Two Locus General Statistics in tskit

Lloyd Kirk, Ragsdale Lab

July 11, 2023

1 Background

The intention of this document is to describe the changes to tskit in order to provide a generalized framework for computing two-locus statistics for branches and sites.

Some of the design proposal is taken from GH-432 There has been a bit of discussion on GH-1900 as well.

2 Goal/Scope

Ultimately, we intend to deprecate the LD calculator that is currently implemented in tskit and implement a general framework for two-locus statistics. This framework will create an interface for implementing new summary statistics and implement the following features:

- Statistics for sites with more than 1 mutation (multiallelic sites)
- Polarization, where applicable
- Branch Statistics
- LD Stats beyond r^2

3 Implementation

We have been trying to follow the general design pattern laid out in the C api. Most of the initial design documentation will focus on creating the necessary C interfaces to consume from the python api.

Some of this documentation is based on my (LK's) interpretation of the design patterns in the existing code. I have no insight into how things might change apart from the scattered TODOs in the code. In short, please feel free to correct my assumptions/understandings.

3.1 Outer Layer

Each summary function we intend to implement will have an outer wrapper. For example, the r^2 statistic, will have a `r2_summary_func` and a corresponding `tsk_treeseq_r2`. These wrappers will call into `tsk_treeseq_sample_count_stat`, which will check our sample sets and produce a matrix of weights.

After these generic preparations (we will want to consider how we're handling windows, see my proposal for the window design notebook), the code will call into `tsk_treeseq_general_stat`, which is the main entrypoint for computing stats from tree sequences. Within `tsk_treeseq_general_stat`, we will add two more conditions and a couple more flags (`tsk_flags_t`).

```
bool stat_two_site = !!(options & TSK_STAT_TWO_SITE);
bool stat_two_branch = !!(options & TSK_STAT_TWO_BRANCH);
```

These flags will dispatch our entrypoints for computing two site/branch statistics, allowing us to use the same entrypoint function for all site statistics.

3.2 Two Site Statistics

We will provide a similar interface to computing `tsk_treeseq_site_general_stat`, calling our version `tsk_treeseq_two_site_general_stat`. This function will first populate an array of all samples under each given mutation. Then, it will free some intermediate data and allocate the output array. Finally, it will compute the two site stat results and store them in the output array, normalized by the normalization strategy for the summary function (see 4.1).

3.2.1 Bit Arrays

Because we're storing samples instead of node counts, we must be efficient in the way that we store samples.

The data structure that allows us to efficiently store samples is known in our implementation as a `tsk_bit_array_t`. This is an array of unsigned 32-bit integers, setting one bit for each sample. I've chosen to implement these as 32-bit integers instead of 64-bit integers because I assume that they will be easier to auto-vectorize (unverified). The length of a bit array is the number of possible items floor divided by the size of the unsigned integer type used in the array (plus one more unsigned integer if the remainder of the division is > 1).

Another benefit of storing samples in this way is that we can efficiently perform intersection operations on two arrays of integers. We achieve this by performing a bitwise "and" on the chunks within the array (which seems to be autovectorized by gcc – see godbolt).

a	10000000000000000000000000000000	10000000000000000000000000000001
b	11111111111111111111111111111111	11111111111111111111111111110000

Table 1: **Examples of byte arrays** Case a represents an array for 64 samples, where sample 0, 32, 63 are present. Case b represents an array for 60 samples where all samples (0-59) are present. In both of these cases, there are two "sample chunks", which refer to the number of unsigned integers needed to store all of the samples.

3.2.2 Walking the tree diffs

In order to analyze the tree structure, we need to first build a small index of the parents, right children, and left/right siblings of each node. This prepares us to perform a preorder tree traversal in the next step of the algorithm. This index is built as we apply the diffs for the next tree.

3.2.3 Tree Traversal

After a set of tree diffs has been applied, we have a valid tree to analyze. We perform a single preorder traversal of the tree, where we track the path through the tree for each mutation. This traversal does not start at the root of the tree, but instead starts at the node above the uppermost mutation in the tree. The paths are stored in another bit array of size (num node chunks × num mutations). The number of node chunks is determined by the method described in 3.2.1. Memory usage scales by $\mathcal{O}(m \times n)$, where m is the number of mutations and n is the number of nodes. We only allocate this matrix per-tree, however, so the max memory usage is limited to the maximum number of mutations on a single tree.

Following the paths down from the mutations to the samples allows us to see which samples are child to the mutations in the tree. While we are tracing this path, when we reach a sample node, we add the sample node to our intermediate array of mutations × samples.

After we accumulate the samples under each mutation on our focal tree, we enumerate the alleles for each site, using the same algorithm employed in `get_allele_weights`, except that we're passing around sample bit arrays instead of node counts. The result of our routine (named `get_allele_samples`) gets stored in an output array that holds results for all mutations and all samples (sample sets are not considered at this stage).

A simple benchmark of chromosome 1 of the sgdp dataset shows that applying all tree diffs + tree traversals take 7.5 seconds and about 750M of memory.

Tree Traversal Algorithm

```
Allocate memory to hold this tree's data
node_paths = calloc (num_node_chunks * num_mutations, sizeof (* node_paths))
stack = malloc((1 + num_samples + num_edges) * sizeof (* stack))

Initialize the stack with the parent of the uppermost node containing a mutation
stack.push(parent[top_mut_node])
while !stack.empty() do
    node = stack.pop()
    for mut_id = 0 to num_mutations do
        path = node_paths[m]

        If the current node contains this mutation, store it for the node
        if node == mutation.node then
            path[node] = 1
            if node.is_sample then
                If the current node is a sample, store it for this mutation
                mutation_samples[node] = 1
            end
        end

        If this node's parent is in this mutation's path, then store the current node
        if parent[node] in path then
            path[node] = 1
            if node.is_sample then
                If the current node is a sample, store it for this mutation
                mutation_samples[node] = 1
            end
        end
    end

    Continue the postorder traversal (handles polysomies)
    u = right_child[node]
    while u $\neq$ NULL do
        stack.push(u)
        u = left_sib[u]
    end
end
```

3.3 Computing results

Given our matrix of alleles × samples, two focal sites, and our sample sets, we summarize the data into the desired statistics. We achieve this by iterating over all pairs of allele states for the two sites (skipping the ancestral allele state if polarized). For each pair, we compute the allele weights (*w_AB*, *w_Ab*, and *w_aB*), which get passed into the summary function. After we compute the stat result from the summary function, we normalize the statistic with the selected normalization function (see 4.1 for more on normalization). Throughout this process, we keep a running total for the pair of sites, storing the result in a scalar value.

Algorithm for computing stat results

```
Loop over pairs
first_allele = 1 if polarized else 0
for allele_left = first_allele to num_alleles_left do
    for allele_right = first_allele to num_alleles_right do
        Find all A/B samples for alleles
        A_samples = allele_samples[allele_left]
        B_samples = allele_samples[allele_right]
        AB_samples = intersect(A_samples, B_samples)
        for k = 0 to num_sample_sets do
            Intersect all A/B samples with the current sample set
            sample_set = sample_sets[k]
            sample_set_A_samples = intersect(A_samples, sample_set)
            sample_set_B_samples = intersect(B_samples, sample_set)
            sample_set_AB_samples = intersect(AB_samples, sample_set)
            weight[k] = (w_AB, w_Ab, w_aB)
        end
        Both of these functions operate at the sample set level
        stat_result = stat_func(weight, params)
        norm_result = norm_func(weight, params)
        for k = 0 to num_sample_sets do
            Store running total for sample set result in the current pair
            result[k] += stat_result[k] * norm_result[k]
        end
    end
end
```

4 Summary Functions

Table 4 provides an overview of the summary functions that we intend to implement as site and branch statistics. The normalization strategy is described in further detail in 4.1.

Statistic	Polarization	Normalization	Equation
D	Polarized	Total	$D = f_{ab} - f_a f_b$
D'	Unpolarized	Haplotype Weighted	$D' = \frac{D}{D_{max}}$
D^2	Unpolarized	Total	$D^2 = D^2$
D_z	Unpolarized	Total	$D_z = D(1-2f_a)(1-2f_b)$
π_2	Unpolarized	Total	$\pi_2 = f_a f_b (1-f_a)(1-f_b)$
r	Polarized	Total	$r = \frac{D}{\sqrt{f_a f_b (1-f_a)(1-f_b)}}$
r^2	Unpolarized	Haplotype Weighted	$r^2 = \frac{D^2}{f_a f_b (1-f_a)(1-f_b)}$

Where D_{max} is defined as:

$$D_{max} = \begin{cases} \min\{f_a(1-f_b), f_b(1-f_b)\} & \text{if } D >= 0 \\ \min\{f_a f_b, (1-f_b)(1-f_b)\} & \text{otherwise} \end{cases}$$

4.1 Normalization

In our testing of summary functions, we found that the appropriate normalization procedure can vary depending on the summary function. We've settled on three normalization procedures: "Haplotype Weighted", "Allele Frequency Weighted", and "Total".

4.1.1 Hapltype Weighted

In the "Haplotype Weighted" normalization method, we weight the statistics by the frequency of its haplotype. This is necessary in ratio statistics, such as D', r and r^2.

$$\sum_{i=1}^{n} \sum_{j=1}^{m} p(A_i B_j) F_{ij}$$

where F is the summary function and $p(A_i B_j)$ is the frequency of haplotype $A_i B_j$. This method can be found in [2].

4.1.2 Allele Frequency Weighted

In the "Allele Frequency" normalization method, we weight the statistics by the product of the allele frequencies. We do not currently implement any statistics with this normalization strategy, but felt it was useful to implement, for the sake of completeness.

$$\sum_{i=1}^{n} \sum_{j=1}^{m} p(A_i) p(B_j) F_{ij}$$

where F is the summary function and $p(A_i)p(B_j)$ is the product of the allele frequencies of the A_i allele and the B_j allele. This method is described in [1].

4.1.3 Total

In the "Total" normalization method, we simply divide by the number of haplotypes. If we're using a non-ratio statistic, this is likely the desired normalization stategy.

$$\frac{1}{(n - \mathbb{1}_p)(m - \mathbb{1}_p)} \sum_{i=1}^{n} \sum_{j=1}^{m} F_{ij}$$

where $\mathbb{1}_p$ is an indicator function conditioned on whether or not our statistic is polarized, n is the number of alleles in site a, and m is the number of alleles in site b.

4.2 Evaluation

To ensure the correctness of our implementation, we have devised a number of test scenarios that will produce data at the theoretical limits of the statistics we've implemented. Note that our validation is not exhaustive, but it is a reasonable starting point for the purposes of verifying the correctness of our normalization strategy and our C implementation. These results were first produced with a python prototype, which we verified to be correct before using it to validate the C code.

4.2.1 Test cases

Table 2 enumerates the various test cases that we are using to validate the correctness of our metrics. The subsequent sections refer to test cases by name. Each test case is a two-site state matrix with 8-9 samples. Each number in our matricies represents the enumerated allelic state for each sample. For example, in the corerlated case, sample 4 from site A has allelic state number 2 and sample 0 on site A has the ancestral allele, allelic state number 0.

Name	$\begin{pmatrix} SiteAAllele \\ SiteBAllele \end{pmatrix}$
Correlated	$\begin{pmatrix} 0 & 1 & 1 & 0 & 2 & 2 & 1 & 0 & 1 \\ 1 & 2 & 2 & 1 & 0 & 0 & 2 & 1 & 2 \end{pmatrix}$
Uncorrelated	$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \end{pmatrix}$
Correlated Biallelic	$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$
Uncorrelated Biallelic	$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$
Repulsion Biallelic	$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$

Table 2: **Test cases for validating statistics.** In each case, we have an A and B site, representing the two sites under consideration for computaiton of our statistics.

4.2.2 Results

Here, we provide results for a few common statistics, exercising the diversity of strategies for normalization and polarization. The biallelic cases are useful for verifying the correctness of r and D. We also obtain the expected results in multiallelic cases in r^2.

Name	Result
Correlated	$\begin{pmatrix} 0.0556 & -0.01851 \\ -0.0185 & 0.0432 \end{pmatrix}$
Uncorrelated	$\begin{pmatrix} 0.0556 & 0.0 \\ 0.0 & 0.0556 \end{pmatrix}$
Correlated Biallelic	$\begin{pmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{pmatrix}$
Uncorrelated Biallelic	$\begin{pmatrix} 0.25 & 0.0 \\ 0.0 & 0.25 \end{pmatrix}$
Repulsion Biallelic	$\begin{pmatrix} 0.25 & -0.25 \\ -0.25 & 0.25 \end{pmatrix}$

Table 3: **Validation results for** D D is polarized and weighted by the total number of haplotypes

Name	Result
Correlated	$\begin{pmatrix} 0.2610 & -0.1221 \\ -0.1221 & 0.1838 \end{pmatrix}$
Correlated Biallelic	$\begin{pmatrix} 1. & 1. \\ 1. & 1. \end{pmatrix}$
Uncorrelated	$\begin{pmatrix} 0.25 & 0. \\ 0. & 0.25 \end{pmatrix}$
Uncorrelated Biallelic	$\begin{pmatrix} 1. & 0. \\ 0. & 1. \end{pmatrix}$
Repulsion Biallelic	$\begin{pmatrix} 1. & -1. \\ -1. & 1. \end{pmatrix}$

Table 4: **Validation results for** r**.** r is polarized and weighted by the total number of haplotypes.

Name	Result
Correlated	$\begin{pmatrix} 1. & 1. \\ 1. & 1. \end{pmatrix}$
Correlated Biallelic	$\begin{pmatrix} 1. & 1. \\ 1. & 1. \end{pmatrix}$
Uncorrelated	$\begin{pmatrix} 1. & 0. \\ 0. & 1. \end{pmatrix}$
Uncorrelated Biallelic	$\begin{pmatrix} 1. & 0. \\ 0. & 1. \end{pmatrix}$
Repulsion Biallelic	$\begin{pmatrix} 1. & 1. \\ 1. & 1. \end{pmatrix}$

Table 5: **Validation results for r^2.** r^2 is unpolarized and normalized by haplotype weighting.

5 Prototypes

The prototype code repo can be found here. It contains python/c/jupyter notebooks, which were used to prototype/test/benchmark/validate the approach.

The work described above has been split into two goals: 1. ensure mathematical correctness 2. ensure algorithmic efficiency/fit within current tskit framework. To achieve 1, a python prototype was made and unit tested for mathematical correctness. The core of that prototype can be found here and the unit tests can be seen here. The C prototype code is much more feature complete and can be found in this directory (described below).

File	Description
prototype.c	Main algorithm and supporting bit array / normalization code. Also contains a small outer wrapper for running the code from a tree file
summary_functions.c	Summary function code for our 7 two-site statistics
main.c	Simple command line interface for testing out the prototype
test_prototype.c	Unit testing for the bit array and sample manipulation code
test_stats.c	End to end style tests, taking tree files as an input and verifying the results output

Table 6: **Breakdown of C source code in our prototype implementation.** The first column links to the code directly.

References

[1] Philip W Hedrick. Gametic disequilibrium measures: proceed with caution. *Genetics*, 117(2):331–341, 1987.

[2] Honghua Zhao, D Nettleton, and Jack CM Dekkers. Evaluation of linkage disequilibrium measures between multi-allelic markers as predictors of linkage disequilibrium between single nucleotide polymorphisms. *Genetics Research*, 89(1):1–6, 2007.