Detailed *fragScaff* description

A. Sequence data processing prior to *fragScaff*

i. **Group assignment of reads (optional).** CPT-seq default read names are a concatenation of the four index reads from the sequencing run in the following format: @[Nextera i7 Index][PCR i7 Index][PCR i5 Index][Nextera i5 Index], which corresponds to: @[Read 2][Read 3][Read 4][Read 5] of the run. Barcode deconvolution can be performed using the provided script (`CPTseq_barcode_split.pl`) that looks at each read in the full barcode and performs an edit-distance based assignment. If the run is of high quality, particularly for the index reads, this step can be skipped and *fragScaff* will only use perfect-match index reads which will typically result in a loss of roughly 8% of the reads, though for lower quality runs this can be up to 40% in which case barcode deconvolution can reduce it down to roughly 5% depending on the run.

ii. **Read alignment.** Alignment of reads to a reference created from the input contig/scaffold assembly and creation of a sorted bam file with valid group and sequence header lines. In order to be sure these lines are correct it is recommended to first generate a sequence dictionary for the reference, eg. `myReference.fa.dict`, and then use the provided group header (`group_header.txt`). These can then be used instead of the default header lines that alignment software will output by using the following call where "`myAlignmentCall`" is whatever aligner you prefer that outputs in a sam file format:

```
(cat myReference.fa.dict group_header.txt &&
 myAlignmentCall | awk '($1!~/^@/)') |
samtools view -bSu - | samtools sort - myOutfile;
```

B. Input contig/scaffold pre-processing (optional but recommended)

i. **N-base bed file.** Generation of a bed file that contains the coordinates of all N-base stretches in the assembly (ie. when the input is in the form of scaffolds and not contigs). This can be generated using the included script: `fragScaff_generate_Nbase_bed.pl`. This file will aid in the determining of node-end boundaries such that N-bases will not be included and each node end will ideally have the same number of valid bases.

ii. **Repeat bed file.** Generation of a bed file that contains all high-identity repeats in the assembly. This can be generated by performing a blastn alignment of the input assembly to itself. We recommend using the following blast parameters: `-word_size 36 -perc_identity 95 -outfmt 6`. The output can then be filtered using the provided script (`fragScaff_self_blast_to_repeat_bed.pl`) followed by sorting and merging of entries. Alternatively identifying alignments to Repbase also works well.

C. *fragScaff* algorithm

The *fragScaff* program has three major stages that can be run independently. This allows for the first two time consuming stages that require very little parameter optimization to only be run once and the much quicker graph filtering and manipulation stage with a large amount of parameter optimization to be run multiple times.

i. **Bam file parsing and node determination.** In this stage the bam file is parsed first to read in the contigs, groups, and then the alignments.

   a. **Bam header parsing.** The header of the bam file is parsed to read in the names and lengths of all of the input contigs or scaffolds as well as the names of all of the read groups (9,216 for CPT-seq).

   b. **Node ends are determined.** The size of the node ends is defined by `-E` with a default of 5,000 bp. If a N-base (`-N`) and/or repeat bed file (`-J`) is defined, *fragScaff* will read in the coordinates and take them into consideration when defining the node boundaries such that there is a total of `-E` bases of viable sequence included at each node end up to a certain maximum limit (`-o`, default 10,000 bp). If a contig or scaffold is shorter than `-E` then then entire contig or scaffold serves as the node and downstream orientation is not possible. It is also possible for the end nodes to overlap in the middle of the input scaffold or contig which will therefore result in limited ability for orientation.

   c. **Node group hit identification.** The reads are then parsed with a defined alignment quality score threshold (`-q`, default 10) and read groups determined by either the read name (`-G N`, default for CPT-seq), a tag at the end of the read name (`-G H`), or the standard RG:Z:[read group] sam field (`-G R`). If the read aligns to a node it is considered a hit and stored.

   d. **bamParse output.** After all reads are parsed a "bamParse" file can be generated (if `-b [0/1]` is defined, 0 will make the bamParse file and continue, 1 will make the bamParse file and exit) that contains all of the node and group hit information so that later runs do not have to parse through the entire bam file again which can take several hours for very large bam files. It is also useful since the bam parsing is single thread and the all-by-all shared fraction calculations in the next step are designed to be multithreaded (see Multithreading box below).

ii. **All-by-all shared fraction calculations.** In this stage the fraction of shared groups within any two nodes' set of groups is calculated. The distribution of this shared fraction for each node to all other nodes is calculated and the outlier probability score is calculated for each.

   a. **Node filtering.** After the bamParse file has been generated or read in, the nodes can then be filtered according to several specifications. First, only groups which have a minimum number (`-C`) reads that hit a node will be counted as true hits (default is only 1 read for CPT-seq and more for LFR or Fosmid approaches hat have a higher density of reads). Next, the number group hits for each node is calculated and the nodes on the extremes of the distribution (minimum and maximum fractional cut set by `-d` and `-D` respectively with defaults of 0.05 and 0.95 to exclude the lowest and highest 5 percent). This allows the removal of nodes that likely have unidentified repeats that may result in increased false links as well as those that may be improperly assembled sequence.

      It is important to note that the threshold fractions are designed to identify the threshold hit counts and ensue only that a minimum of the extreme fractions identified will be excluded. For instance, if a sample has 10% of nodes with zero group hits, the

threshold (with a `-d` of 0.05) would be set to 0, and the entire 10% at zero would be excluded. Additionally the option `-U` exists which allows a minimum number of groups to have hit the node for inclusion. However the default is set to 0 since nodes included with a low number of hits, eg. one group, will not produce significant outlier scores and links would not be established.

b. Node shared fraction calculations. Each node then has its shared fraction with all other nodes calculated. The shared fraction is defined as the number of groups that hit both of the nodes over the total number of unique groups that hit either one of the nodes. After the shared fraction is calculated between the initial node and each other node, the mean and standard deviation of the shared fractions can be used to calculate the Gaussian probability of each other node's shared fraction falling within the distribution of the initial node's distribution. The score is then the $-\log_{10}$ of the Gaussian probability and is calculated for each other node with respect to the initial node.

c. Link file output. The scores that are above a minimum report threshold (`-r`, default 1) are included in the link file output. This file is designed to be inclusive such that later, more stringent score cutoffs can be used without having to re-do the all-by-all node calculations. If *fragScaff* is run with the `-A` option, it will exit after the generation of the link file to allow for more control of thread usage as all further steps are single-thread and much quicker.

Multithreading all-by-all shared fraction calculations.

The all-by-all shared fraction calculations is the most computationally intense portion of *fragScaff* and therefore variations of multithreading have been included. This option is specified by the –t option and can be specified in several ways described below.

`-t 1` **Single thread.** This is the default *fragScaff* threading option, though is not recommended as the all-by-all calculations can take some time, especially when there is a high number of input contigs or scaffolds.

`-t [>1]` **Multithread, single machine.** When specifying –t to more than one, this number of threads will be spun off on the same machine that the parent instance of *fragScaff* was run on. It is important to note that these are separate instances of the *fragScaff* program and not true multithreading as this was found to have optimal memory usage. For this run mode the number of nodes per thread can be set by `-S` (default is 100) and also the `-k` option must be the valid call for *fragScaff*.

`-t Q` **SGE mode.** In this mode qsub functionality will be used. It is required that the qsub command is callable by the system to use this option. It also has another set of options to control the qsub job array call in addition to the `-k` and `-S` commands when performing the standard multithreading. `-T` allows specification of the max number of jobs to run at a given time (`-tc` for the qsub command, default N, which means no throttling but an integer can be set), and `-e` is used to define the amount of memory to allocate per qsub job (default is 2G, but more may be required for large number of input contigs or scaffolds).

iii. **Graph manipulation.** In this stage the links are used from the previous step or the link file is loaded (specified by `-K`) and edges are created between nodes if they meet various thresholds. The resulting graph is then manipulated to produce a final node ordering.

a. **Score cutoff determination.** All links with a score greater than `-r` (default 1) are included in the link file and loaded in. However this score is designed to be extremely permissive and only there to reduce the file size of the links file by eliminating the majority of potential links that fall well within a node's shared fraction distribution. To identify outliers that are true links the score can either be manually set or automatically determined using the option `-p`, with the default as `-p A` for automatic determination. If `-p` is set to a value then this score will be used for all link filtering.

For automatic score cutoff estimation (`-p A`), *fragScaff* will attempt to determine the optimum score cutoff based on what seems to have worked on a number of genomes. First, *fragScaff* finds the mean number of links across all nodes that would pass a given threshold score for each integer score ranging from 1 to 200. It then finds the minimum score threshold such that the mean number of passing links hits `-j` (default is 1.25, for a mean of 1.25 passing links per node). **Setting `-j` is highly dependent on the input assembly and is the most important parameter to tweak during the scaffolding process.** For an assembly with a fairly high N50 (eg. 75+ kbp) setting `-j` to 1.25 (the default) seems to work well. However for smaller N50 assemblies (eg. 50 kbp) it is recommended to increase this value as a higher number of links would be expected due to some links being able to skip a contig and link to the next. Varying this parameter and even following up by manually setting `-p` may be necessary.

To reduce the time it takes to run each variant of the link filtering options it is recommended to have previously generated the bamParse file (specify with `-B`) and the link file (specify with `-K`) so that they do not need to be re-generated, and also to run using the `-A` option which will exit after the graph manipulations and not go through the process of reading in the input assembly fasta file and outputting the final fasta assembly which is I/O intensive and can take some time, whereas the graph manipulation steps should take between 1 and 20 minutes depending on the number of input assembly contigs or scaffolds.

b. **Link filtering and edge determination.** After the score threshold has been set, *fragScaff* will read in all of the links that pass the threshold and perform further filtering of the links to produce the final edges of the graph. For each node the set of passing links is first trimmed to include only the maximum scoring set, for which the amount is determined by `-l` (default 5), though most nodes should have a number of passing links below this cutoff. This filtering is done so that the lower-scoring links that may be more distant are not included, as the high-scoring links are the only ones that are relevant. An additional filter will entirely remove nodes that have a number of links greater than `-a` (default 20) as they are likely due to unidentified repeats, though as with the `-l` filtering, very few nodes, if any, should be removed using this filter, particularly if a repeat bed file is used for the initial bam parsing.

The next filtering requirement for a link to be used as an edge is that the link is reciprocated. For instance, if node1 has a passing link to node2, the link from node2 to node1 must also be passing. This can also be adjusted by the use of the `-u` option

which is the multiplier of the score cutoff that is required for the link to be considered valid. The default `-u` value is 2, which imposes no additional link stringency, however this can be adjusted to be more inclusive. For instance, a slightly more permissive score cutoff can be used (either by increasing `-j` in the automatic determination, or by specifying a lower `-p` than the recommended value produces using the default `-j`) and a higher `-u` multiplier can be used.

Example: In one case the –p value that is determined automatically using –j 1.25 is 20 and in a second case a more permissive run where –j 2.0 produces a score cutoff of 16. For the second case the –u value can be set to require a higher total reciprocated link score, such as 2.5 to allow recovery of the weaker links if the reciprocating link is strong enough to compensate.

In the first case, a link with scores of 18 and 26 would not form an edge since the link with a score of 18 does not pass the score cutoff and a link with scores 17 and 18 would also fail as neither link passes the cutoff.

In the second case, the first link with scores of 18 and 26 would pass with a –u set to 2.5 since both links pass and the combined score is at least 2.5 times the score cutoff ($18 + 26 \geq 16*2.5 = 44 \geq 40$). However, the second link would not pass even though both of the link scores are above the score threshold but the combined score is not high enough ($17 + 18 < 40$).

c. **Minimum spanning tree identification.** Next, subgraphs are identified and the maximum scoring minimum spanning tree (MST) is determined. An MST is used for three reasons: i) it is computationally tractable using Prim's Algorithm, ii) the majority of subgraphs are very close to an MST already, except for very short input contigs or scaffolds, and iii) since the edges are weighted, the highest weight edges are likely the closest. It is also important to note that even though an MST is determined, the edges that are not included in the MST are still stored and not discarded as they are utilized in the placement of branches in the graph.

d. **Trunk identification.** The longest path through the MST is then determined and classified as the trunk. This is done by walking through the graph between every pair of degree one nodes in the MST and finding the longest of these paths. If no degree one nodes are present, degree two nodes are used and so on, though this has not yet been observed as the majority of MSTs are already very close to the trunk with only very few branches.

e. **Branch placement.** The branches that are not present in the MST are then placed by finding the highest weighted path through the new trunk as each node is placed. It is important to note that the vast majority of the branches are single-node branches in which their node partner (the node on the other side of the input contig or scaffold) is part of the trunk. In these cases the placement of the branch is purely an orientation issue. Furthermore, the branches are restricted from being placed between a contig or scaffold node pair, which would be extremely unlikely given that the edge weights for node pairs are set much higher than the possible range of non-node pair edges.

D. *fragScaff* output and interpretation

i. **Log file.** The most informative output file for *fragScaff* is the log file. This file is output as the `-O` output prefix followed by "`.fragScaff.log`". (Note: if the *fragScaff* run is just generating a bamParse file, the log will be "`.bamParse.log`"). The log file has details regarding the input provided as well as a full call for the run with all options listed. It also has a wealth of information regarding the performance at each step of the way detailed below.

   a. **Input information.** The first lines are regarding the input files provided. If a bamParse is provided it will use the file handle (auto-generated when creating a bamParse) to determine the `-E` option. If this is incorrect in the log file, there will likely be complications. If a links file is provided it will also be listed along with the detected options `-d`, `-D`, `-E`, and `-r`. The next line lists all of the options that correspond to the run of *fragScaff*.

   b. **Node determination.** The next lines are regarding either parsing through the bam file to generate a bamParse, or loading in the bamParse. It will output the number of contigs, nodes, and groups as well as the minimum and maximum thresholds for the number of group hits per node in order to exclude the extreme high and low group hit windows. The lower cut should be in the range of 50-300, and upper range from 300 to 800. If these numbers are too high, it likely means the CPT-seq run had too much input going into the PCR, if it is substantially lower (particularly for the upper bound) it either means too little into the PCR, or difficulties in the run.

   c. **Link calculations.** The next line is either for reading in the link file provided by `-K`, or the execution of the sub-threads or SGE jobs for the all-by-all node calculations. For the SGE run mode the job name is provided in case any modifications to the job array must be made.

   d. **Edge determination and graph manipulations.** The next lines first describe the automatic determination of `-p`, assuming `-p` is set to `A`. If it is not set to auto determine, it will instead list the **–**p cutoff provided. If the links file is generated in this run (and not provided by `-K`) it will output the recommended `-p` score cutoff (based on the provided mean link cutoff, `-j`) regardless of whether or not it is set to auto determine. It will then list the number of nodes that are included in the *fragScaff* assembly, the number that were trimmed due to an excessively high valid link count (set by `-l`) which should be low for high N50 input assemblies and higher (up to ~40%) for assemblies with a shorter N50, as well as the number of nodes excluded due to high node counts (set by `-a`) which should be no more than 1-3%. The next lines describe building the graph and the subsequent graph manipulations.

   e. **Estimated N50 improvement.** *fragScaff* will attempt to estimate the N10, N50, and N90 improvements from the scaffolding from the input scaffold/contig lengths. It does not take into account N-bases, so the numbers are just estimates and will not be the same as the final N50 numbers (though they should be fairly close, especially if the input is contigs as opposed to scaffolds). The improvement estimations will be made even if –I is specified and *fragScaff* exits prior to printing the final fasta output.

f. **Final output.** The final lines are regarding the fasta output file and the final scaffold count of the assembly (this number includes input scaffolds that did not get included in the *fragScaff* process).

ii. **bamParse file.** The bamParse file is generated after the bam header has been read, the node boundaries determined, and the alignments stored. This file has the details required for the *fragScaff* assembly in a much more condensed form and allows subsequent runs of the program to skip parsing through the bam file. This file can be provided as `-B` instead of the bam file for subsequent runs.

iii. **Links file.** The links file is generated after the all-by-all node calculations. It reports all of the links for each node above a certain threshold (set by `-r`). Additionally it contains the shared fraction of groups hitting the node-pairs that belong to the same input contig/scaffold which is used for determining the orientation quality score. The information in this file is designed to be inclusive such that future runs of the program can take care of the link filtering. It can be provided as `-K` in order to skip the all-by-all node calculations step for subsequent runs.

iv. **Ordered node file.** The ordered node file lists the scaffolds that are generated by *fragScaff*. It can be used as the input to the script: `fragScaff_check_ordering.pl` which compares the assembles scaffolds to their alignment to a trusted reference and will output relevant accuracy statistics. This script additionally requires a mapping file for the contigs to a trusted reference in a tab delimited format with the columns: contig/scaffold name, chromosome, start position, and end position. If there is no valid alignment for the contig or scaffold, the chromosome and coordinate columns should be replaced with a "`-1`".

v. **Output fasta.** This file contains the newly scaffolded fasta file with N-bases placed between the joins made by *fragScaff*. It also contains all input contigs/scaffolds that were not included in the assembly.

vi. **Qual file.** This file provides the quality information for the links as well as the orientation for the scaffolds. It is intended to guide future analysis or future scaffolding by informing where the weak points are in the assembly.

vii. **N50 file.** This file provides details on the input and output max scaffold size, N10, N50, and N90 with and without N-bases.

viii. **Cytoscape file.** If the `-V` option is specified a ".csv" file will be generated that can be used as an input to cytoscape for visualization of the graphs. If a number is specified to `-V` it will output only that many scaffolds semi-randomly (biased towards larger scaffolds). If it is set to "`A`" it will output all scaffolds included in *fragScaff*. Edges are listed as either "`TRUNK`" which is a subset of the minimum spanning tree edges "`MST`" which is a subset of all of the edges "`EDGE`". The file also has the edge weight score and the scaffold ID that the vertices and edges belong to.

E. *fragScaff* computational requirements

The computational requirements and run times of *fragScaff* are directly tied to the size of the bam file (initial I/O) and the number of input contigs or scaffolds. In order to optimize the process, *fragScaff* is divided into the three main steps described above: (i) bam file parsing and node determination, (ii) all-by-all shared fraction calculations, and (iii) graph manipulation followed by the final assembly output. Since each step can be performed independently, the requirements are addressed separately below. Specific examples are also provided for the 437 kbp N50 input human assembly as well as the fly assembly. (Human = 18,922 input scaffolds, 2.73 Gbp assembled, 334 M reads; Fly = 7,109 input scaffolds, 127 Mbp assembled, 1,324 M reads).

i. **Bam file parsing and node determination.** This initial step is primarily comprised of reading in the bam file and is single-threaded. The memory requirement is low (<1 Gb), and is purely based on the size of the bam file. This process takes approximately twice as long as running a bam->sam conversion using samtools. (Human = 4 hours, < 1Gb memory; Fly = 48 minutes, < 1 Gb memory)

ii. **All-by-all shared fraction calculations.** After the ".bamParse" file has been generated, the all-by-all calculations must be performed. This step is highly dependent on the number of input contigs and scaffolds with respect to run time ($O(n^2)$); however the memory footprint is similar to that of the first step at <1 Gb. *fragScaff* has been developed with multithreading capability either on the node in which it is being run, or with the capability to execute jobs using SGE. The following table provides approximate run times using varying number of threads or concurrent SGE jobs. Again these numbers are approximations and vary based on the assembly, particularly with respect to repeat content which can either increase the run times if the repeats are not masked out, or decrease them if thorough repeat masking is performed. (Human = 22 minutes (50 jobs on SGE), < 1 Gb memory; Fly = 6 minutes (8 threads); < 1 Gb memory)

Input Contig/Scaffolds	Number of threads (or concurrent SGE jobs)	Approximate Run Time
< 10,000	8 (threads)	10 – 20 min
10,000 – 30,000	50 (SGE)	10 – 60 min
30,000 – 100,000	50 (SGE)	1 – 15 hr
100,000 – 250,000	50 (SGE)	15 – 48 hr

iii. **Graph manipulation.** Both the run time and memory of the graph manipulation steps are based on the number of input contigs and scaffolds. The runtime also varies considerably based on the repeat content and the masking of repeats in the input assembly. All things considered, this is the fastest step, with input assemblies containing just over 100,000 input contigs requiring less than 15 minutes. (16.3 Gb memory footprint). (Human = 2 minutes, 8 Gb memory; Fly = seconds, < 1 Gb memory)

iv. **Final assembly output.** This is another single thread process and first involves running the graph manipulation step, but allowing *fragScaff* to continue on to produce the final output. The memory requirement is approximately 5*(size of assembly). The run time of this step is also heavily I/O bound and takes approximately three times as long as it would take to cat the input assembly fasta. (Human = 1.8 hours, 13.2 Gb memory; Fly = 4 minutes, < 1 Gb memory)