Computational Genomics
Project 3 - Report

Agata Kaczmarek, Władysław Olejnik and Mateusz Stączek

June 7, 2024

1 Introduction

Structural variants (SVs) are large genomic alterations that can significantly impact gene function and phenotype. They include deletions, insertions, duplications, inversions, and translocations of DNA segments. Detecting SVs is crucial for understanding genetic diseases and evolutionary biology and for applications in personalized medicine.

With the advent of high-throughput sequencing technologies, detecting SVs comprehensively and efficiently has become feasible. When aligned to a reference genome, short-read sequencing data can reveal discrepancies indicative of SVs. However, the accurate identification of these variants remains challenging due to the genome's complexity and sequencing technologies' limitations.

This project aims to develop a computational pipeline to detect structural variants from short-read data alignment files against the GRCh38 reference genome. The input data is provided in SAM/BAM format, and the output will be a list of sorted structural variants in VCF format. The detection method focuses on analyzing read depth (RD) to identify regions with significant deviations, indicative of deletions or insertions.

The Methodology section details the steps and algorithms used for data processing, read depth calculation, smoothing, variant detection, and merging of adjacent variants. The Results section presents the outcomes of the algorithm applied to test datasets, including performance metrics and visualizations. Finally, the Conclusions section summarizes the findings, discusses the implications, and suggests potential improvements for future work.

2 Methodology

2.1 Algorithm implementation

The implemented algorithm for detecting structural variants in short-read data alignment files against the GrCH38 reference genome consists of several key steps. These include calculating read depth, smoothing the signal, detecting variants, merging adjacent variants, and outputting the results in VCF format. The following details each step of the process:

Calculate Read Depth The read depth is calculated by processing the BAM file using the `pysam` library. The genome is divided into bins of specified width, and the number of reads mapping to each bin is aggregated. To improve memory efficiency and results accuracy, loci with a read depth of zero are skipped. The function `calculate_read_depth` performs this task and returns a dictionary of read depths for each bin.

Smooth Signal To reduce noise in the read depth signal, a smoothing function is applied. The `smooth_signal` function uses a one-dimensional Gaussian kernel to smooth the read depths. This results in a more stable signal for subsequent variant detection.

Find Cutoff Thresholds The thresholds for detecting deletions and insertions are determined by analyzing the distribution of the read depth values. The `find_cutoff` function calculates the mean and standard deviation of the log-transformed non-zero read depths. These values are used to define exponential thresholds for deletions and insertions.

Detect Variants Using the smoothed read depth data and the calculated thresholds, the `detect_variants` function identifies potential deletions and insertions. Bins with read depths below the deletion threshold are marked as deletions, while bins with read depths above the insertion threshold are marked as insertions.

Merge Variants Adjacent variants of the same type are merged to form longer contiguous variants. The `merge_variants` function iterates through the list of detected variants and merges adjacent ones if they are of the same type and consecutive.

Create VCF The final step involves creating a VCF file to store the detected variants. The `create_vcf` function formats the merged variants into the standard VCF format, including necessary metadata and variant information.

Main Function The `main` function orchestrates the entire process. It calls the functions for calculating read depth, smoothing the signal, detecting and merging variants, and creating the VCF file. Parameters such as the bin width and smoothing window size can be adjusted to optimize the algorithm for different datasets.

2.2 Metrics

To measure the performance of our implementation of the read depth algorithm we decided to implement and use the Intersection over Union (IoU) metric.

We compute IoU for a pair of VCF files in the following manner. First, we filter both VCF files by the SV type to compute the IoU score for deletions and insertions separately. Then, we create a long vector of zeros representing genome positions. For every row in the first VCF file, we add 1 to every position that is between the start and end of this SV and repeat the procedure for the second VCF file. Lastly, we divide the count of two-s by the count of non-zero elements. This is equivalent to a fraction, where the nominator is the number of positions that belong to some SV according to both VCF files and the denominator is the number of positions that belong to some SV in at least one VCF file. We returned 0 in case the denominator was also 0. The pseudocode snippet can be found below.

```
tot_counts = {
    'both': 0,
    'one': 0
}

for chrom in chromosomes:
    np_array = np.zeros(big_number_greater_than_max_end)

    # count common SV positions in an array, with SV being from POS to END
    for _, row in df1_chrom.iterrows():
        np_array[row['POS']:row['END']] += 1
    for _, row in df2_chrom.iterrows():
        np_array[row['POS']:row['END']] += 1

    # Intersection Over Union metric = count of twos / count of ones and twos
    count_of_ones = np.sum(np_array == 1)
    count_of_twos = np.sum(np_array == 2)
    tot_counts['one'] += count_of_ones
    tot_counts['both'] += count_of_twos
```

```
if tot_counts['one'] + tot_counts['both'] > 0:
    iou = tot_counts['both'] / (tot_counts['one'] + tot_counts['both'])
else:
    iou = 0
```

3 Results

We later compared the results we achieved with the ones we got from already published tools. Those tools were delly [1] and CNVpytor [2].

We run all three methods on the same two files: GRCh38_full_analysis_set_plus_decoy_hla.fa and SRR_final_sorted.bam. For delly we got results containing only 5 records with deletions and no insertions. From CNVpytor we got 28267 results, which were either deletions or duplicates (for our comparison we treated all duplicates as insertions). For our method, we decided to compare two versions: one had 21466 records and the other 217571 records. The distribution of deletions and insertions for the smaller version can be seen in Figure 1.

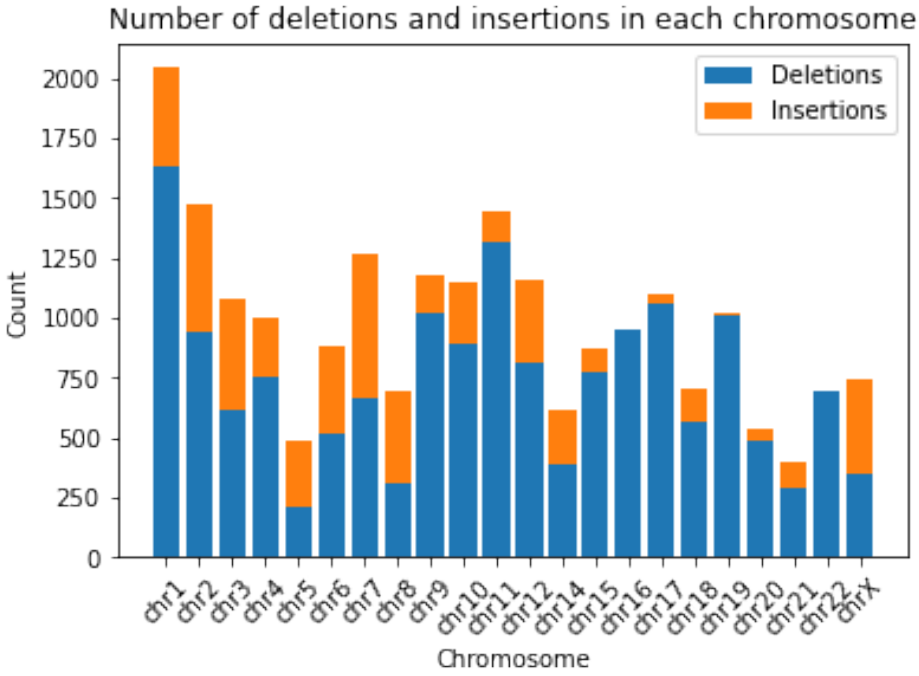

Figure 1: Number of deletions and insertions in each chromosome for smaller results of our method.

```
print('IOU for deletions:', compare_2_dfs_ALT_DEL(vcf_delly, vcf_ours))
print('IOU for insertions:', compare_2_dfs_ALT_INS(vcf_delly, vcf_ours))
```

```
IOU for deletions: 4.09294181082423e-05
IOU for insertions: 0
```

Figure 2: Results for comparison between delly and ours larger results.

As we can see in Figures 2, 3, 4, the highest IoU score for deletions achieved the combination between CNVpytor results and ours smaller version. It was also the combination that achieved the

[1] https://github.com/dellytools/delly
[2] https://github.com/abyzovlab/CNVpytor/tree/master

```
print('IOU for deletions:', compare_2_dfs_ALT_DEL(vcf_delly, vcf_ours))
print('IOU for insertions:', compare_2_dfs_ALT_INS(vcf_delly, vcf_ours))
```
✓ 18.0s

```
IOU for deletions: 0.0
IOU for insertions: 0
```

Figure 3: Results for comparison between delly and ours smaller results.

```
print('IOU for deletions:', compare_2_dfs_ALT_DEL(df1_cnv, vcf_ours))
print('IOU for duplicates (=insertions):', compare_2_dfs_ALT_INS(df1_cnv, vcf_ours))
```
✓ 1m 4.9s

```
IOU for deletions: 0.0041476278976378105
IOU for duplicates (=insertions): 0.028192146113593216
```

Figure 4: Results for comparison between CNVpytor and ours smaller results.

highest score for insertions. In the second case, it was not possible that a comparison with delly would have a higher score than 0 because it had 0 rows with insertions. The best score when it comes to comparison with delly achieved our larger version. Even the best from those results are not great.

We believe that this is caused by several factors:

- our approach is a naive ReadDepth - the easiest with the highest number of various assumptions,
- delly method uses another, more sophisticated approach to find SVs,
- CNVpytor also implements ReadDepth, but probably with other assumptions, and as we can see from our results, small changes in its parameters can change the results a lot,
- in our approach we mainly manipulated with two factors: `cuf-off` - which results we treated as normal and which were below or above it, and `smooth_signal` - parameter related to reducing noise in the read depth signal.

4 Conclusions

In our project, we implemented an algorithm for detecting structural variants in short-read data alignment files against the GrCH38 reference genome. We also implemented the IoU metric to check the results of our method against others already published. The results we got are interesting and show the influence of various parameters and methods on the final results. This field requires further work on understanding the exact influence of separate parameters on the results.