Bilkent University
Department of Computer Engineering

Generating SUNK and SUN from the reference genome

Farnush Farhadi

Date Performed: July 2014 - Sept. 2014
Supervisor: Professor C. Alkan

1 Objective

For sequencing studies, we need to discover the differences between highly similar duplicated sequences. We generate SUNK and SUN (as defined in 1.1) from the reference genome.

1.1 Definitions

SUNK **S**ingly **U**nique **N**ucleotide **k**-mers are a class of k-mers that uniquely tag a specific paralog.

SUN **S**ingly **U**nique **N**ucleotides are the nucleotides that uniquely discriminate paralogs of highly identical segmental duplications. Actually, SUNs are a distinct class of paralogous sequence variants that uniquely tag a specific paralog.

By *paralog* we already include the k-mers that are seen in alternative haplotypes. If two parts of the genome, A and B are very similar (99% similar), then we use the SUNs and SUNKs to be able to differentiate them. Sometimes there can be additional copies, like 2A and 3B; but without the SUNs we would just say there are total 5 copies of A and/or B. With SUNs, we can say there are 2 As and 3 Bs.

2 Generate SUNKs

2.1 Generating initial set of SUNKs

This set includes kmers (with k=30 for human) which are unique through whole original chromosomes of reference genome (human genome with hg19 assembly) except alternative haplotypes.

Note 1 As repetitive DNA sequences comprises more than 50% of the human genome, we masked the reference genome (results in **hg19_masked** file previously saved in **/mnt/storage1/projects/sunk/hg19_masked**) to get rid of already-known repetitive parts.

Note 2 hg19_masked file consists of below sequence names:

a. chr#

b. chr#_***_random

c. chrUn_***

d. chr#_***_hap

We behaved the first 3 groups as original chromosomes and the last one as alternative haplotypes. There are around 60 different chromosome names including all 4 groups.

Note 3 If a k-mer from *chri* is unique through original chromosomes and also within each alternative haplotypes of *chri* (*chri_***_hapj*), it is still globally unique.

Algorithm for finding globally unique k-mers consists of 4 steps:

a. Generate unique kmers within each chromosome (from all 4 name groups) and write them in file.
For this part, we defined a header named **Kmer.h** which has two variables:

   (a) Kmer's **sequenece** which is a k-length long substring of the chromosome.

   (b) Kmer's **flag** telling us whether this kmer was locally unique within chromosome till now or not.

   We construct a set (using *set* class of c++ STL) of Kmer.h objects for each chromosome. We insert kmers to the set with a unique flag if they were not seen before. For not unique kmers, we just update their flag to zero.
   After scanning chromosome is finished, we write all kmers to a fasta file named same as chromosome name.

   - Output file format:

(a) Unique kmers:
$> Chri\_name|S(i)|E(i)|uniqueFlag|IndexInFile$
$KmerSequence$
I.e., following kmer shows that it comes from position 10 to 29 (0-based position) of chromosome 1 and it is the 9th kmer in the related file:
$> Chr1|10|29|1|9$
$ACGTACGTACGTACGTACGTACGTACGTAC$

(b) Not unique kmer:
$> Chr|0|0|0|IndexInFile$
$KmerSequence$

Note 4 As the cost of inserting, deleting and finding an elemnt is $\log n$ in set, using *set* structure is the most efficient method to generate a alphabetic sorted (based on sequence) kmers.

b. Merge kmers coming from original chromosomes together that is resulting in **finalOriginal**.
As we keep kmers of each chromosome with their alphabetic order in the *set*, we can merge kmers by using merge strategy of *Merge Sort*. The *unique flag* of kmer is updated like following table (0,1 and NA show the kmer is not-unique, unique and not exist in file, respectively) (Note that wehen a kmer has a NA status in Chr2, it means that there is a kmer in Chr1 which is not exists in Chr2):

Chromosomes to merge		
Chr1	Chr2	Merge Result
0	0/1/NA	0
0/1/NA	0	0
NA	1	1
1	NA	1
1	1	0

If a kmer is not unique through one chromosome, it is not globally unique, neither.
If a kmer is unique through one chromosome and not exists in another one, it is still unique after merging two chromosomes.
If a kmer is locally unique through both chromosomes, it is not globally unique.

Note 5 Not unique kmers are written with the following format: (Note that we need to keep them for further mergings)
$> Chr23|0|0|0|IndexInFile$
$KmerSequence$
This format is same for part (c) and (d), too.

Note 6 As the set of globally unique kmers within original chromosomes is huge, we can't keep it in program memory. Accordingly, we work on each chromosome and write their unique kmers in seperate files and then merge all chromosomes together like merge sort strategy by. I.e., if there are 5 files like 1,2,3,4 and 5, merging is done like this:

- 12
  34
  5
- 1234
  5
- 12345

The result of merging all original chromosomes together is written in **finalOriginal**.

c. Merge alternative haplotypes of one chromosome together.
At first, we find all chromosomes which their alternative haplotypes are available. Then we merge only all alternative haplotypes of each chromosome like following table:

Haplotypes to merge		
Chr1_hap1	Chr1_hap2	Merge Result
0	0/1/NA	0
0/1/NA	0	0
NA	1	1
1	NA	1
1	1	1

The difference from part (b) is if a kmer is locally unique within both haplotypes of chromosome, it is still unique after merging two haploypes of chromosome.

Note 7 For the last row of the table, we need to update start/end position of kmer. Actually, the kmer follows bellow format after being updated:
$> Chri\_hapj, Chri\_hapk, ...|S(j), S(k), ...|E(j), E(k), ...|1|IndexInFile$
$KmerSequence$

Which means that the kmer comes from the position *S(j)* to *E(j)* of alternative haplotype *Chri_hapj* and *S(k)* to *E(k)* of alternative haplotype *Chri_hapk* and so on.
In this part, we have output files such as *final_chri_hap, final_chrj_hap, ...* .

d. Merge *chromosome-with-hap* (i.e., chromosomes which their alternative haplotypes exist) together that resulting in **PutativeSUNK**.
We merge the outputs of the previous step together following the rules of table of part (b). Note that if a kmer is unique within both merged_haplotypes of two chromosomes, it is not globally unique since it is repetitive in two different chromosomes.

e. Merge finalHap and finalOriginal resulting in **putativeSunk**.
Now, we have locally unique kmers within original chromosomes and also all haplotypes of any chromosomes. We merge them such as following table:

Kmer_files to merge		
finalHap	finalOriginal	Merge Result
0	0/1/NA	0
0/1/NA	0	0
NA	1	1
1	NA	1
1	1	1 *if same chromosome*

If a kmer is unique through both files, it is globally uniqe if its chromosome in *finalHap* file is as same as the one in *finalOriginal* file. Then we update its vector of start/end positions.

The final fasta format of putative SUNK is:
$> Chri\_name|S(i)|E(i)|IndexInFile$
$KmerSequence$
Which Chri_name, S(i), E(i) are a set of probably more than one.

The implementation is done by c++ and all source code of this part is in **putative SUNK** folder and it can be run by using following commnads. Comments through functions in .h files can be helpful.

```
g++ main-finalKmers.cpp Kmer.cpp uniqueKmerFinder.cpp -o p_sunk
./p_sunk K genome_address
```

Note that K is kmer length and genome_address is the address of reference genome which is better to be masked. The result of this step will be saved in

putative- SUNK/res/putativeSunk directory. We don't save any of repetitive kmers in this part since we don't need them any more. A sample test is provided in **putative-SUNK/sample test** directory with $k = 4$ and *genome.txt* as the arguments.

2.2 Map putative SUNKs back to reference genome

In this part, we map the putative SUNKs back to reference genome for further filtering. The mapping is done by **mrFast** tool with edit distance (e) 0, 1 and 2. For the sake of time, we map the kmers (which are counted as the reads by mrFast) with 8 parallel processes. Actually we partition **putativeSunk** to 8 files (putativeSunk-1.fasta to putativeSunk-8.fasta) runnig **main-partition.cpp**:

```
g++ main-partition.cpp -o part.out
./part.out putativeSunk_address 8
```

After ruunig it, we have 8 files in the same directory. Then, we map each of them in parallel with this command:

```
cat mrfast.sh | parallel -j 8
```

mrFast.sh is available in the attached files. It consists of 8 seperate lines, each of them mapping a different file in reference. Now, mrFast mapping results are like **out1** to **out8** in **mrFast_res** directory which is specified in mrfast.sh. For assuring the accuracy of putative SUNKs, we request mrFast to report us unmapped reads, too. We see there is no unmapped reads.

The filtering criteria is based on e:

- $e = 0$
  We remove those kmers that are mapped more than once (forward or reverse).
- $e = 1$ **or** $e = 2$
  We remove those kmers that are mapped more than 500 times (forward or reverse).

Note 8 Note that mrFast result is in .sam format thus a read is mapped both in forward and reverse complemented strand.

For a summary of how we perform filtering, first we define **readInSamFile.h**. Accordingly, a kmer has 3 attributes, *sequence*, *name* and *Count* that the last one shows how many times the kmer is mapped in reference. For simplifying, we suppose that we are working only on *out1*. We keep mrfast results for each chromosome (which the kmers are mapped in) in seperate files and then we merge kmers mapped in different chromosomes together which is resulting in **reads-count-1.fasta**. Note that kmers are sorted by their sequence and the *count* attribute in their names is updated through merging. At the end, it is

clear that each kmer has been mapped how many times in reference. We repeat this for other *out*s to complete the jop.
After merging, we have files **reads-count-1.fasta** to **reads-count-8.fasta** (in **Analyze-mrFast/e0** directory for edit distance 0 and in **Analyze-mrFast/e1-2** directory for edit distance 1 or 2) which have following format:

$>Chri\_name|S(i)|E(i)|Count$
$KmerSequence$
Which Chri_name, S(i), E(i) can be a set of more than one element and the kmer is mapped *count* times in reference.

Filtering criteria in more details:

- $e = 0$
  We remove the kmer if it's *Count* doesn't match the number of elements in *Chri_name*. So we consider alternavie haplotypes.
  For two following examples, the first kmer passes the filter but the second one will be removed from our accepted set of kmers:
  $>Chr1, Chr1\_hap1|1, 2|30, 31|4$
  $AAATTTTTTTTTTTTCAATTGCGTAAAAAA$

  $>Chr1, Chr1\_hap1, Chr1\_hap2|1, 2, 3|30, 31, 32|3$
  $AAACGTCGACGTGCCCAATTGCGTAAAAAA$

  The kmer's *Count* in first example shows that this kmer has been mapped twice in backward strand.

- $e = 1$ **or** $e = 2$
  We remove the kmer if it's *Count* is greater than *limit* which is 500 here.

We run following commands to filter mrFast results based on e.

```
g++ main-filterBaseOnE.cpp readInSamFile.cpp
filterBaseOnE.cpp -o filter.out
./filter.out e add_ref add_mrfast_res mrfast_res_num limit
```

add_ref and *add_mrfast_res* are the address of reference that putativeSunk-1.fasta to putativeSunk-8.fasta are mapped in and the address of mrfast results directory (which has out1 to out8 in it), respectively (*add_ref* is needed because the order of reference chromosome names are same as the reporting order of mrfast in sam file). Note that mrfast results should be exactly like out1 to out8. *e* is edit distance (so we run the code twice, one by e=0 and the other one by e=1 for edit distance 1 or 2). *mrfast_res_num* is 8 here. *limit* is the threshold for filtering based on e=1 which is 500 here.
Implementations are in **Analyze-mrFast** directory. readInSamFile.h is for getting a read (Kmer) information including *read count* which is initially 1, from

a sam file produced by mrFast. filterBaseOnE.h is for filtering and commnets through functions can be helpful. A sample test is provided in **Analyze-mrFast/sample test** directory with mrfast_res_num=2 and limit=2.
After filtering, **reads-count-1.fasta** is partitioned to **reads-count-1OK.fasta** and **reads-count-1OK.fasta** which contains kmers passed this filter and **reads-count-1OK.fasta** containing kmers failed this filter.

Note 9 The maximum *Count* is around 200 for human genome with inputs (reference, K and etc.) satisfying our strategy conditions, so all the kmers pass this filter for edit distance 1 or 2 and the final filtering is based on edit distance 0. Generally, a kmer passes based-on-e-filtering if and only if, it passes both e=0 and e=1/2 filterings.

Now, we reassemble **reads-count-1OK.fasta** to **reads-count-8OK.fasta** in **Analyze-mrFast/e0/** together to generate our **semi-final-sunk.fasta**. We can do that through following command:

```
g++ main-assemble.cpp -o assemble
./assemble reads_count_add semi_final_sunk_add number_of_files
```

reads_count_add and *semi_final_sunk_add* are the directory addresses of **reads-count-1OK.fasta** to **reads-count-8OK.fasta** and assmbled file, respectively. *number_of_files* is the number of input files to be assemble. So now, we have the **semi_final_sunk_add/semi_final_sunk.fasta**.

2.3 Generate final SUNKs

In this part, we count the number of distinct 15-mers in the reference. Actually we creat a hash table for 15-mers, then we walk along the reference and increment the *count* for each 15-mer in the hash table. After that, we sum the frequencies of all 15-mers that compose each previously filtered SUNK (kmers which are kept in files **reads-count-1OK.fasta** to **reads-count-8OK.fasta** in *add_mrfast_res*/e0).

- **Generate 15-mers**
  For this part, we defined a header named **Kmer15.h** which has two variables:

  a. Kmer's **sequence** which is a k-length long substring of the chromosome.

  b. Kmer's **count** telling us how many times this kmer is seen in reference genome.

  We construct a set (using *set* class of c++ STL) of Kmer15.h objects for each chromosome. If a kmer is already exists in the set, we increment the *count* variable through inserting.
  After scanning chromosome is finished, we write each chromosome kmers to a fasta file named same as chromosome name with the following format:

$> kmer\_count|IndexInFile$
$KmerSequence$

Then, we merge all chromosomes together. The *count* of kmer is updated like following table (C is the kmer *count* and NA shows the kmer is not exists in Chr, respectively):

Chromosomes to merge		
Chr1	Chr2	Merge Result
C_1	NA	C_1
NA	C_2	C_2
C_1	C_2	$C_1 + C_2$

Note that we don't care for unique 15-mers in this level.
Source codes of this part is in **generate-15mers** directory and it can be run by using following commnads. Comments through functions in .h files can be helpful.

```
g++ main-15mer.cpp Kmer15.cpp mersFinder.cpp -o 15mers
./15mers address_genome K
```

address_genome is the address of reference which the putative sunks are made of and K is the k length for kmers. The result is saved in **generate-15mers/res/final15Mers.fasta**. A sample test is provided in **generate-15mers/sample test** with $K = 5$ and *genome*. Comments through .h files can be helpful.

- **Calculate each sunk frequencies based on 15-mers Count**
  Now, we calculate a score for each of kmers in **semi_final_sunk.fasta**. We walk along each kmer and extract the sliding-by-one-nucleotide 15-mers composing kmer. We search for these 15-mers in **final15Mers.fasta** and calculate the kmer's score by summing 15-mers' *count*s.

  Note 10 For the sake of time, we construct a set of 15-mers to find each 15-mer in $\mathcal{O}\log(n)$ time instead of a $\mathcal{O}(n)$ search in file. Moving beyond the time, for the sake of memory, We construct a STL set of **mersHash.h** objects from **final15Mers.fasta**. Actually we hash the 15-length strings (15-mers) to 30-length boolian arrays, using 2 bits per nucleotide such as following table:

Nucleotide	Hash value
A	00
C	01
G	10
T	11

So now, we consume almost $4 \times n$ Bytes (or $n \times 30$ bits) of memory instead of $8 \times n$ Bytes (or $n \times 8 \times 8$ bits) for 15-mers sequence, where n is the number of 15-mers.

We generate the **final SUNKs** by following commands:

```
g++ main.cpp mersHash.cpp finalSunk.cpp -o finalsunk
./finalsunk 15mers_add semi_final_sunk_add k threshold
```

15mers_add and *semi_final_sunk_add* are the addresses of **final15Mers.fasta** and **semi_final_sunk.fasta**, respectively. k and threshold are the length of this hashed kmers which the putative sunks are made of and the maximum accepted score, respectively (15 and 500 here). Source codes are in **final-SUNK** directory and the output result is saved in **final-SUNK/res/final_sunk.fasta**. A sample test is provided in **final-SUNK/sample test** with $threshold = 5$ and $K = 5$, using *semi_final_sunk* and *final15Mers.fasta* in the same directory. Comments through .h files can be helpful.

Final SUNKs have following format:

$>Chri\_name|S(i)|E(i)|Count|Score$
$KmerSequence$

Which *Count* says this kmer has been mapped how many times in reference and *Score* is the sum of 15-mers' counts. So now, we have the final list of **SUNK**s.

3 Calculate SUNs

3.1 Calculate genomic coordinates

For each segmental duplication pairwise alignmnet (from WGAC), we find genomic coodinates for mismatchs and indels between segmental duplication pairs. The **GRCh37GenomicSuperDup.tab** file contains the alignments information including which locations are paired with each other, and which file contains the necessary alignment. So we read these information and go to realted files and report mismatches and indels such as following examples:

- Chr1 is aligned to Chr2 in forward (+) strand. Suppose that both starting locations are 1 and the following figure shows only the first 45 bases of alignment.

```
chr1    TGGA---TAAACAACACCTCGTTTACCAACCAATG
1       ||||   ||||**||||    ||||****|||*||
chr2    TGGAAATTAAAGTACAC----TTTATTTTCCACTG
```

Figure 1: Chr1 from 1 to 1000 is aligned to Chr2 from 1 to 1000.

We report mismatches, deletions and insertions such as following tables:

- Mismatches:

Chromosome	Start	End
chr1	9	11
chr2	12	14
chr1	23	27
chr2	22	26
chr1	30	31
chr2	29	30

- Deletions

Chromosome	Start	End	gap numbers
chr1	4	5	3
chr2	17	18	4

- Insertions

Chromosome	Start	End
chr2	5	8
chr1	15	19

- Suppose figure one but chr1 is aligned to chr2 with backward (-) strand. It means that $TGGA$ and $AATG$ are the first and last chr1 nucleotides, respectively. But for chr2, $TGGA$ and $ACTG$ are the last and first nucleotides, respectively. So, we report genomic coordinates such as following tables:

- Mismatches:

Chromosome	Start	End
chr1	9	11
chr2	988	990
chr1	23	27
chr2	976	980
chr1	30	31
chr2	972	973

– Deletions

Chromosome	Start	End	gap numbers
chr1	4	5	3
chr2	983	984	4

– Insertions

Chromosome	Start	End
chr2	994	997
chr1	15	19

Note 11 The nucleotides which are locationg before and after the gaps, have 1 base difference in numbers. Below table shows the nucleotide numbers of chromosome 1 (same for both alignments):

T	G	G	A	-	-	-	T	A	A
1	2	3	4	-	-	-	5	6	7

We can report all segmental duplication alignment pairs through following commands:

```
g++ main.cpp findSDalignInf.cpp -o coordinates
./coordinates GRCh37GenomicSuperDup.tab_add align_both_dir_add
```

GRCh37GenomicSuperDup.tab_add and *align_both_dir_add* are the addresses of **GRCh37GenomicSuperDup.tab** and segmental duplication alignment files (align_both directory here), respectively. Source codes are in **SD-alignments** directory and the output result is saved in **SD-alignments/res/** directory. A sample test is provided in **SD-alignments/sample test** directory which is the result of processing only first four lines of GRCh37GenomicSuperDup.tab file.

3.2 Intersect SD with SUNKs

After finding genomic coodinates, we intersect differences between segmental duplications with SUNKs to get **SUN**s. We intersect genomic coordinates (mismatches, insertions and deletions) with our previously generated **SUNK**s using intersect commands.

Thanks to Prof. **Can Alkan**.