Reconstructing Trees with Cassiopeia

Contents

1 Introduction

In this document, we'll provide you with more in depth discussion for each of the parameters for tree reconstruction with Cassiopeia (from the *reconstruct-lineage* command line tool) and motivate these parameters with real examples. This document will be somewhat redundant with our existing documentation website but still will hopefully provide you with a better foundation for getting your analysis started faster. Importantly, this document does not contain information regarding generally how tree inference is performed in Cassioepia; for this, we suggest you read our recently published manuscript.

To make tree inference and using Cassiopeia as easy as possible, we've provided a command line interface for tree reconstruction. This means that as long as you've followed the instructions to download and install Cassiopeia, you'll have at your disposal a set of commands that you can call directly from the command line - there is no need to enter into a Python session to run Casssiopeia. The command that we're concerned with specifically here is *reconstruct-lineage*.

The *reconstruct-lineage* command line tool has several parameters that you can be specified, giving the user exquisite control over how exactly the reconstruction will be performed. Undoubtedly, this also creates quite a confusion around which are fundamentally necessary and which are for advanced users only.

2 The Basics: Cassiopeia-Greedy

The *reconstruct-lineage* tool takes, at a minimum, three items:

1. A character matrix, as created with the `cassiopeia.utilities.alleletable_to_character_matrix` functionality.
2. An algorithm to run: Cassiopeia-Greedy, -Hybrid, -ILP, Neighbor-Joining, or Camin-Sokal.
3. An output path, relative to your current directory. **Note: this is assumed to be a text file! Cassiopeia be default also writes out a *pickle* file, storing the Cassiopeia-Tree.**

Taken together, the most simple command possible would look something like this:

```
reconstruct-lineage my_character_matrix.txt -greedy my_tree.nwk
```

This will run Cassiopeia-Greedy, in its most simple invocation, on the character matrix `my_character_matrix.txt` and output the result in `my_tree.nwk`.

2.1 Advanced Greedy Usage

There a few parameters that are useful to incorporate into Cassiopeia-Greedy, especially with respect to missing data. To handle missing data, we have two parameters that can be used:

1. `greedy_max_missing_rep`: This tells Cassiopeia-Greedy the maximum proportion of missing data that will tolerated when considering a Greedy split. If it is exceeded, the character will not be considered. By default this is 1.0 (meaning any character can be used as a character split), but we've found success reducing this around 0.3. You can invoke it using `-greedy_max_missing_rep 0.3`, or whatever threshold you'd like to use.
2. `greedy_missing_data_mode`: This tells Cassiopeia-Greedy how to handle missing data during character splits - i.e. this helps the algorithm answer the question "Which side does this cell with the missing value belong to? The side with character-state or the side without?" We have three methods for addressing this: *avg*, *knn*, and *lookahead*. Both *KNN* (controlled additionally with the `num_neighbors` parameter) and the *lookahead* (controlled additionally with the `greedy_lookahead_depth` parameter) methods are experimental though, and should be used wisely. **We suggest sticking to *avg* for now.**

Thus, to incorporate these new parameters into the Cassiopeia-Greedy run, you can use a command like this:

```
reconstruct-lineage my_character_matrix.txt -greedy my_tree.nwk -greedy_max_missing_rep
0.3 -greedy_missing_data_mode knn -num_neighbors 10
```

Or,

```
reconstruct-lineage my_character_matrix.txt -greedy my_tree.nwk -greedy_max_missing_rep
0.3 -greedy_missing_data_mode avg
```

2.2 Incorporating Prior Probabilities

The first improvement one may want to make is to improve the way that Cassiopeia-Greedy chooses the characters on which to split. In its most simple implementation, this is done simply by the *frequency* of a given character-state at the leaves; of course this frequency is confounded, potentially, by the recurrent & independent mutation of that character-state. In this case, we can try to penalize a character-state if it is very likely to occur, as determined by our *state-priors*.

Indel priors can be calculated across several independent clonal populations using the `cassiopeia.utilities.get_indel_props` to produce a table relating each unique indel to the number of times it arose *at least once* across several independent clones. This data table can in turn be fed into `cassiopeia.utilities.alleletable_to_character_matrix` to create *state-priors* for a given character matrix. **Note: though these state-priors are derived from general indel priors, this dictionary *only relates to the character matrix it was formed with.*** For more details on how to calculate these items, refer to our tutorial notebook.

To add priors to the command line call, you can do so like this:

```
reconstruct-lineage my_character_matrix.txt -greedy my_tree.nwk -mutation_map
my_character_matrix_priors.pkl
```

3 Using Cassiopeia-ILP

In smaller datasets, you have the opportunity to solve the maximum parsimony problem more precisely and thus can use Cassiopeia-ILP. This module attempts to find a Steiner-Tree over all reasonable unobserved evolutionary intermediates (summarized in a *Potential Graph*) using a exhaustive optimization technique called *Integer Linear Programming (ILP)*.

The most basic invocation of Cassiopeia-ILP looks very similar to that of Cassiopeia-Greedy, though requires you to specify a maximum time allowed for convergence and the maximum size of the potential graph to infer; our default runs look like this:

```
reconstruct-lineage my_character_matrix.txt -ilp my_tree.nwk -time_limit 12600
-max_neighborhood_size 10000
```

3.1 How & when to change parameters in Cassiopeia-ILP

There are two failure points to pay particular attention to for Cassiopeia-ILP: the complexity of the potential graph (the data structure we're trying to solve a Steiner Tree on) and the convergence.

3.1.1 Maximum Neighborhood Size

The complexity of the potential graph can be measured by the number of nodes added to each layer of the Potential Graph (for more information on this, please see our manuscript, linked above). If too many nodes are being added to the layer, the algorithm aborts and either completes inference using Cassiopeia-Greedy or takes a potential graph that is only partially solved. Particularly, the algorithm aborts if the size of the layer exceeds the user-input `max_neighborhood_size` Compared to the scenario where analysis would end, this early abortion might make sense but surely we could do better either by increasing the `max_neighborhood_size` parameter or running Cassiopeia-Hybrid (described below).

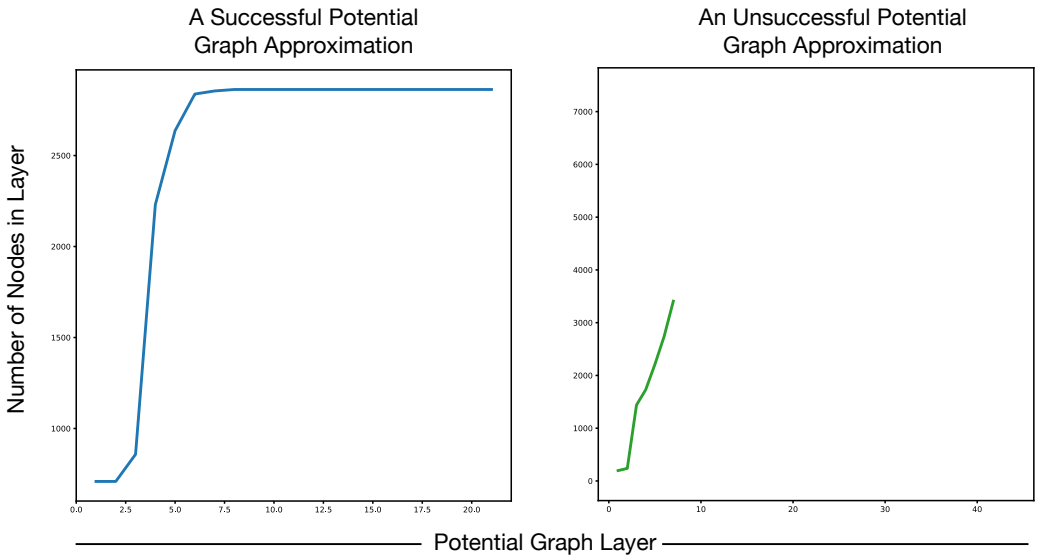

Figure 1: A successful and unsuccessful spaghetti test. Two different trees, both reconstructed with a maximum neighborhood size of 10000.

The way we normally attempt to diagnose these issues is with the informal **"spaghetti test"** in which we observe the diagnostic plot of number of nodes added to the graph as a function of the layer. Observe the plot shown in Figure 1. In this figure, we contrast what a "successful" reconstruction looks like, which is characterized by a saturation of the number of nodes added in deeper layers, against an unsuccessful one, in which the number of nodes that need to be added to a layer quickly explodes outside of our maximum neighborhood size constraint.

One may ask why not just set the maximum neighborhood size to something extreme like 50,000? For two reasons, we think 10,000 is a reasonable number. First, our inference of the potential graph is relatively slow and so going to larger numbers require quite large CPU times. Secondly, we find that the optimization problem becomes less efficient on larger potential graphs. For this reason, we suggest that if you can't infer a suitable potential graph on a given dataset with between a maximum neighborhood size of 8,000 and 15,000 you should consider running Cassiopeia-Hybrid.

3.1.2 Time Limit

The second parameter that can be used to control how well Cassiopeia-ILP reconstructs a tree is the time to convergence, controlled with the `time_limit` parameter. Because solving an ILP is NP-Complete, the problem could potentially take forever to complete. As a result, we suggest setting a time limit on the time that the ILP is allowed to run - normally around 3.5hr but this can of course be increased. If you notice that the ILP has not fully converged by the end of 3.5hr (see below for a tutorial on how to read Cassiopeia-ILP output) you can increase the time limit. **Note: the time limit specified for Cassiopeia-ILP is always in seconds!**

3.2 Performing Weighted Parsimony Optimization

Because parsimony can break down in the case where there are heavily imbalanced probabilities of certain states arising, Cassiopeia supports a "weighted" optimization. By using the `weighted_ilp` parameter, we can weight the edges of the Potential Graph by the negative log probability of the

mutations that occur along that edge. This means that the optimizer will prefer to add edges to the Steiner Tree if they contain mutations that are less likely to have occurred and thus more information-rich. To invoke this, you can use a command that looks like this:

```
reconstruct-lineage my_character_matrix.txt -ilp my_tree.nwk -mutation_map
my_character_matrix_priors.pkl -time_limit 12600 -max_neighborhood_size 10000
-weighted_ilp
```

3.3 Analyzing output from Cassiopeia-ILP

Cassiopeia-ILP's output can be quite difficult to interpret, but if looked at correctly it will give you crucial insight into how the reconstruction fared. For the purposes of the discussion here, consider the output in Figure 2. An ILP log begins with information regarding the inferred phylogenetic root and number of samples that are being processed before logging the complexity of the potential graph as a function of the layer (described in how far the latest common ancestor (LCA) is from the current group of cells). Once this is completed, an ILP problem will be launched. **Note: Before moving on, always be sure to observe that the potential graph was able to be solved to a reasonable depth; foe example, in Figure 2 the potential graph went to a depth of 21.**

At this point, the log now is produced by Gurobi not by Cassiopeia and so there is a lot of information that does not necessarily pertain to what a user might be most interested in. While you can read up on what exactly each piece of information means here, we highlight 5 key pieces of information: how to see how the problem is converging, the time elapsed during the solving, whether or not the problem was able to converge, and what the final convergence was.

4 Cassiopeia-Hybrid

By this point, most of the tools necessary for using Cassiopeia-Hybrid have already been described because this algorithm is a combination of Cassiopeia-ILP and Cassiopeia-Greedy. Thus, any of the parameters previously described apply directly to Cassiopeia-Hybrid as well. The only additional parameters that must be added for Cassiopeia-Hybrid are the number of threads desired (`num_threads`) and the `cutoff` parameter - in other words, when to transition from Cassiopeia-Greedy to -ILP.

We offer two criteria-regimes for switching between Cassiopeia-Greedy and -ILP: the number of cells and the maximum distance to the latest common ancestor (LCA) of a group of cells. If you'd like to transition to ILP for groups with fewer than 200 cells, you can use a command that looks like this:

```
reconstruct-lineage my_character_matrix.txt --hybrid my_tree.nwk --time_limit 12600
--max_neighborhood_size 10000 --cutoff 200 --num_threads 10
```

On the other hand, if you'd like to transition when the most distant cell from the LCA of a group is a hamming distance of at most 20, you can use a command that looks like this:

```
reconstruct-lineage my_character_matrix.txt --hybrid my_tree.nwk --time_limit 12600
--max_neighborhood_size 10000 --cutoff 20 --hybrid_lca_mode --num_threads 10
```

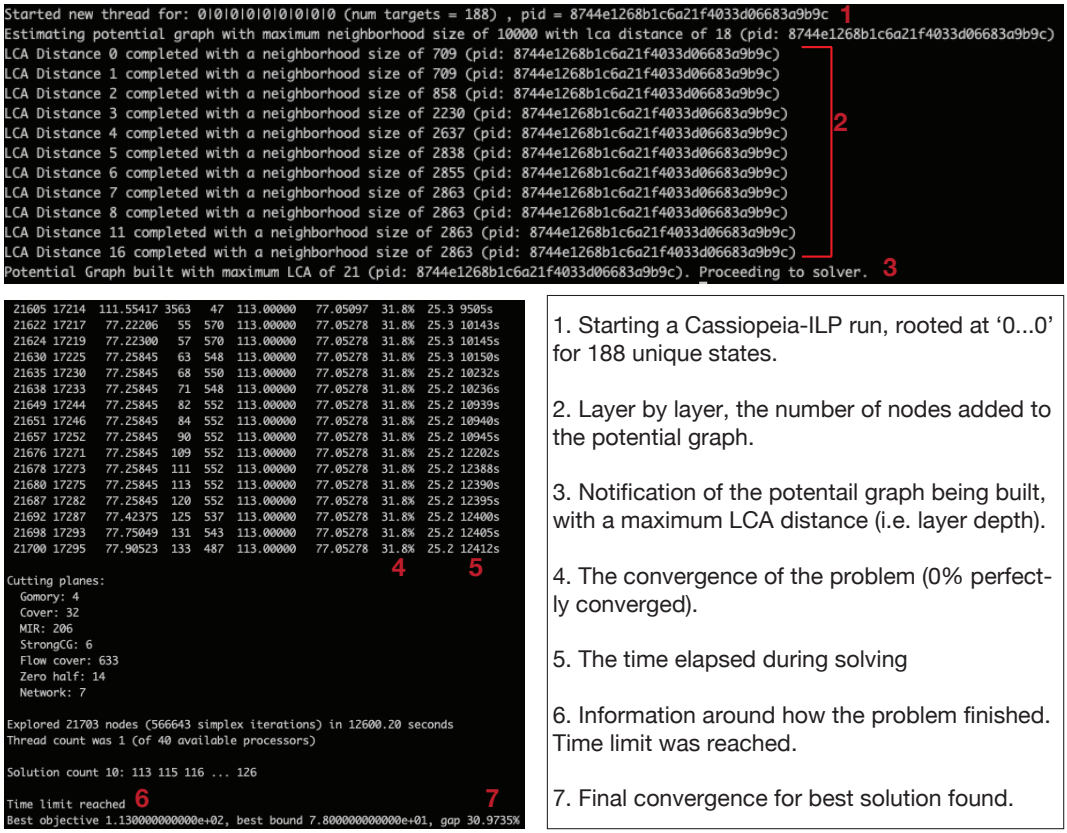

Figure 2: Example of ILP logging information.

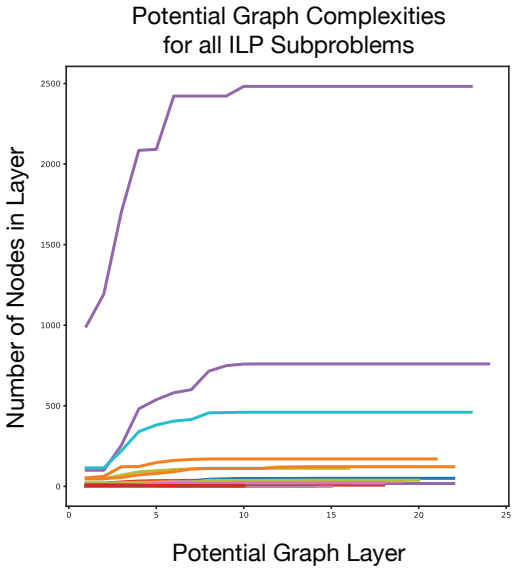

Figure 3: Potential Graph complexities for each subproblem from a given Cassiopeia-Hybrid run.

A single hybrid run will spawn off multiple Cassiopeia-ILP subproblems and thus the resulting "spaghetti-plot" will have several lines on it (ex. Figure 3). **On these plots, you'll want to make sure that *all* of the sub-problems successfully solved the potential graph, else**

you should re-run Cassiopeia-Hybrid with a lower cutoff.

Another consideration is that the output log will be interleaved between each sub-problem. To track more carefully the development of a given subproblem, you can grep the log by the process id, as illustrated in Figure **??**.

Figure 4: Restructuring output from Cassiopeia-Hybrid logging.

5 Recommended Cassiopeia Settings

By this point, you should be familiar with how to control all of the parameters in Cassiopeia. So taken together, you might be asking what's the best set of parameters to use? Well, this will inevitably depend on the dataset and we are still working on ways for predicting the best parameters given summary statistics that are easy to infer from the input data. However, there are some invariant that we suggest users start with: for example, **users should always use state priors**. Our internal default settings, for pretty much any dataset is:

```
reconstruct-lineage my_character_matrix.txt --hybrid my_tree.nwk --mutation_map
my_character_matrix_priors.pkl --time_limit 12600 --max_neighborhood_size 10000
--cutoff 20 --hybrid_lca_mode --greedy_missing_data_mode avg --greedy_max_missing_rep
0.3 --num_threads 10 --verbose > _log.stdout 2> _log.stderr
```

What's nice about this invocation is that it stores the standard and error logs as the algorithm goes so you can constantly check in. At the end of the run, we suggest you take a look at the potential graph diagnostic plot and if the problems were too complex, run the reconstruction algorithm again with a lower cutoff or with a higher maximum neighborhood size (the former is more effective, in our experience).