Two Locus General Statistics in tskit

Lloyd Kirk, Ragsdale Lab

April 25, 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

For each summary function we intend to implement, we will have an outer wrapper for that function. For example, for the r^2 statistic, we 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 (which will remain untouched for our purposes), 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 some more conditionals in the form of `tsk_flags_t`.

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

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

3.2 Two Site Statistics

3.2.1 Node counting to compare pairs of sites

We are considering two algorithms for traversing the trees and storing haplotype counts. One is more CPU intensive and the other is more memory intensive. Here, I will outline both for consideration:

Option 1 (memory intensive)

```
Loop over all trees trees, computing their state and storing in a matrix
state = calloc (num_nodes * num_trees, sizeof (* double))
for tree_index = 0 to num_trees do
    count_edges_under_nodes (tree, &state)
end
Compute two locus statistics for all combinations of sites (with replacement)
sites_subloop_start = 0 for site_index_left = 0 to num_sites_left do
    for site_index_right = sites_subloop_start to num_sites_right do
        compute_general_stat_two_site_result (site_index_left, site_index_right, state)
    end
    sites_subloop_start++
end
```

Using this strategy, we will end up with memory scaling of $\mathcal{O}(n \times m)$, where n is the number of edges in the tree sequence and m is the number of trees. The computational complexity should be roughly $\mathcal{O}(\binom{m}{2})$ where m is the number of sites.

Option 2 (CPU intensive)

```
Loop over pairs of trees and compute statistics after computing the left and right
  state for each tree pair
subloop_start = 0
for tree_index_left = 0 to num_trees do
    Count edges on the left tree
    count_edges_under_nodes (left_tree, &left_state)
    for tree_index_right = subloop_start to num_trees_right do
        Count edges on the right tree
        count_edges_under_nodes (right_tree, &right_state)
        Compute two locus statistics for all combinations of sites within left and right
          tree (with replacement)
        sites_subloop_start = 0 for site_index_left = 0 to num_sites_left do
            for site_index_right = sites_subloop_start to num_sites_right do
                compute_general_stat_two_site_result (site_index_left, site_index_right,
                  left_state, right_state)
            end
            sites_subloop_start++
        end
    end
    tree_index_right++
end
```

In this approach, we end up with memory scaling of $\mathcal{O}(2 \times n)$, where n is the number of edges in a given tree. The computational scaling of this approach is roughly $\mathcal{O}(\binom{m}{2} \times \binom{n}{2})$ where m is the number of sites and n is the number of trees.

Without profiling, I am not sure how these approaches behave under different topologies, but I would lean towards **Option 1** unless there are strict memory requirements for tskit. In any case, comparing these two algorithms would not be difficult. I'm also open to suggestions here.

3.2.2 Two-Site General Stats Interface

We will provide a similar interface to computing `tsk_treeseq_site_general_stat`, with a different tree traversal algorithm (see Section 3.2.1), providing haplotype counts instead of site counts (see Section 3.3.1). In our implementation, we will use `get_allele_weights` to compute the allele counts for our left and right site. We will then pass the allele states for the lefthand site and the righthand site into `compute_general_stat_two_site_result` (described in Section 3.3).

```
static int
tsk_treeseq_two_site_general_stat(
        const tsk_treeseq_t *self,
        tsk_size_t state_dim,
        const double *sample_weights,
        tsk_size_t result_dim,
```

```
        general_stat_func_t *f,
        void *f_params,  // We can use sample_count_stat_params_t
        tsk_size_t num_windows,
        const double *windows,
        tsk_flags_t options,
        double *result
)
```

3.3 Computing results

The two site result function will accept the allele states from two sites as parameters. This means that we can avoid running `get_allele_weights` multiple times for a single site. Our two site results function will compute the counts of haplotypes for a given pair of sites and pass these haplotype counts to our summary functions.

```
static int
compute_general_stat_two_site_result(
        double *site_a_allele_states,
        double *site_b_allele_states,
        tsk_size_t num_a_alleles,
        tsk_size_t num_b_alleles,
        tsk_size_t allele_state_dim,
        tsk_size_t result_dim,
        general_stat_func_t *f,
        void *f_params,
        double *total_weight,
        bool polarised,
        double *result,
)
```

3.3.1 Haplotype Counting Algorithm

We use pairs of 1-d row vectors (allele_states in the previous section) to describe the terminal allelic state in a given pair of sites. For example, if we're given a tree sequence like the one shown in Figure 1, we will loop over the combinations of sites and compare them in pairs: (0, 0), (0, 1), (1, 1). In the portion of the loop where we're comparing site 0 and site 1, the lefthand vector will be $\begin{pmatrix} 2 & 2 & 3 & 3 & 1 & 1 & 1 & 1 \end{pmatrix}$ and the righthand vector will be $\begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 2 & 0 \end{pmatrix}$. When we compute the haplotype counts for the comparison of site 0 and site 1, we get a matrix with dimensions n_alleles_A×n_alleles_B. In our case 4×3. The resulting haplotype count matrix for our example is:

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 3 \\ 0 & 2 & 0 \\ 0 & 2 & 0 \end{pmatrix}$$

To produce these counts, we address the haplotype matrix with the enumerated allelic state as the x and y coordinate and increment the count by one for each that we encounter (very similar to the algorithm in the python prototype).

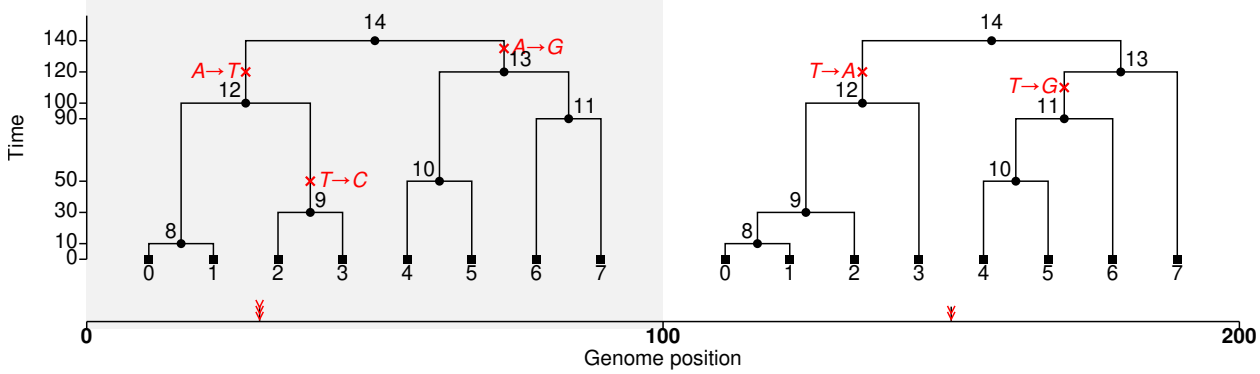

Figure 1: **An example of a tree sequence used to compute two-locus statistics.** The left tree has one site with four states and the right tree has one site with three states.

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 4.2.

Statistic	Polarization	Normalization	Equation
D	Polarized	Total	$D = f_{ab} - f_a f_b$
D'	Polarized	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	Haplotype Weighted	$r = \frac{D}{\sqrt{f_a f_b (1-f_a)(1-f_b)}}$
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 Summary Function Signature

Two locus statistics need to know the number of AB, Ab, and aB haplotypes. They also need to know the total number of haplotypes being considered in order to properly convert the counts of each haplotype to proportions.

```
static int
summary_func(int w_AB, int w_Ab, int w_aB, int n)

two_site_summary_func(
        tsk_size_t state_dim,
        const double *state,
        tsk_size_t TSK_UNUSED(result_dim),
        double *result,
        void *params
)
```

4.2 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 two normalization procedures: "Haplotype Weighted" and "Total".

4.2.1 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.

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

where F is the summary function and $p(A_iB_j)$ is the frequency of haplotype A_iB_j. This method can be found in [1]. We apply this

4.2.2 Total

In the "Total" normalization method, we simply divide by the number of haplotypes that we've visited. 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.

4.3 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 will be useful for our C implementation.

4.3.1 Test cases

Table 1 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.

Name	$\begin{pmatrix} SiteA \\ SiteB \end{pmatrix}$
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 1: **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.

4.3.2 Polarized

We'll begin with the polarized statistics. D and D' sum to zero when they are unpolarized, so we are only providing a polarized method to compute them. The correctness of polarized D is best verified with bi-allelic sites. We can use the unpolarized statistics to verify the correctness of D in multi-allelic scenarios. These results tell us that our normalization strategy is correct and that D has been properly implemented.

Name	Result
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	$\begin{pmatrix} 0.25 & -0.25 \\ -0.25 & 0.25 \end{pmatrix}$
Correlated (unpolarized)	$\begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix}$

Table 2: **Validation results for** D All results are polarized unless marked unpolarized.

Name	Result
Correlated Biallelic	$\begin{pmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{pmatrix}$
Uncorrelated Biallelic	$\begin{pmatrix} 0.5 & 0.0 \\ 0.0 & 0.5 \end{pmatrix}$
Repulsion Biallelic	$\begin{pmatrix} 0.5 & 0.0 \\ 0.0 & 0.5 \end{pmatrix}$
Correlated (unpolarized)	$\begin{pmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{pmatrix}$
Unorrelated (unpolarized)	$\begin{pmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{pmatrix}$

Table 3: **Validation results for r.** All results are polarized unless marked unpolarized.

References

[1] 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.