A Computation Note for Assembling Plasmodium 3D7 with CLEAR, Part I

Jason Chin
Pacific Biosciences

June 4, 2013

1 A Computation Note for Assembling Plasmodium 3D7 with CLEAR, Part I

1.1 What Do You Need To Reproduce The Assembly Shown Here

- data: `pread.fa` and `pr_pr_strigent.m4`
- `python` 2.7 / `IPython` 0.13.2
- `pbcore` from `https://github.com/PacificBiosciences/pbcore`
- optional: `summarizeAssembly.py` from `PBJelly_12.7.25` installed in `~/bin/PBJelly_12.7.25/`
- optional: `nucmer` and `mummerplot` from `mummer3` (`http://mummer.sourceforge.net/`)
- optional data: reference `PlasmoDB-9.2_Pfalciparum3D7_Genome.fasta` from `PlasmoDB` (`http://plasmodb.org/plasmo/`)

1.2 Introduction

This is a brief note and code to show how to assemble Plasmodium 3D7 (http://www.ncbi.nlm.nih.gov/Taxonomy/Browser/wwwtax.cgi?id=36329) genome with PacBio(R) pre-assembled reads by a "Consistent Long-read Evidence Assembling pRocess (CLEAR)".

1.3 Why Do We Sequence Plasmodium 3D7 for This Assembly Example?

Plasmodium is a parasite that causes malaria. Understanding its genetics will help to find a cure to the disease. From the sequencing technology point of view, it posts a great challenge to sequence and assembly the genome. Due to its very in-balanced AT/GC content (AT ~= 80% and GC ~=20%), most sequence technology can not produce good and long sequences that enables assembling the genome into long contigs. For example, the earlier publication using 2nd geneneration sequence technology can only get contig N50 about 1 to 4 kbp (BMC Genomics. 2011; 12: 116, http://www.biomedcentral.com/1471-2164/12/116). (See other related assembly statistics from http://www.broadinstitute.org/annotation/genome/plasmodium_falciparum_spp/AssemblyStats.html) Using Sanger sequencing technology will get a better results of which the contig N50 is about 10 to 20kb. Here we demostrate that using PacBio(R) *RS* Single Molecule Real-Time (SMRT(R))

sequencing technology, we can easily assemble the genome much better results (N50 ˜= 954kb about 43x of the) than the earlier 2nd gen. sequencing results even with some simple home-made assembly code.

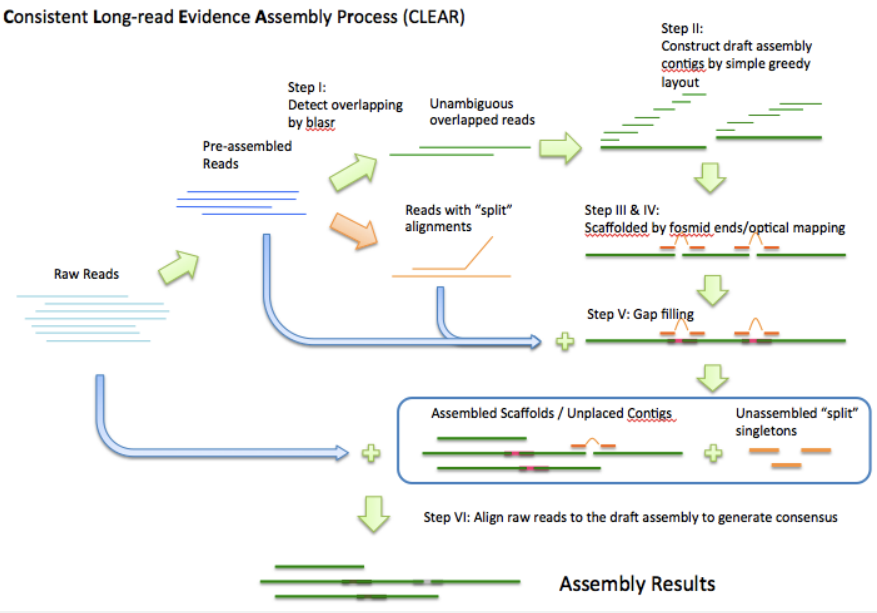
We choose the 3D7 strain because the avaiability of DNA and it is the only one that has good finished reference that we can compare our results. (see also http://www.broadinstitute.org/annotation/genome/plasmodium_falciparum_spp/GenomesIndex.html). We expect the performance will be similar to other strains of Plasmodium.

1.4 About "Consistent Long-read Evidence Assembling pRocess (CLEAR)"

The idea is very simple. When the read length is getting long, there should be more and more reads that can be "consistently" overlapped such that they are easy to be assembled with a very simple greedy layout algorithm. Namely, if there is no repeat or just short repeats within a long read, we should be able to get consistent overlapping alignments of such read to other long reads that come from the overlapped regions in the genome. We can examine the overlapping alignment information to indentify such reads. Those "split-reads," namely reads with non-consistent "split" structure in their alignments to others, are excluded initially in the assembly process. The "non-split" reads are assembled first as unitigs. We can then bring back the "split-reads" back to the assembly if the orders of the untigs can be resolved by other means. Or, we can assemble the "split-reads" using more sophisticated algorithms to resolve the repeats then bring the resulting contigs back to the assembly. Here we show the first few steps to assemble the Plasmodium 3D7 using such strategy.

The notebook covers the Step I and II for assembling Plasmodium 3D7.

```
from IPython.display import Image
Image(filename="CLEAR_Sketch.png")
```



A set of Plasmodium 3D7 sequencing data was generated using the PacBio(R) *RS* with C2 chemisty. Data from 30 SMRT(R) Cells was collected. After a pre-assembly process,

the raw reads were coverted to preassembled reads (p-reads). Here is a brief statistics of those p-reads. Some of the details for this preassembly strp is published in this paper URL: http://www.nature.com/nmeth/journal/v10/n6/full/nmeth.2474.html. The code for the preassembly step can be download from https://github.com/PacificBiosciences/HBAR-DTK.

```
!~/bin/PBJelly_12.7.25/summarizeAssembly.py preads.fa
```

```
Scaffold Stats
#Seqs 105869
Min 401
1st Qu. 2273
Median 4193
Mean 4232
3rd Qu. 5737
Max 16282
Total 448122809
n50 5398
n90 2506
n95 1831

====================
Contig Stats
#Seqs 105869
Min 401
1st Qu. 2273
Median 4193
Mean 4232
3rd Qu. 5737
Max 16282
Total 448122809
n50 5398
n90 2506
n95 1831

====================
Gap Stats
No Gaps!
====================
```

1.5 Using `blasr` to Get the Overlapping Information

For assembling the Plasmodium data with CLEAR, we first generate overlapping alignments using `blasr`.

The p-reads are aligned against each other to detect overlaps. We use the following `blasr` command to generate the alignment information.

```
blasr preads.fa preads.fa -maxScore -150 -m 4 -nproc 32 -bestn 64 -nCandidates 128\
-maxLCPLength 25 -minMatch 24 -scoreMatrix \
"-1 10 10 10 10  10 -1 10 10 10  10 10 -1 10 10  10 10 10 -1 10  10 10 10 10 10" -indel 10 \
-noSplitSubreads -out pr_pr_strigent.m4
```

The `blasr` alignment for this dataset takes quite a while. We use a pre-generated `pr_pr_strigent.m4` for this notebook.

1.6 The `simple_asm` Code Used to Generate the Assembly

The following the code can be used to generate the initial unitigs.

Import some standard modules for later:

```
import sys
import os
from pbcore.io import FastaIO
```

Define two utility functions:

- `rev_aln_strand()`: take the alignment information of a pair of read, return the alignment information of swapped query-target pair.
- `rev_cmp()`: reverse compliment sequence

```
def rev_aln_strand(aln):
    q_strand, q_s, q_e, q_l = aln[0]
    t_strand, t_s, t_e, t_l = aln[1]
    q_strand = 1 - q_strand
    q_s, q_e = q_l - q_e, q_l - q_s
    t_strand = 1 - t_strand
    t_s, t_e = t_l - t_e, t_l - t_s
    return ( (q_strand, q_s, q_e, q_l), (t_strand, t_s, t_e, t_l) )

rev_map = dict(zip("ACGTacgtNn","TGCAtgcaNn"))
def rev_cmp(seq):
    return "".join([rev_map[c] for c in seq[::-1]])
```

The `Overlap` class for creating objects to store the overlap information and related operations:

```
class Overlap(object):
    """
    represent the overlap information.
    """
    def __init__(self, query_id, target_id, aln, aln_score, aln_idt,
        containment_tolerance):
        "the alginment data will be normalized to such the the query is always 5' -> 3'"
        self.query_id = query_id
        self.target_id = target_id
        q_strand, q_s, q_e, q_l = aln[0]
        t_strand, t_s, t_e, t_l = aln[1]
        self.aln_score = aln_score
        self.aln_idt = aln_idt
        if q_strand == 1:
            self.aln = rev_aln_strand(aln)
```

```
        else:
            self.aln = aln
        self.t_offset = 0
        self.containment_tolerance = containment_tolerance
        self.overlap_type, self.t_offset = self.get_overlap_type(self.
            containment_tolerance)

    def get_overlap_type(self, containment_tolerance):
        q_strand, q_s, q_e, q_l = self.aln[0]
        t_strand, t_s, t_e, t_l = self.aln[1]
        t_offset = None
        if q_s < containment_tolerance and q_e > q_l - containment_tolerance:
            """ ----------------------------> target """
            """ ----------------------> query """
            """ or """
            """ <---------------------------- target """
            """ ----------------------> query """
            overlap_type = "contained"
            t_offset = - q_s
        elif t_s < containment_tolerance and t_e > t_l - containment_tolerance:
            """ ----------------------> target """
            """ ----------------------------> query """
            """ or """
            """ <---------------------- target """
            """ ---------------------------> query """
            overlap_type = "contains"
            t_offset = q_s
        elif q_s <= containment_tolerance and t_l - t_e <= containment_tolerance:
            """ ----------------------> target """
            """ --------------------> query """
            """ or """
            """ <---------------------- target """
            """ --------------------> query """
            overlap_type = "5p"
            t_offset = - q_s
        elif q_l - q_e <= containment_tolerance and t_s <= containment_tolerance:
            """ --------------------> target """
            """ ----------------------> query """
            """ or """
            """ <---------------------- target """
            """ --------------------> query """
            overlap_type = "3p"
            t_offset = q_s
        else:
            overlap_type = "split"

        #print q_strand, q_s, q_e, q_l
        #print t_strand, t_s, t_e, t_l
        #print overlap_type, t_offset
        return overlap_type, t_offset
```

`SequenceFragment` class for creating objects that store the sequence and overlapping information of a DNA fragment (sequence read):

```
class SequenceFragment(object):

    def __init__(self, id_):
        self.id_ = id_
        self.seq = None
        self.overlaps = []

    def append_overlap(self, overlap):
        self.overlaps.append(overlap)

    def load_seq(self, seq):
        self.seq = seq

    def best_overlap(self, end):
        score_ovlp = []
        for ovlp in self.overlaps:
            if ovlp.overlap_type != end:
                continue
            score_ovlp.append( ovlp )
        if score_ovlp:
            score_ovlp.sort(key = lambda x: x.aln_score )
            return score_ovlp[0]
        else:
            return None

    @property
    def best_3p_overlap(self):
        return self.best_overlap("3p")

    @property
    def best_5p_overlap(self):
        return self.best_overlap("5p")
```

Here we define a function `get_contained_or_split_reads_from_m4()` which will scan through the `blasr` m4 output and identify reads that have split alignment and the reads that are fully contained in the other reads.

Note that if a read has unique sequence > 500 bp for both three-prime and five-prime ends, it will not be called as "split" read.

We scan the `m4` file twice. The first time we only identify the split reads. The second time the code finds out the contained reads. A query read is called as "contained" if the target is not a "split read" or both query and target are "split".

```
def get_contained_or_split_reads_from_m4(m4_filename, permitted_error_pct,
    containment_tolerance):
    split_read_frgs = set()
    contained_read_frgs = set()
    with open(m4_filename) as m4_f:
```

```
        for l in m4_f:
            l = l.strip().split()
            q_name, t_name = l[0:2]
            if q_name == t_name:
                continue
            aln_score = int(l[2])
            aln_idt = float(l[3])
            if aln_idt < 100 - permitted_error_pct:
                continue
            q_strand, q_s, q_e, q_l = ( int(x) for x in l[4:8])
            t_strand, t_s, t_e, t_l = ( int(x) for x in l[8:12])
            aln = ( (q_strand, q_s, q_e, q_l), (t_strand, t_s, t_e, t_l) )
            ovlp = Overlap(q_name, t_name, aln, aln_score, aln_idt, containment_tolerance)
            #print " ".join(l), ovlp.overlap_type
            if ovlp.overlap_type == "split":
                if q_s < 500 or q_l - q_e < 500:
                    split_read_frgs.add( q_name )
                if t_s < 500 or t_l - t_e < 500:
                    split_read_frgs.add( t_name )

    with open(m4_filename) as m4_f:
        for l in m4_f:
            l = l.strip().split()
            q_name, t_name = l[0:2]
            if q_name == t_name:
                continue
            aln_score = int(l[2])
            aln_idt = float(l[3])
            if aln_idt < 100 - permitted_error_pct:
                continue
            q_strand, q_s, q_e, q_l = ( int(x) for x in l[4:8])
            t_strand, t_s, t_e, t_l = ( int(x) for x in l[8:12])
            aln = ( (q_strand, q_s, q_e, q_l), (t_strand, t_s, t_e, t_l) )
            ovlp = Overlap(q_name, t_name, aln, aln_score, aln_idt, containment_tolerance)
            #print " ".join(l), ovlp.overlap_type
            if ovlp.overlap_type == "split":
                continue
            elif ovlp.overlap_type == "contained":
                if t_name in split_read_frgs:
                    if q_name in split_read_frgs:
                        contained_read_frgs.add( q_name )
                else:
                    contained_read_frgs.add( q_name )
            elif ovlp.overlap_type == "contains":
                if q_name in split_read_frgs:
                    if t_name in split_read_frgs:
                        contained_read_frgs.add( t_name )
                else:
                    contained_read_frgs.add( t_name )
    return split_read_frgs, contained_read_frgs
```

The `get_unique_ovlp_reads_from_m4()` function scans the `blasr m4` file again to constuct `SequenceFragment` objects and the associated alignment. The "contained" reads and "split" are excluded. We expect the overlapping of non-split and non-contained reads are all "consistent".

```
def get_unique_ovlp_reads_from_m4(m4_filename, split_reads, contained_reads,
    permitted_error_pct, containment_tolerance):
    read_frgs = {}
    excluded_reads = split_reads | contained_reads
    #excluded_reads = contained_reads
    with open(m4_filename) as m4_f:
        for l in m4_f:
            l = l.strip().split()
            q_name, t_name = l[0:2]
            if q_name == t_name:
                continue
            if q_name in excluded_reads:
                continue
            if t_name in excluded_reads:
                continue

            aln_score = int(l[2])
            aln_idt = float(l[3])
            if aln_idt < 100 - permitted_error_pct:
                continue
            q_strand, q_s, q_e, q_l = ( int(x) for x in l[4:8])
            t_strand, t_s, t_e, t_l = ( int(x) for x in l[8:12])
            aln = ( (q_strand, q_s, q_e, q_l), (t_strand, t_s, t_e, t_l) )
            ovlp = Overlap(q_name, t_name, aln, aln_score, aln_idt, containment_tolerance)
            #print " ".join(l), ovlp.overlap_type
            if ovlp.overlap_type not in ["3p", "5p"]:
                continue

            read_frgs[q_name] = read_frgs.get( q_name, SequenceFragment(q_name) )
            read_frgs[q_name].append_overlap(ovlp)

            aln1, aln2 = ovlp.aln
            aln = aln2, aln1
            read_frgs[t_name] = read_frgs.get( t_name, SequenceFragment(t_name) )
            new_ovlp = Overlap(t_name, q_name, aln, ovlp.aln_score, ovlp.aln_idt,
                containment_tolerance)
            read_frgs[t_name].append_overlap(new_ovlp)

    return read_frgs
```

Take the DNA fragment and its overlapping data, we can start to walk through the overlapped reads to construt contigs just using the read sequences. Since we do expect to do a final round of consensus, we keep it simple here by ignoring minor errors within the reads. `get_path()` takes the read fragment data and an initial fragment identifier and the direction, 3-prime end or 5-prime end to extend, it will then find best overlaper from one end of the contig and extend the contig. The extension ends when there is no un-used fragment as the best overlapped read.

```
def get_path(read_frgs, init_read_frg, direction="3p", visited=set()):
    reverse_orientation = { "=>":"<=", "<=":"=>" }
    #visited = set()
    cur_frg = init_read_frg
    cur_orientation = "=>"
    cum_len = 0
    path = []
    seqs = []
    c_s = 0
    c_e = len(cur_frg.seq)
    while 1:

        visited.add(cur_frg.id_)

        if direction == "3p":

            if cur_orientation == "=>":

                if cur_frg.best_3p_overlap == None:
                    seqs.append( cur_frg.seq[c_s:] )
                    path.append( (direction, cur_orientation, cur_frg.id_, c_s, c_e,
                        next_overlap.query_id, q_strand, q_s, q_e, q_l, next_overlap.
                            target_id, t_strand, t_s, t_e, t_l) )
                    break
                else:
                    """
                                t_s t_e
                                -----------------> next_frg
                      --------------------> cur_frg
                                q_s q_e

                                t_e t_s
                                <----------------- next_frg
                      --------------------> cur_frg
                                q_s q_e
                    """
                    next_overlap = cur_frg.best_3p_overlap
                    q_aln, t_aln = next_overlap.aln
                    q_strand, q_s, q_e, q_l = q_aln
                    t_strand, t_s, t_e, t_l = t_aln
                    c_e = q_s
                    seqs.append( cur_frg.seq[c_s:c_e] )
                    path.append( (direction, cur_orientation, cur_frg.id_, c_s, c_e,
                        next_overlap.query_id, q_strand, q_s, q_e, q_l, next_overlap.
                            target_id, t_strand, t_s, t_e, t_l) )

                    #print "1:",c_s, c_e

                    if t_strand == 0:
                        c_s = t_s
                    else:
```

```
                    c_e = t_l - t_s
                    cur_orientation = reverse_orientation[cur_orientation]

            elif cur_orientation == "<=":

                if cur_frg.best_5p_overlap == None:
                    seqs.append( rev_cmp(cur_frg.seq[:c_e]) )
                    path.append( (direction, cur_orientation, cur_frg.id_, c_s, c_e,
                        next_overlap.query_id, q_strand, q_s, q_e, q_l, next_overlap.
                            target_id, t_strand, t_s, t_e, t_l) )
                    break
                else:
                    """
                                t_e t_s
                                <----------------- next_frg
                      <-------------------- cur_frg
                                q_e q_s

                                t_s t_e
                                -----------------> next_frg
                      <-------------------- cur_frg
                                q_e q_s

                    """
                    next_overlap = cur_frg.best_5p_overlap
                    q_aln, t_aln = next_overlap.aln
                    q_strand, q_s, q_e, q_l = q_aln
                    t_strand, t_s, t_e, t_l = t_aln
                    c_s = q_e
                    seqs.append( rev_cmp(cur_frg.seq[c_s:c_e]) )
                    path.append( (direction, cur_orientation, cur_frg.id_, c_s, c_e,
                        next_overlap.query_id, q_strand, q_s, q_e, q_l, next_overlap.
                            target_id, t_strand, t_s, t_e, t_l) )

                    #print "3:",c_s, c_e

                    if t_strand == 0:
                        c_e = t_e
                    else:
                        c_s = t_l - t_e
                        cur_orientation = reverse_orientation[cur_orientation]

        if direction == "5p":

            if cur_orientation == "=>":

                if cur_frg.best_5p_overlap == None:
                    seqs.append( cur_frg.seq[:c_e] )
                    path.append( (direction, cur_orientation, cur_frg.id_, c_s, c_e,
                        next_overlap.query_id, q_strand, q_s, q_e, q_l, next_overlap.
                            target_id, t_strand, t_s, t_e, t_l) )
```

```
                break
            else:
                """
                     t_s t_e
                ------------> next_frg
                      --------------------> cur_frg
                     q_s q_e

                     t_e t_s
                <------------ next_frg
                      --------------------> cur_frg
                     q_s q_e
                """
                next_overlap = cur_frg.best_5p_overlap
                q_aln, t_aln = next_overlap.aln
                q_strand, q_s, q_e, q_l = q_aln
                t_strand, t_s, t_e, t_l = t_aln
                c_s = q_e
                seqs.append( cur_frg.seq[c_s:c_e] )
                path.append( (direction, cur_orientation, cur_frg.id_, c_s, c_e,
                    next_overlap.query_id, q_strand, q_s, q_e, q_l, next_overlap.
                        target_id, t_strand, t_s, t_e, t_l) )

                #print "2:",c_s, c_e

                if t_strand == 0:
                    c_e = t_e
                else:
                    c_s = t_l - t_e
                    cur_orientation = reverse_orientation[cur_orientation]

        elif cur_orientation == "<=":

            if cur_frg.best_3p_overlap == None:
                seqs.append( rev_cmp(cur_frg.seq[c_s:]) )
                path.append( (direction, cur_orientation, cur_frg.id_, c_s, c_e,
                    next_overlap.query_id, q_strand, q_s, q_e, q_l, next_overlap.
                        target_id, t_strand, t_s, t_e, t_l) )
                break
            else:
                """
                         t_e t_s
                  <----------------- next_frg
                          <-------------------- cur_frg
                         q_e q_s

                         t_s t_e
                  -----------------> next_frg
                          <-------------------- cur_frg
                         q_e q_s
```

```
                    """
                    next_overlap = cur_frg.best_3p_overlap
                    q_aln, t_aln = next_overlap.aln
                    q_strand, q_s, q_e, q_l = q_aln
                    t_strand, t_s, t_e, t_l = t_aln
                    c_e = q_s
                    seqs.append( rev_cmp(cur_frg.seq[c_s:c_e]) )
                    path.append( (direction, cur_orientation, cur_frg.id_, c_s, c_e,
                        next_overlap.query_id, q_strand, q_s, q_e, q_l, next_overlap.
                            target_id, t_strand, t_s, t_e, t_l) )

                    #print "4:",c_s, c_e

                    if t_strand == 0:
                        c_s = t_s
                    else:
                        c_e = t_l - t_s
                        cur_orientation = reverse_orientation[cur_orientation]



        #print direction, next_overlap.query_id, next_overlap.target_id, next_overlap.aln
        #print c_s, c_e
        #if next_overlap.aln[1][0] == 1:
        # cur_orientation = reverse_orientation[cur_orientation]
        cur_frg = read_frgs[next_overlap.target_id]
        if cur_frg.id_ in visited:
            break

    if direction == "5p":
        seqs.reverse()

    return "".join(seqs), path
```

The two utility funtions below are defined for loading the sequence data into the `SequenceFragment` objects and outputing the split read fasta file.

```
def load_seq(frgs, fasta_fn):
    f = FastaIO.FastaReader(fasta_fn)
    for r in f:
        if r.name in frgs:
            frgs[r.name].seq = r.sequence

def output_split_reads(split_reads, contained_reads, fasta_fn, out_fn):
    f = FastaIO.FastaReader(fasta_fn)
    included = split_reads - contained_reads
    with open(out_fn, "w") as out:
        for r in f:
            if r.name in included:
                print >> out, ">"+r.name
                print >> out, r.sequence
```

Calling the funtions `get_contained_or_split_reads_from_m4()` and `get_unique_ovlp_reads_from_m4()` defined above provide all necessary data for assembly. The split reads are output to a fasta file.

```
pread_fn = "preads.fa"
aln_m4_fn = "pr_pr_strigent.m4"
output_prefix = "asm"

split_reads, contained_reads = get_contained_or_split_reads_from_m4(aln_m4_fn, 4, 100)
uniq_ovlp_reads = get_unique_ovlp_reads_from_m4( aln_m4_fn, split_reads, contained_reads
    , 4, 100)

load_seq(uniq_ovlp_reads, pread_fn)
output_split_reads(split_reads, contained_reads, pread_fn, output_prefix+"_split.fa")
```

We first scan through the overlapping information of each fragment. If a fragment only has three-prime or five-prime overlap, we put them into the `seeds` list which contains fragements used as "seeds" for contructing contigs by simple extension. We also sort these "seeds" by the fragment length. We will start to extend to construct contigs from the longest seed.

```
seeds = []
all_reads = set()
for read_id, read_frg in uniq_ovlp_reads.items():
    all_reads.add(read_id)
    #print ctg_id, ctg_frg.best_5p_overlap, ctg_frg.best_3p_overlap
    if read_frg.best_5p_overlap == None and read_frg.best_3p_overlap != None:
        seeds.append( (len(read_frg.seq), read_id, read_frg, "3p") )
    if read_frg.best_3p_overlap == None and read_frg.best_5p_overlap != None:
        seeds.append( (len(read_frg.seq), read_id, read_frg, "5p") )

seeds.sort(reverse=True)
```

Here is the main loop to contruct contigs. It first goes through all seed sequences and contruct contigs using the function `get_path()`. After that, it will try to contruct contigs using those reads that are not yet in any contigs. This way, we will use all non-split and non-contained reads to construt contigs. The assembled utitigs are in `asm.fa`.

```
visited = set()
ctg_id = 0
ctg_layout_fn = output_prefix + ".lay"
ctg_layout = open(ctg_layout_fn, "w")
with open(output_prefix + ".fa","w") as f:
    for read_len, read_id, read_frg, direction in seeds:
        if read_id not in visited:
            seq, path = get_path(uniq_ovlp_reads, read_frg, direction=direction, visited =
                visited)
            if len(seq) < 200:
                continue
            print >>f, ">ctg_%06d" % ctg_id
            print >>f, seq
```

```
            print >> ctg_layout, ">ctg_%06d" % ctg_id
            for p in path:
                print >> ctg_layout, " ".join([str(c) for c in p])
            ctg_id += 1

    for read_id in all_reads - visited:
        if read_id not in visited:
            read_frg = uniq_ovlp_reads[read_id]
            seq, path = get_path(uniq_ovlp_reads, read_frg, direction="3p", visited =
                visited)
            if len(seq) < 200:
                continue
            print >>f, ">ctg_%06d_s" % ctg_id
            print >>f, seq
            print >> ctg_layout, ">ctg_%06d_s" % ctg_id
            for p in path:
                print >> ctg_layout, " ".join([str(c) for c in p])
            ctg_id += 1
    print >> ctg_layout, "#", all_reads - visited, len(uniq_ovlp_reads)
ctg_layout.close()
```

1.7 Check Out The Assembly Results

Here we show the assembly summary statistics. We get pretty good N50 = 953.7 kb. The total size of the assembly is 23.3 Mb, which is very close to the reference.

```
%%bash
~/bin/PBJelly_12.7.25/summarizeAssembly.py asm.fa
```

```
Scaffold Stats
#Seqs 487
Min 202
1st Qu. 466
Median 1180
Mean 47981
3rd Qu. 3456
Max 1899581
Total 23367177
n50 953696
n90 99851
n95 41702

====================
Contig Stats
#Seqs 487
Min 202
1st Qu. 466
Median 1180
Mean 47981
3rd Qu. 3456
Max 1899581
```

```
Total 23367177
n50 953696
n90 99851
n95 41702

====================
Gap Stats
No Gaps!
====================
```

It is interesting to see what percentage of reads are classified as "split".

```
%%bash
grep -c ">" preads.fa
grep -c ">" asm_split.fa
```

```
105869
707
```

Only about $707/105869 = 0.668\%$ of reads are split. This says the simple logic to separate "easy to assemble" potion of reads from the "diffculte to assemble" potion is quite efficient even in this genome, which is almost 80% AT. Why? The fundementals here are when the reads are getting long, most of the overlapping of the reads is not confounded by shorter repeats. We can easily assemble a large amount of the genome without worrying about those shorter repeats. In other words, it is quite possible to use a simple algorithm to get very good assembly when the reads are very long.

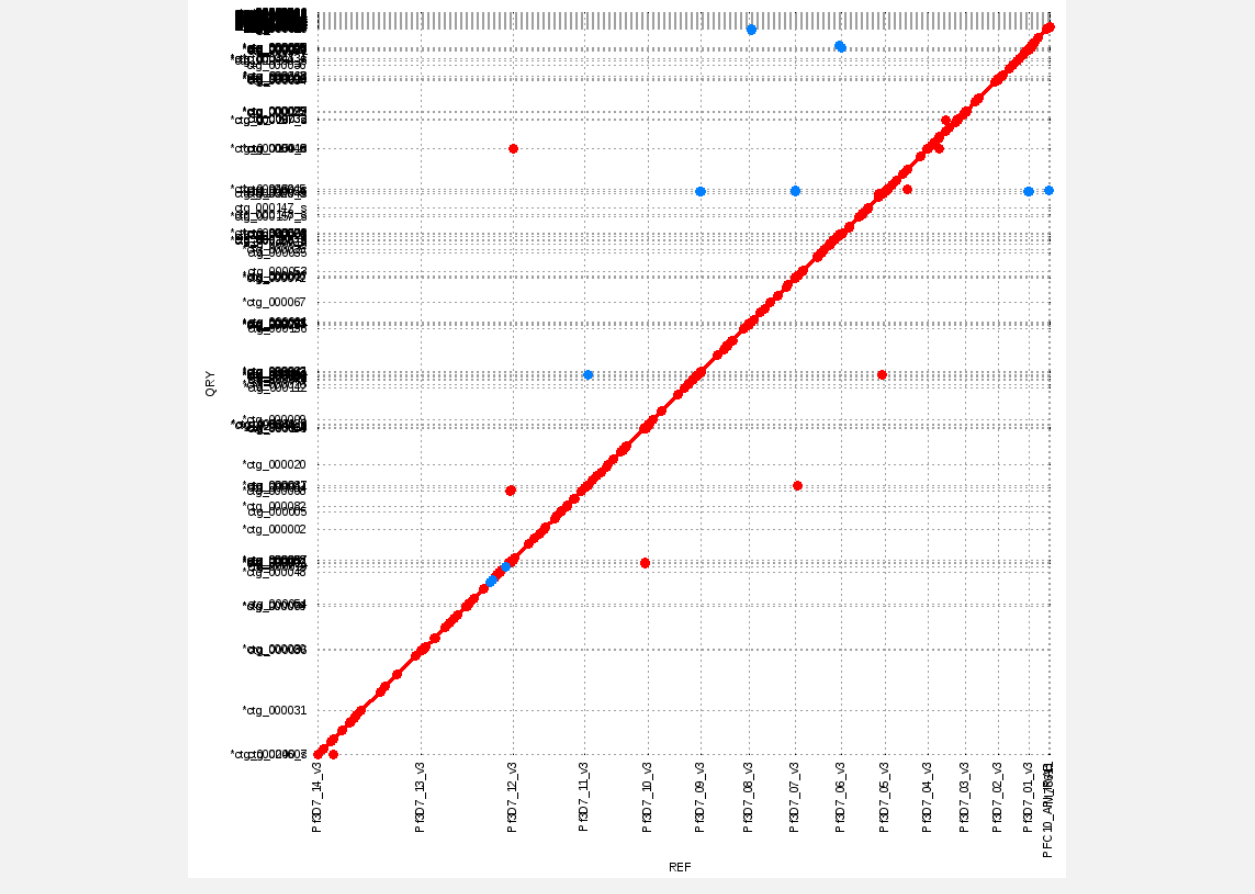
How does the result look like comparing to the Plasmodium 3D7 refrenece? We can use `nucmer` to do a genome-wide alignment to check the results.

```
nucmer -mum PlasmoDB-9.2_Pfalciparum3D7_Genome.fasta asm.fa -p genome_wide_aln
```

We can use `mummerplot` to compare the assembly from the alignment result:

```
!mummerplot -t png -l -fat -f genome_wide_aln.delta -p genome_wide_aln
Image(filename = "genome_wide_aln.png")
```

```
gnuplot 4.2 patchlevel 6
Writing filtered delta file out.filter
Reading delta file out.filter
Writing plot files out.fplot, out.rplot
Writing gnuplot script out.gp
Rendering plot out.png
```



As one can see from the alignment above, the assembly is already very good in term of contiguity. The largest chromosome is bascially covered by two contigs.

At this stage, we might need some data, e.g. fosmid ends or physical mapping, for longer range information to improve the assembly. Once the longer range helps to resolve the order of the contigs, we will be able to unambigiously place reads back in the gaps. After the gap-filling step, we can then apply the Quiver consensus algorithm to get better accuracy at the end of the assembly process.