pair.add(rel_id_out_source, rel_id_in_sink);
      done
done
for [rel_id_out_source, rel_id_in_sink] in genome_list, read_list
do
      if [rel_id_out_source, rel_id_in_sink] is equal
            return False;
      else
            return True;
      done
end
```

2.3.6 Accuracy assessment

We compared the SNP coordinates found by our algorithm to those found in the indicated way, by mapping reads which we consider the ground truth. We align the reads using Bowtie2 (Langmead *et al.* 2012) version 2.2.6. We used default settings .

We used samtools mpileup (Li Heng *et al.,* 2009) version 0.1.19 to retrieve all information on mapped reads such as matches and mismatches. Mpileup by default only considers a depth up to 10x; to include all reads we used parameters -BQ 0 -d10000000. From the mpileup output, we filtered out SNP location using.

3 RESULTS AND DISCUSSION

3.1 Read mapping can be performed by "the construct pan-genome" algorithm in PanTools

First we set out an experiment to verify that read mapping can be performed by PanTools, we therefore performed read mapping of 20% of the available data (6x fold coverage) on two E.coli (5 Mb) genomes in PanTools. It generated 1.4 million nodes and 6.7 million edges when the reads are added. The exploding number of edges can result in extreme decrease in performance in the variant calling process. Therefore, we performed read mapping and variant calling using subsets of the data with only one genome of E. coli, to see how the performance relates to the data size. We extract read maps from the first 5, 10, 50, 100, 500, 1000kb of the genome and construct pan-genome on that data.

Table 1 shows the properties of the resulting graph as well as the time needed to build the pan-genome graph. We set *k* = 11 to increase the variant, while still keeping the space complexity reasonable. We observed that the number of *k*-mer, nodes, and edges grow in linear fashion. However, the number of nodes grows slower, linearly with larger slope, compared to the other two properties. This is because the nodes in PanTools can store sequences in forward and reverse orientation, making it possible to save a lot of space. The number of edges grows quickly because they store the orientation and the position of each sequence.

Table 1 Scalability of the program to increasing genome length.

Length (K)	k-mers	Nodes	Edges	Time (s)
5	5060	255	731	8
10	10255	639	1786	4
20	20151	1491	4175	6
50	50440	5245	14364	12
100	99133	15015	40502	45
200	189096	44335	121197	59
500	418699	161807	454138	303
1000	688074	379622	1151726	821

3.2 SNPs can be detected by detecting bubbles

Next we wanted to detect SNPs from pang-genome graph produced by PanTools. The resulting graph from read mapping step is used as an input for our algorithm. We tested its performance on the graph resulting from mapping reads to the first 5, 10, 20, 50, 100, 200, 300, 400, and 500kb of genome. We extrapolated the result to estimate how much time needed for algorithm to call SNPs in realistic genome size (Figure 5). We found that the computation time grows exponentially (y = 0.1022x^2 – 15.5x+269), whereas the number of nodes and edges grow in linear fashion (y=1.29x-34.33 and y=1.26x-34.48). We therefore hypothesize that the computation time depends on the degree of the nodes in the pan-genome graph.

Table 2 shows that the degree indeed increases as we add more sequences to the graph. In the graph with genome length smaller than 50 the largest degree is only 10, whereas if we increase the genome length the degree could be 18. The degree almost follows a normal distribution, eventhough we can see some strange occurance. The highest number of degree for every genome length is four or five, indicating that the nodes are mostly can be considered as a source.

Our algorithm starts by finding two edges leaving a node that satisfy the requirement of being a source node. One node can have multiple edge pairs to be considered as a source node. For each node traversed by find sinks function also could have multiple branch. The time complexity of SNP calling algorithm is O(m^d) where d is the number of average degree of the graph.

This performance problem could be avoided by choosing the larger number of *k.* If we construct a pan-genome graph with smaller *k* we would have more nodes, since the *k*-mer is repeated more frequently by chance. This repeated *k*-mer will also creates more edge and increasing the degree in that node. However, this strategy could decrease the number of detected SNP as we will discuss later. We could also make the running time shorter by parallelizing each child of the node but this is still computationally expensive and there could be an overhead time to assign an instance to the CPU.

3.3 The number of SNPs detected depends on *k*

For assessing the accuracy, we used the reads mapped ofnto the first 50kb. We found that the number of SNPs detected increased as we choose smaller *k*-mer as shown in Table 3. Undetected SNP are those SNPs that are located in less than *k* base pairs from the end of sequence (Figure 6) and SNPs within *k* base pairs of each other (Figure 7). The first type of undetected SNP does not have source or sink node, because the sequence does not have the same *k*-mer before the *k*-mer with SNP occurs. The second type of the undetected SNP can only detect the last occurrence of SNP.

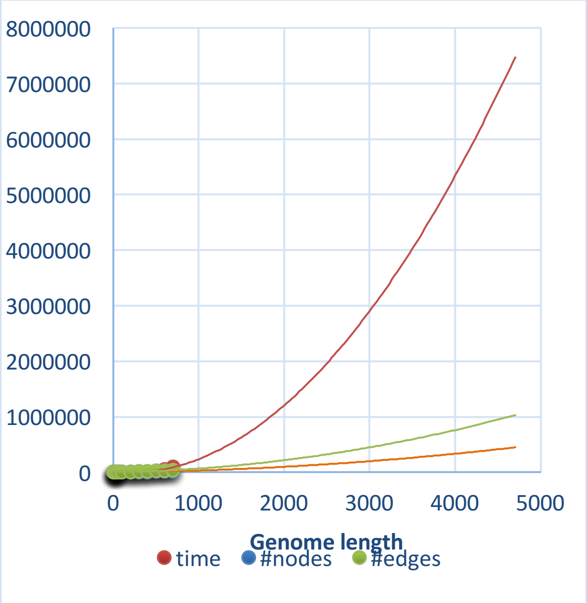


Figure 5 SNP calling algorithm computation time as a function of genome length.

The first type is more abundant because, as shown in Figure 8, the occurrence of the SNPs is higher towards the start and the end of the reads. This can be explained by low reads quality in those region (Figure 9) where the quality is also lower at the end of the reads. Undetected SNPs at the start can be explained by reads that map in reverse direction.

This problem can be minimized by choosing a lower *k* so that the SNP can be detected. But usually the SNP also occurs in the coordinate less than 7 bp and PanTools can not construct a pan-genome with *k* less than 7. Choosing a very low *k* can also lead to a long running time.

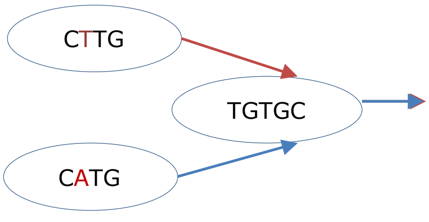
Another way to solve this problem is by modifying the algorithm to do base comparison in every start of the read and the end of the read, but it is not efficient and can adds a running time. It is also not effective because the main idea of variant calling in PanTools is to mine the variation from the graph structure.

Table 2 Degree distribution of the pan-genome graph as function of genome length.

	Genome length							
Degree	10	20	50	100	200	300	400	500
1	0	0	0	0	0	0	0	0
2	44	15	92	1214	3337	6552	0	13502
3	15	5	29	175	307	469	0	791
4	187	76	466	5903	18525	33073	20722	65584
5	233	93	516	3223	6908	10076	13300	16428
6	115	51	252	1687	4975	9280	14413	19996
7	8	6	21	252	853	1602	2567	3661
8	34	8	100	2165	7493	13995	20927	29412
9	3	1	12	249	984	1927	3079	4598
10			3	93	598	1455	2588	4399
11				19	112	278	509	966
12				27	175	477	890	1664
13				6	31	111	201	395
14				1	25	73	138	281
15				1	6	10	31	67
16					4	9	19	43
17					2	7	8	17
18						1	3	3

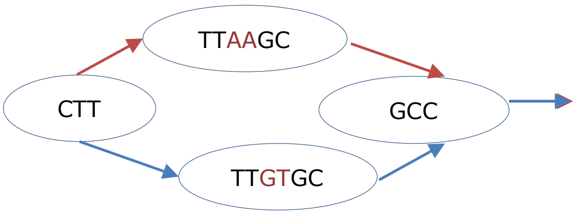
Table 3 The accuracy of the variant calling algorithm at different values of *k*.

K	Detected	Undetected
11	113	45
10	118	40
9	124	34
8	128	30



Position	0	1	2	3	4	5	6
Genome1	C	A	T	G	T	G	C
Read	C	T	T	G	T	G	C

Figure 6. A SNP closer than *k* base pairs from the start of sequence. *k*=3



Position	0	1	2	3	4	5	6	7
Genome1	C	T	T	A	A	G	C	C
Read	C	T	T	G	T	G	C	C

Figure 7. Two SNPs with a distance shorter than k. (here *k*=3)

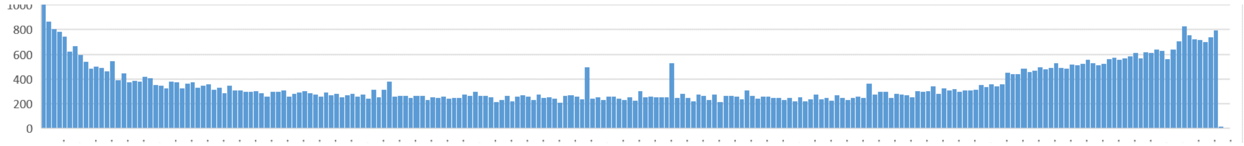
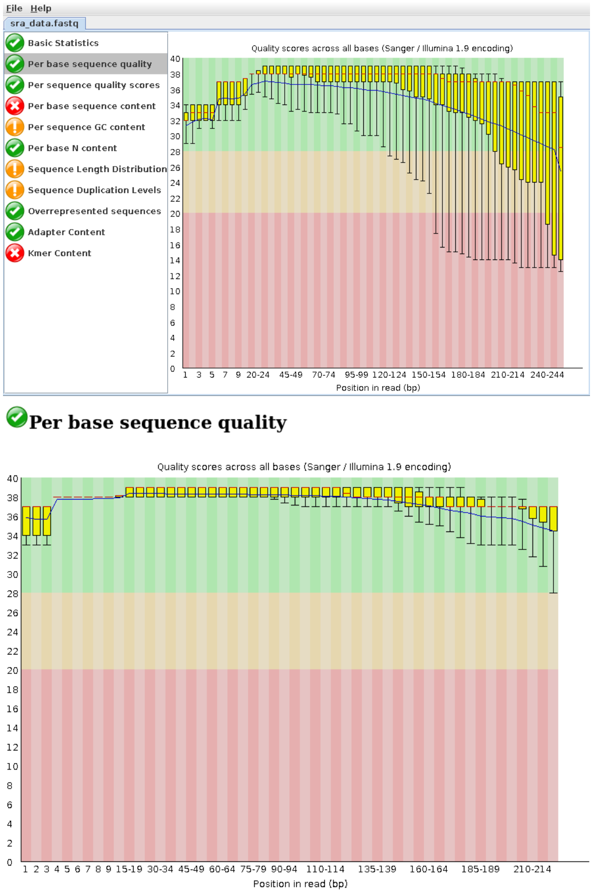


Figure 5 SNP distribution accross the read


File Help
sra_data.fastq
Basic Statistics
Per base sequence quality
Per sequence quality scores
Per base sequence content
Per sequence GC content
Per base N content
Sequence Length Distribution
Sequence Duplication Levels
Overrepresented sequences
Adapter Content
Kmer Content
Quality scores across all bases (Sanger / Illumina 1.9 encoding)
Position in read (bp)

Per base sequence quality

Quality scores across all bases (Sanger / Illumina 1.9 encoding)
Position in read (bp)

Figure 6. Reads quality before and after trimming

CONCLUSION

Detecting variants is important for genetic studies. Applications range from detecting SNPs underlying certain diseases in human, to finding variants that make a plant resistant to certain pests. Variant detection is mostly performed by aligning a sequence sample produced by NGS technology to a reference genome and calling the variants afterwards. The advance of NGS has made the re-sequencing became easier thus produce many sequenced genome that can be used as a reference. It is beneficial to do read mapping to the collection of sequenced genome but it is also computationally inefficient. We tackle this problem by doing mapping reads to a pan-genome, where it represents a set of genome sequences.

Read mapping to a pan-genome in PanTools is possible and it has the same behaviour as adding a genome but if we include all reads with coverage higher than 10x, it will become slow because the space required will explode.

Variant calling can be performed directly in the pan-genome graph using our algorithm. It has around 70% accuracy because it cannot detect SNPs closer together than *k* bp and SNPs located less than *k bp* from the start or the end of reads. This algorithm is still not efficient because of its traversal strategy. This performance prolem can be solved by carefully choosing the *k*. The lower k can improve accuracy but the larger *k* can improve the running time.

Data downloaded from

https://www.ncbi.nlm.nih.gov/nuccore/NZ_CP015832.1?report=fasta
https://www.ncbi.nlm.nih.gov/sra/?term=SRX3316781

Reference

1. Clark RM, Schweikert G, Toomajian C, Ossowski S, Zeller G, Shinn P, Warthmann N, Hu TT, Fu G, Hinds DA, Chen H, Frazer KA, Huson DH, Schölkopf B, Nordborg M, Rätsch G, Ecker JR, Weigel D: Com- mon sequence polymorphisms shaping genetic diversity in Arabidopsis thaliana . Science 2007, 317:338-342.
2. Marschall et al. (2016). Computational pan-genomics: status, promises and challenges. Briefings in Bioinformatics, (May), http://doi.org/10.1093/bib/bbw089
3. Sheikhizadeh, S., Schranz, M. E., Akdel, M., De Ridder, D., & Smit, S. (2016). PanTools: Representation, storage and exploration of pan-genomic data. Bioinformatics, 32(17), i487–i493.
4. Schneeberger, K., Hagmann, J., Ossowski, S., Warthmann, N., Gesing, S., Kohlbacher, O., & Weigel, D. (2009). Simultaneous alignment of short reads against multiple genomes. Genome Biology, 10(9), R98.
5. Langmead, B., Trapnell, C., Pop, M., & Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of

short DNA sequences to the human genome. Genome Biology, 10(3), R25. http://doi.org/10.1186/gb-2009-10-3-r25

6. Limasset, A., Cazaux, B., Rivals, ., & Peterlongo, P. (2015). Read Mapping on de Bruijn graph. BMC Bioinformatics, 1–12. http://doi.org/10.1186/s12859-016-1103-9
7. Joshi, N. A., & Fass, J. N. (2011). Sickle: A sliding-window, adaptive, quality-based trimming tool for FastQ files (Version 1.33)[Software].
8. Iqbal, Z., et al. (2012) De novo assembly and genotyping of variants using colored de Bruijn graphs,Nature genetics,44 (2), 226-232
9. Younsi, R. and D. Maclean (2014) Using 2k+ 2 bubble searches to find Single Nucleotide Polymorphisms in k-mer graphs,Bioinformatics,31 (5), 642-646
10. Galdon SC. (2015) Mining Structural Variation in Pan-genome Graphs. [Thesis]
11. Langmead B, Salzberg S. Fast gapped-read alignment with Bowtie 2. Nature Methods. 2012, 9:357-359.
12. LI, Heng, et al. The sequence alignment/map format and SAMtools. Bioinformatics, 2009, 25.16: 2078-2079.