drep Documentation

Release 2.0.0

Matt Olm

Feb 06, 2023

Contents

1 Rapid and accurate comparison and de-replication of microbial genomes **1**

2 Contents **3**
- 2.1 Overview . . . 3
- 2.2 Installation . . . 6
- 2.3 Quick Start . . . 7
- 2.4 Example Output . . . 8
- 2.5 Important Concepts . . . 14
- 2.6 User manual . . . 21
- 2.7 Advanced Use . . . 28
- 2.8 dRep API . . . 32

Python Module Index **43**

Index **45**

CHAPTER 1

Rapid and accurate comparison and de-replication of microbial genomes

The publication is available at ISME and an open-source pre-print is available on bioRxiv.

Source code is available on GitHub.

See links to the left for *Installation* and *Quick Start* instructions

Comments and suggestions can be sent to Matt Olm

Bugs reports and feature requests can be submitted through GitHub.

CHAPTER 2

Contents

2.1 Overview

dRep is a python program which performs rapid pair-wise comparison of genome sets. One of it's major purposes is for genome de-replication, but it can do a lot more.

The publication is available at ISME and an open-source pre-print is available on bioRxiv.

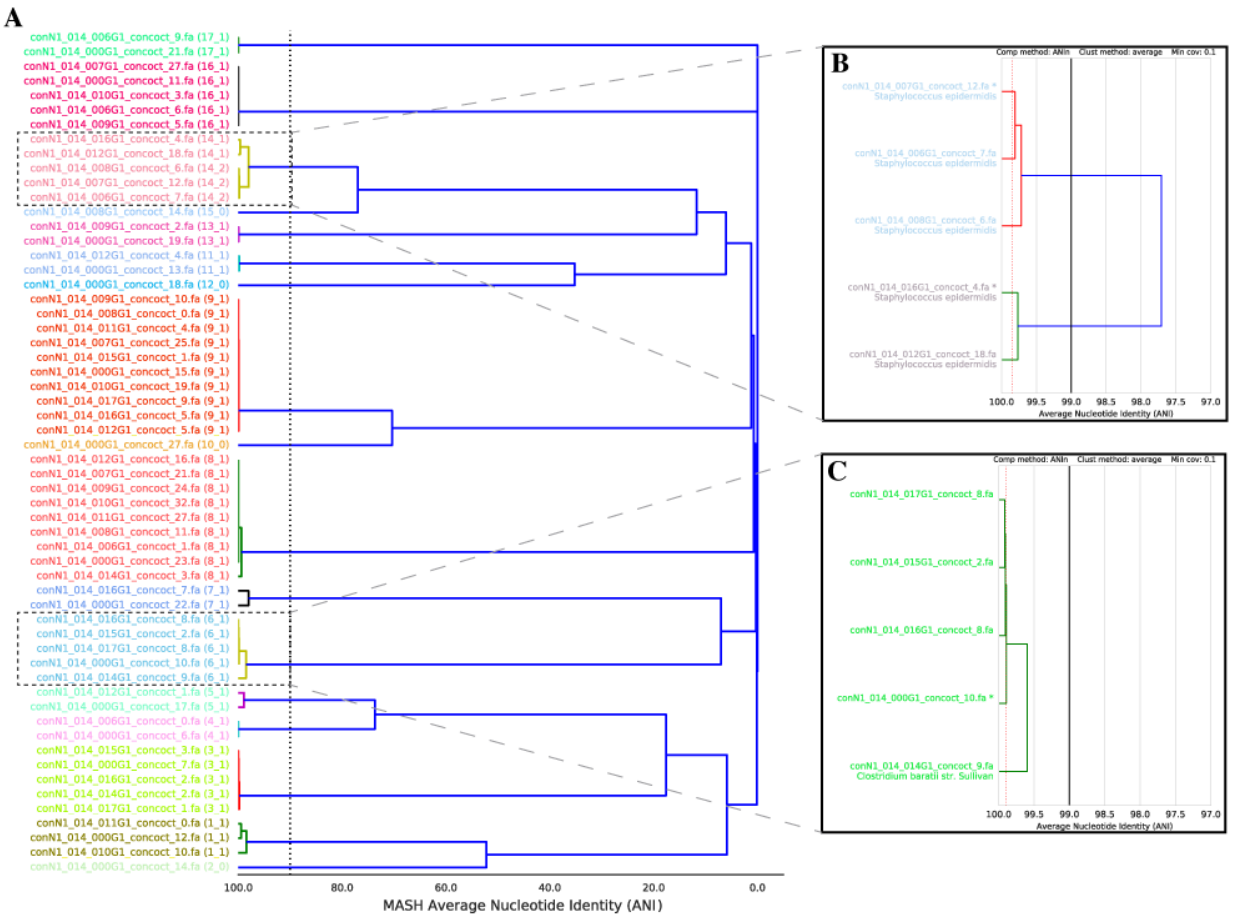
Source code is available on GitHub.

2.1.1 Genome comparison

dRep can rapidly and accurately compare a list of genomes in a pair-wise manner. This allows identification of groups of organisms that share similar DNA content in terms of Average Nucleotide Identity (ANI).

dRep performs this in two steps- first with a rapid primary algorithm (Mash), and second with a more sensitive algorithm (ANIm). We can't just use Mash because, while incredibly fast, it is not robust to genome incompletenss (see *Important Concepts*) and only provides an "estimate" of ANI. ANIm is robust to genome incompleteness and is more accurate, but too slow to perform pair-wise comparisons of longer genome lists.

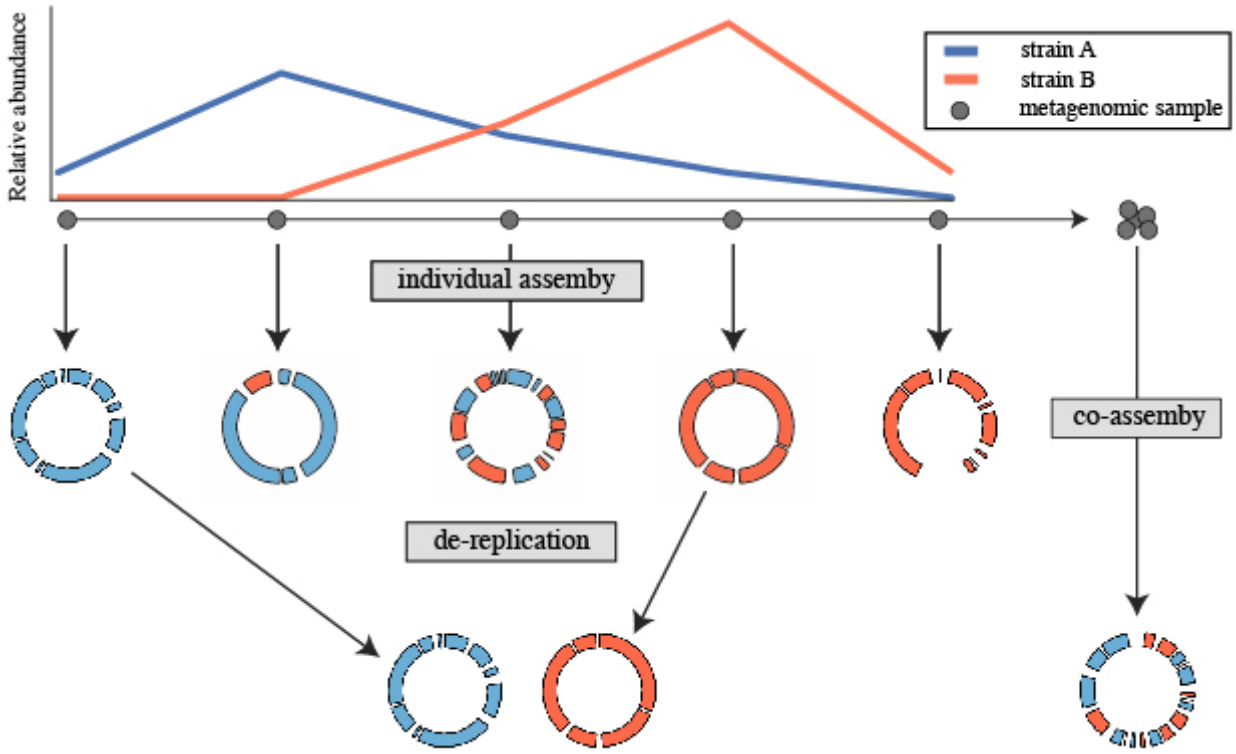
dRep first compares all genomes using Mash, and then only runs the secondary algorithm (ANIm or gANI) on sets of genomes that have at least 90% Mash ANI. This results in a great decrease in the number of (slow) secondary comparisons that need to be run while maintaining the sensitivity of ANIm.



2.1.2 Genome de-replication

De-replication is the process of identifying sets of genomes that are the "same" in a list of genomes, and removing all but the "best" genome from each redundant set. How similar genomes need to be to be considered "same", how to determine which genome is "best", and other important decisions are discussed in *Important Concepts*

A common use for genome de-replication is the case of individual assembly of metagenomic data. If metagenomic samples are collected in a series, a common way to assemble the short reads is with a "co-assembly". That is, combining the reads from all samples and assembling them together. The problem with this is assembling similar strains together can severely fragment assemblies, precluding recovery of a good genome bin. An alternative option is to assemble each sample separately, and then "de-replicate" the bins from each assembly to make a final genome set.



The steps to this process are:

- Assemble each sample separately using your favorite assembler. You can also perform a co-assembly to catch low-abundance microbes
- Bin each assembly (and co-assembly) separately using your favorite binner
- Pull the bins from all assemblies together and run dRep on them
- Perform downstream analysis on the de-replicated genome list

---

Note: See the publication To Dereplicate or Not To Dereplicate? for an outside perspective on the pro's and con's of dereplication

---

2.1.3 A dRep based metagenomic workflow

One method of genome-resolved metagenomic analysis that I am fond of involves the following steps:

1) Assemble and bin all metagenomic samples to generate a large database of metagenome-assembled genomes (MAGs)
2) Dereplicate all of these genomes into a set of species-level representative genomes (SRGs). This is done with dRep using a 95% ANI threshold.
3) Create a mapping database from all SRGs, and map all metagenomes to this database. Each sample will now be represented by a *.bam* file mapped against the same database.
4) Profile all .bam files using inStrain. This provides a huge number of metrics about the species-level composition of each metagenomes.
5) Preform strain-level comparisons between metagenomes using inStrain

Some example publications in which this workflow is used are here and there.

2.2 Installation

2.2.1 Using pip

To install dRep, simply run

```
$ pip install drep
```

OR

```
$ git clone https://github.com/MrOlm/drep.git

$ cd drep

$ pip install .
```

That's it!

Pip is a great package with many options to change the installation parameters in various ways. For details, see pip documentation

2.2.2 Using conda

To install dRep with conda, simply run

```
conda config --add channels bioconda; conda install drep
```

2.2.3 Dependencies

dRep requires other programs to run. Not all dependencies are needed for all operations

To check which dependencies are installed on your system and accessible by dRep, run

```
$ dRep check_dependencies
```

Near Essential

- Mash - Makes primary clusters (v1.1.1 confirmed works)
- MUMmer - Performs default ANIm comparison method (v3.23 confirmed works)

Recommended

- fastANI - A fast secondary clustering algorithm
- CheckM - Determines contamination and completeness of genomes (v1.0.7 confirmed works)
- gANI (aka ANIcalculator) - Performs gANI comparison method (v1.0 confirmed works)
- Prodigal - Used be both checkM and gANI (v2.6.3 confirmed works)

Accessory

- NSimScan - Only needed for goANI algorithm

- Centrifuge - Deprecated; not used anymore

Programs need to be installed to the system path, so that you can call them from anywhere on your computer.

Note: If you already have information on your genome's completeness and contamination, you can input that to dRep without the need to install checkM (see *Advanced Use*))

2.2.4 CheckM

CheckM is the program that dRep uses to calculate completeness and contamination. It's not required for all dRep commands, but if you'd like to de-dereplicate your genome set using completeness and contamination it is required. It takes a bit of work to install (including setting a root directory and downloading a reference database). Installation instructions can be found here

2.3 Quick Start

The functionality of dRep is broken up into modules. To see a list of the available modules, check the help:

```
$ dRep -h
                ..:::: dRep v3.0.0 ::::..

  Matt Olm. MIT License. Banfield Lab, UC Berkeley. 2017 (last updated 2020)

  See https://drep.readthedocs.io/en/latest/index.html for documentation
  Choose one of the operations below for more detailed help.

  Example: dRep dereplicate -h

  Commands:
    compare            -> Compare and cluster a set of genomes
    dereplicate        -> De-replicate a set of genomes
    check_dependencies -> Check which dependencies are properly installed
```

2.3.1 Dereplication

Dereplication is the process of identifying groups of genomes that are the "same" in a genome set and identifying the "best" genome from each set. How similar genomes need to be to be considered "same" and how the "best" genome is chosen are study-specific decisions that can be adjusted (see *Important Concepts*)

To dereplicate a set of genomes, run the following command:

```
$ dRep dereplicate outout_directory -g path/to/genomes/*.fasta
```

This will automatically de-replicate the genome list and produce lots of information about it.

See also:

Example Output to view example output

Important Concepts for guidance changing parameters

2.3.2 Genome comparison

dRep is able to perform rapid genome comparisons for a group of genomes and visualize their relatedness. For example:

```
$ dRep compare output_directory -g path/to/genomes/*.fasta
```

For help understanding the output, see *Example Output*

To change the comparison parameters, see *Important Concepts*

See also:

Example Output to view example output

Important Concepts for guidance changing parameters

2.4 Example Output

dRep produces a variety of output in the work directory depending on which operations are run.

To generate the figures below, dRep `dereplicate` was run on a set of 5 randomly chosen *Klebsiella oxytoca* isolate genomes as follows:

```
$ dRep dereplicate complete_only -g *.fna --S_algorithm gANI
```

See also:

Overview for general information on the program

Quick Start for more information on dereplicate_wf

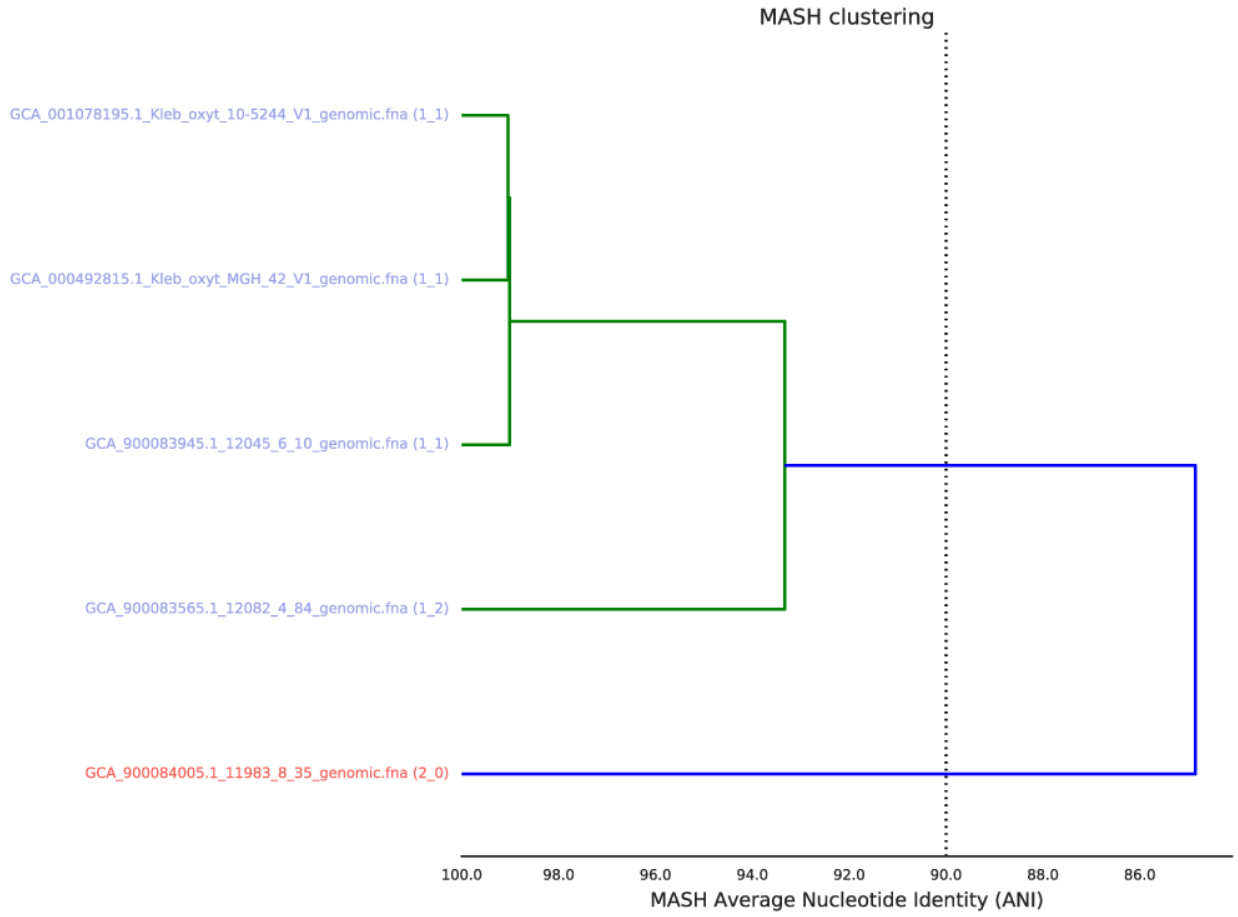
User manual for a more detailed description of what the modules do

2.4.1 Figures

Figures are located within the work directory, in the folder `figures`:

```
$ ls complete_only/figures/
Clustering_scatterplots.pdf
Cluster_scoring.pdf
Primary_clustering_dendrogram.pdf
Secondary_clustering_dendrograms.pdf
Winning_genomes.pdf
```

Primary_clustering_dendrogram

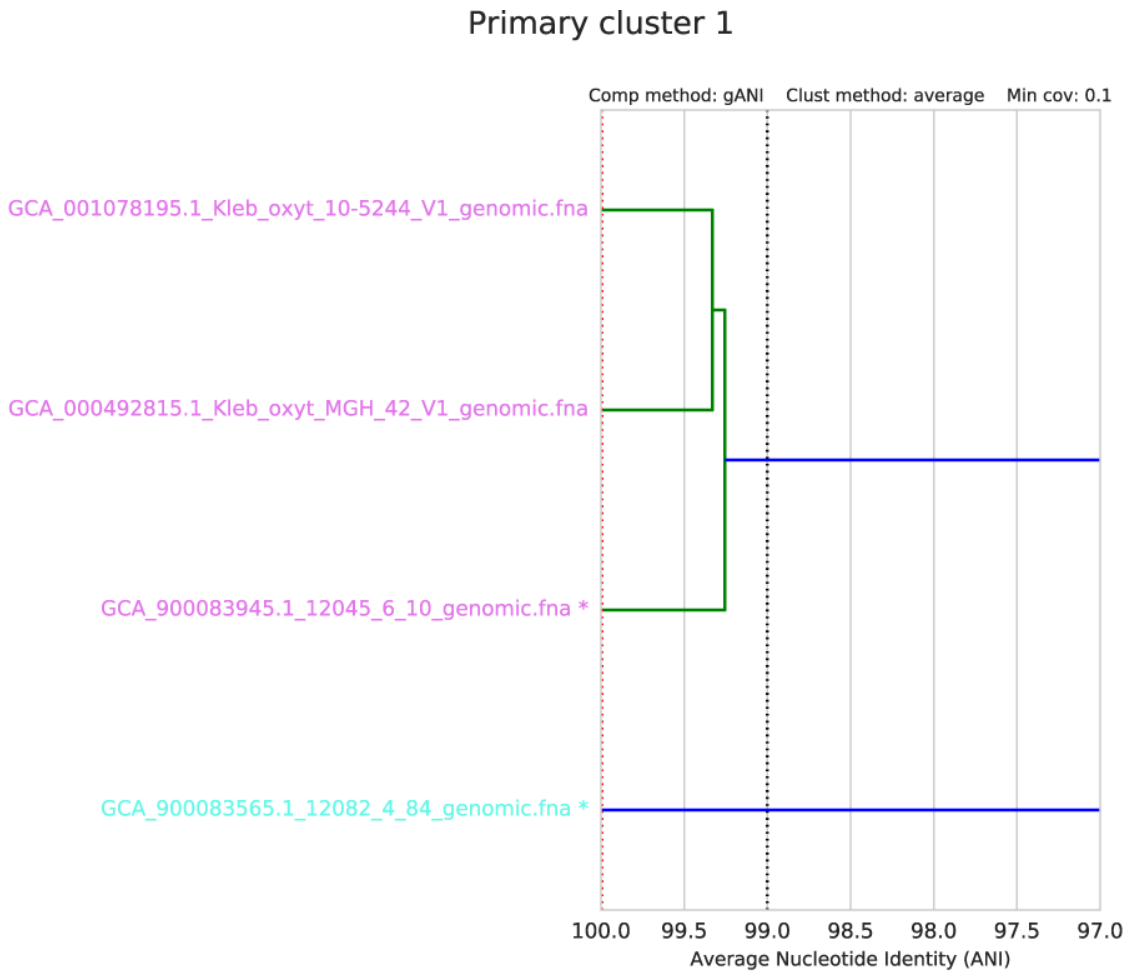


The primary clustering dendrogram summarizes the pair-wise Mash distance between all genomes in the genome list.

The dotted line provides a visualization of the **primary ANI** - the value which determines the creation of primary clusters. It is drawn in the above figure at 90% ANI (the default value). Based on the above figure you can see that two primary clusters were formed- one with genomes colored blue, and one red.

Note: Genomes in the same primary cluster will be compared to each other using a more sensitive algorithm (gANI or ANIm) in order to form secondary clusters. Genomes which are not in the same primary cluster will never be compared again.

Secondary_clustering_dendrograms

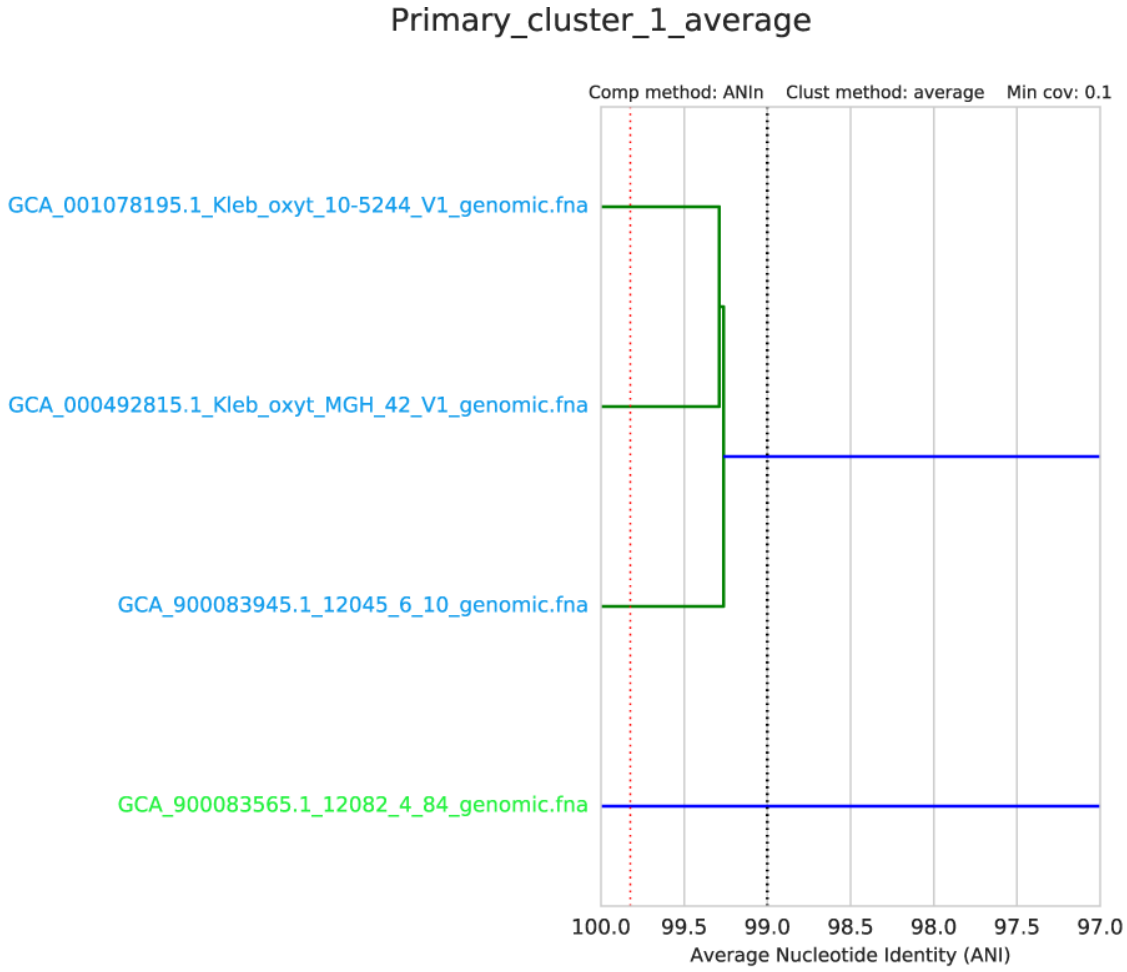


Each primary cluster with more than one member will have a page in the Secondary clustering dendrograms file. In this example, there is only one primary cluster with > 1 member.

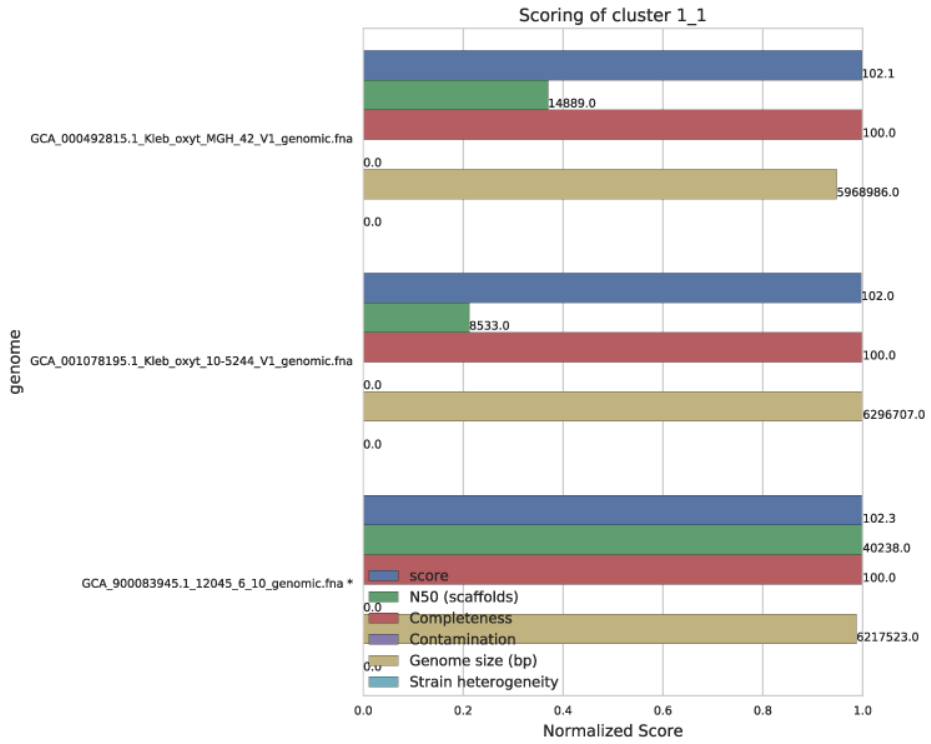
This dendrogram summarizes the pair-wise distance between all organisms in each primary cluster, as determined by the secondary algorithm (gANI / ANIm). At the very top the primary cluster identity is shown, and information on the secondary clustering algorithm parameters are shown above the dendrogram itself. You can see in the above figure that this secondary clustering was performed with gANI, the minimum alignment coverage is 10%, and the hierarchical clustering method is average.

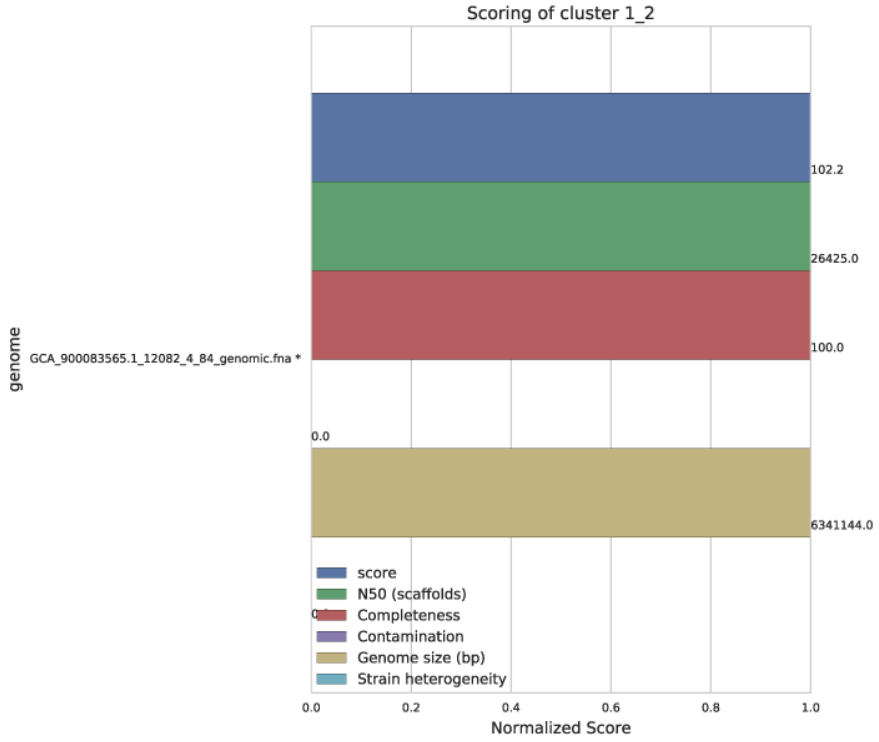
The black dotted line shows the **secondary clustering ANI** (in this case 99%). This value determines which genomes end up in the same secondary cluster, **and thus are considered to be the "same"**. In the above figure, two secondary cluster are formed. The "best" genome of each secondary cluster is marked with as asterisk.

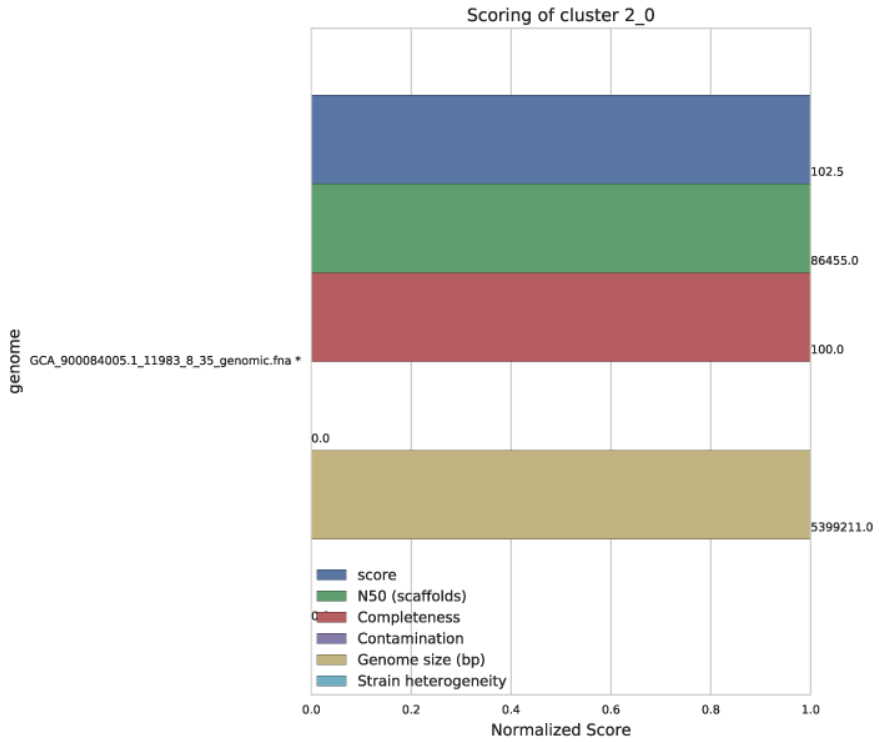
The red line shows the lowest ANI for a "self-vs-self" comparison of all genomes in the genome list. That is, when each genome in this primary cluster is compared to itself, the red line is the lowest ANI you get. This represents a "limit of detection" of sorts. gANI always results in 100% ANI when self-vs-self comparisons are performed, but ANIm does not (as shown in the figure below).


Primary_cluster_1_average
Comp method: ANIn
Clust method: average
Min cov: 0.1
GCA_001078195.1_Kleb_oxyt_10-5244_V1_genomic.fna
GCA_000492815.1_Kleb_oxyt_MGH_42_V1_genomic.fna
GCA_900083945.1_12045_6_10_genomic.fna
GCA_900083565.1_12082_4_84_genomic.fna
100.0
99.5
99.0
98.5
98.0
97.5
97.0
Average Nucleotide Identity (ANI)

Cluster_scoring




Scoring of cluster 1_2
genome
GCA_900083565.1_12082_4_84_genomic.fna *
102.2
26425.0
100.0
0.0
6341144.0
score
N50 (scaffolds)
Completeness
Contamination
Genome size (bp)
Strain heterogeneity
0.0
0.2
0.4
0.6
0.8
1.0
Normalized Score


Scoring of cluster 2_0
genome
GCA_900084005.1_11983_8_35_genomic.fna *
102.5
86455.0
100.0
0.0
5399211.0
score
N50 (scaffolds)
Completeness
Contamination
Genome size (bp)
Strain heterogeneity
0.0
0.2
0.4
0.6
0.8
1.0
Normalized Score

Each secondary cluster will have its own page in the Cluster scoring figure. There are three secondary clusters in this example- 2 of which came from primary cluster 1, and 1 of which is the only member of primary cluster 2.

These figures show the score of each genome, as well as all metrics that can go into determining the score. This helps the user visualize the quality of all genomes, and ensure that they agree with the genome chosen as "best". The "best" genome is marked with an asterisk, and will always be the genome with the highest score.

One genome will be selected from each secondary cluster to be included in the de-replicated genome set. So in the above example, we will have 3 genomes in the de-replicated genome set. This is because the algorithm decided that all genomes in cluster 1_1 were the "same", and chose GCA_900083945 as the "best".

See also:

See *User manual* for information on how scoring is done and how to change it

Other figures

Clustering scatterplots provides some information about genome alignment statistics, and **Winning genomes** provides some information about only the "best" genomes of each replicate set, as well as a couple quick overall statistics.

2.4.2 Warnings

Warnings look for two things: **de-replicated genome similarity** and **secondary clusters that were almost different**. **All warnings are located in the log directory within the work directory, in a file titled warnings.txt**

de-replicated genome similarity warns when de-replicated genomes are similar to each other. This is to try and catch cases where similar genomes were split into different primary clusters, and thus failed to be de-replicated.

secondary clusters that were almost different alerts the user to cases where genomes are on the edge between being considered "same" or "different". That is, if a genome is close to one of the differentiating lines in the Primary and Secondary Clustering Dendrograms shown above.

2.4.3 Other data

The folder `dereplicated_genomes` holds a copy of the "best" genome of each secondary cluster

See also:

Almost all data that dRep generates at any point is able to be accessed by the user. This includes the full checkM results of each genome, the value of all genome comparisons, the raw hierarchical clustering files, the primary and secondary cluster identity of each genome, etc.

For information on where all of this is stored, see *Advanced Use*

2.5 Important Concepts

The parameters selected during de-replication and genome comparison are critical to understanding what dRep is actually doing to your genome set. This section of the docs will address the following key concepts:

1. Choosing an appropriate secondary ANI threshold. This threshold dictates how similar genomes need to be for them to be considered the "same". This section goes into how to choose an appropriate *S_ANI* threshold.

2. Minimum alignment coverage. When calculating the ANI between genomes we also determine what fraction of the genomes were compared. The value can be used to establish minimum needed level of overlap between genomes (*cov_thresh*), but there is a lot to be aware of when using this parameter.

3. Choosing representative genomes. During de-replication the first step is identifying groups of similar genomes, and the second step is picking a Representative Genome (RG) for each cluster. This section gets into the second step.

4. Using greedy algorithms. Greedy algorithms are those that take "shortcuts" to arrive at a solution. dRep can use greedy algorithms at several stages to speed up runtime; this section gets into how to use them and what to be aware of.

5. Importance of genome completeness. Due to it's reliance on Mash genomes must be somewhat complete for dRep to handle them properly. This section explains why.

6. Oddities of hierarchical clustering. After all genomes are compared, dRep uses hierarchical clustering to convert pair-wise ANI values into clusters of genomes. This section gets into how this works and where it can go wrong.

7. Overview of genome comparison algorithms. dRep has lots of different genome comparison algorithms to choose from. This section briefly describes how they work and the differences between them.

8. Comparing and dereplicating non-bacterial genomes. Some thoughts about the appropriate parameters to use when comparing things like bacteriophages, plasmids, and eukaryotes.

See also:

User manual For more general descriptions of routine parameters

Overview For a general overview of dRep concepts

2.5.1 1. Choosing an appropriate secondary ANI threshold

There is no one definition of the average nucleotide identity (ANI) shared between two genomes that are the "same". This is a decision that the user must make on their own depending on their own specific application. The ANI is determined by the **secondary clustering algorithm**, the **minimum secondary ANI** is the minimum ANI between genomes to be considered the "same", and the **minimum aligned fraction** is the minimum amount of genome overlap to believe the reported ANI value. See 7. Overview of genome comparison algorithms for descriptions of **secondary clustering algorithms** and 2. Minimum alignment coverage for concepts about adjusting the **minimum aligned fraction**.

One use-case for dereplication is to take a large group of genomes and pick out set of Species-level Representative Genomes (SRGs). This was recently done using dRep in Almeida et. al. , where the goal was to generate a comprehensive set high-quality reference genomes for microbes that live in the human gut. These is debate about the exact threshold for delineating bacteria of different species, but **most studies agree that 95% ANI is an appropriate threshold for species-level de-replication**. See Olm et. al. 2020 for context on how this threshold was chosen and some debate on whether bacterial species exist as a continuum or as discrete units in nature.

Another use-case for dereplicaiton is to generate a set of genomes that are distinct enough to avoid mis-mapping. If one were to map metagenomic reads (which are usually ~150bp) to two genomes that were 99.9% the same, most reads would map equally well to both genomes, and you wouldn't be able to tell which read truly "belongs" to one genome or the other. **If the goal of dereplication is to generate a set of genomes that are distinct when mapping short reads, 98% ANI is an appropriate threshold.** See the following page for how the 98% value was arrived at.

The default value in dRep is 99%, and this is a limit to how precise the secondary ANI threshold can be. As a rule of thumb thresholds as high as 99.9% are probably safe, but higher than that is beyond the limit of detection. Comparing genomes to themselves will usually yield ANI values of ~99.99%, since the algorithms aren't perfect and get thrown off by genomic repeat regions and things like this. See the publication for the program InStrain for details about testing the limits of detection for dRep, and how to perform more detailed stain-level comparisons if desired.

For additional thoughts on this threshold, see the blog post: Are these microbes the "same"?.

Note: Keep in mind that in all cases you are collapsing closely related, but probably not identical, strains / genomes. This is simply the nature of the beast. If desired, you can compare the strains by mapping the original reads back to

the de-replicated genome to visualize the strain cloud (Bendall 2016, blog post) using the program inStrain

2.5.2 2. Minimum alignment coverage

The **minimum aligned fraction** is the minimum amount that genomes must overlap to make the reported ANI value "count". This value is reported as part of the ANIm/gANI algorithms.

Imagine a scenario where two distantly related genomes share a single identical transposon. When the genome comparison algorithm is run, the transposon is probably the only part of the genomes that aligns, and the alignment will have 100% ANI. This will result in a reported ANI of 100%, and reported **aligned fraction** of ~0.1%. The **minimum aligned fraction** is to handle the above scenario- anything with less than the minimum aligned fraction of genome alignment will have the ANI changed to 0. Default value is 10%.

The way that dRep handles **minimum aligned fraction** can account for the above scenario, but it's pretty clumsy overall. When a comparison is below the **minimum aligned fraction** dRep literally just changes the ANI from whatever the algorithm reported to a "0". This can cause problems with hierarchical clustering down the line (see 7. Overview of genome comparison algorithms). I've tried to think about better ways of handling it, but it's just a difficult thing to account for. How do you handle a set of genomes that share 100% ANI over 20% of their genomes? **After years of using dRep, however, I've come to this conclusion that this is rarely a problem in practical reality.** In most cases the aligned fraction is either really high (>50%) or really low (>10%), so the default value of 10% works in most cases. See Figure 1 in Olm et. al. 2020 for a depiction of typical intra-species (genomes share >95% ANI) alignment coverage values, and notice that genomes with >95% ANI almost never have low alignment coverage.

Note: It has been suggested that a minimum aligned fraction of 60% should be applied to species-level taxonomic definitions (Varghese 2015). However, this is probably too stringent when incomplete genomes are being used (as is often the case with genome de-replication).

2.5.3 3. Choosing representative genomes

dRep uses a score-based system to pick representative genomes. Each genome in the cluster is assigned a score, and the genome with the highest score is chosen as the representative. This score is based on the formula:

$$A * Completeness - B * Contamination + C * (Contamination * (strainheterogeneity/100)) + D * log(N50) + E * log(siz$$

Where A-F are command-line arguments with default values of 1, 5, 1, 0.5, 0, and 1, respectively. Adjusting A-F lets you decide how much to weight particular features when choosing representative genomes. For example, if you really care about having low contamination and high N50, you could increase B and D.

Completeness, Contamination, and strain heterogeneity are provided by the user or calculated with checkM. N50 is a measure of how big the pieces are that make up the genome. size is the total length of the genome. Centrality is a measure of how similar a genome is to all other genomes in it's cluster. This metric helps pick genome that are similar to all other genomes, and avoid picking genomes that are relative outliers.

Some publications have added other metrics to their scoring when picking representative genomes, such as whether or not the genome came from an isolate. See A unified catalog of 204,938 reference genomes from the human gut microbiome and A complete domain-to-species taxonomy for Bacteria and Archaea for examples.

2.5.4 4. Using greedy algorithms

In layman's terms, greedy algorithms are those that take shortcuts to run faster and arrive at solutions that may not be optimal but are "close enough". The better the greedy algorithm, the smaller the difference between the optimal

solution and the greedy solution. Since pair-wise comparisons quickly scale to a level that would take decades to compute, dRep uses a number of greedy algorithms to speed things up.

One greedy algorithm dRep uses is primary clustering. Performing this step dramatically reduces the number of genome comparisons have to be made, decreasing run-time. The cost of this is that if genomes end up in different primary clusters they will never be compared, and thus will never be in the same final clusters. That's why the section below (Importance of genome completeness) is important.

Note: In 2021 (dRep v3) several additional greedy algorithms were introduced, described below. These are relatively new features, so please don't hesitate to reach out if you notice problems or have suggestions.

–multiround_primary_clustering performs primary clustering in a series of groups that are than merged at the end with single linkage clustering. This dramatically decreases RAM usage and increases speed, and the cost of a minor loss in precision and the inability to plot primary_clustering_dendrograms. Especially helpful when clustering 5000+ genomes.

–greedy_secondary_clustering use a heuristic to avoid pair-wise comparisons when doing secondary clustering. The way this works is that one genome is randomly chosen to represent a cluster. Then the next genome is compared to that one. If it's below ANI thresholds with that genome, it will be put in that cluster. If it's not, it will be put into a new cluster and made the representative genome of the new cluster. The 3rd genome will then be comparing to all cluster representatives, and so on. This essentially results in single linkage clustering without the need for pair-wise comparisons. Unfortunately this doesn't increase speed as much as you would expect due to the need of FastANI to continually re-sketch genomes. This option only works for the fastANI *S_algorithm* at the moment.

–run_tertiary_clustering is not a greedy algorithm, but is a way to handle potential inconsistencies introduced by greedy algorithms. Once clustering is complete and representative genomes are chosen, this option run an additional round of clustering on the final genome set. This is especially useful when greedy clustering is performed and/or to handle cases where similar genomes end up in different primary clusters. It's essentially a check to make sure that all genomes are as distinct from one another as they should be based on the parameters given.

2.5.5 5. Importance of genome completeness

This decision is much more complicated than the previous. Essentially, there exists a trade-off between computational efficiency and the minimum genome completeness.

As shown in the above Figure A, the lower the limit of genome completeness, the lower possible aligned fraction of two genomes. This makes sense- if you randomly take 20% of a genome, and than do the same thing again, when you compare these two random 20% subsets you would not expect very much of them to align. This "aligned fraction" really becomes a problem when you consider it's effect on Mash:

As shown in the above Figure B, the lower the aligned fraction, the lower the reported Mash ANI **for identical genomes**.

Remember- genomes are first divided into primary clusters using Mash, and then each primary cluster is divided into secondary clusters of the "same" genomes. Therefore, genomes which fit the definition of "same" **must** end up in the same primary cluster, or the program will never realize they're the same. As more incomplete genomes have lower Mash values (even if the genomes are truly identical; see **Figure B**), the more incomplete of genomes you allow into your genome list, the more you must decrease the **primary cluster threshold**.

Note: Having a lower **primary cluster threshold** which will result in larger primary clusters, which will result in more required secondary comparisons. This will result in a longer run-time.

Still with me?

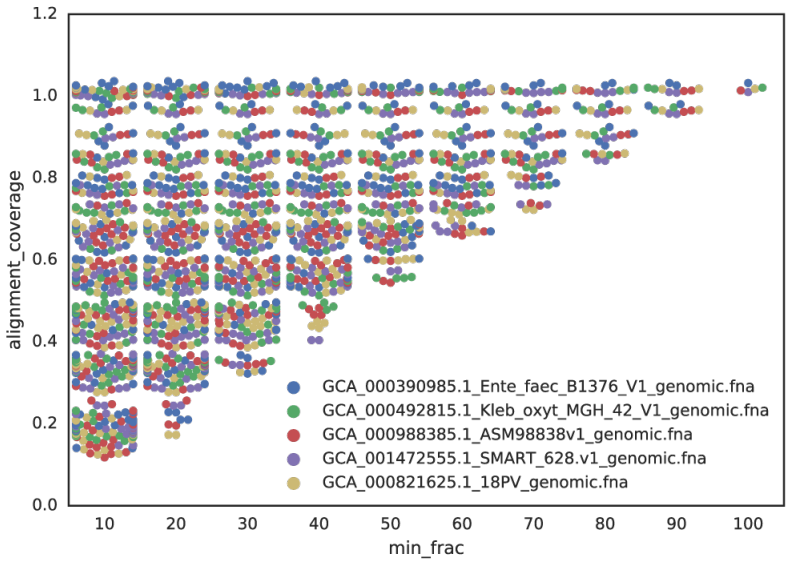


Fig. 1: **Figure A**: Five genomes are subset to fractions ranging from 10% - 100%, and fractions from the same genome are compared. The x-axis is the minimum genome completeness allowed. The looser this value is, the wider the range of aligned fractions.

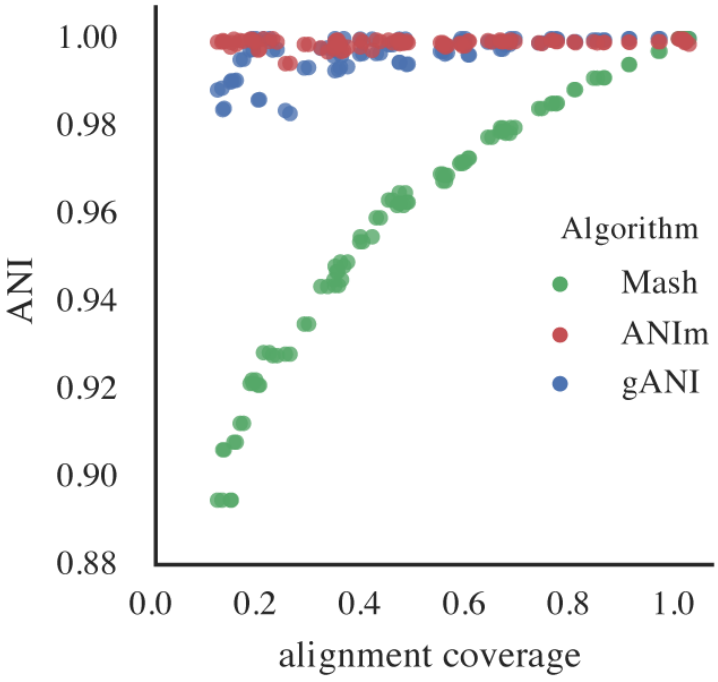
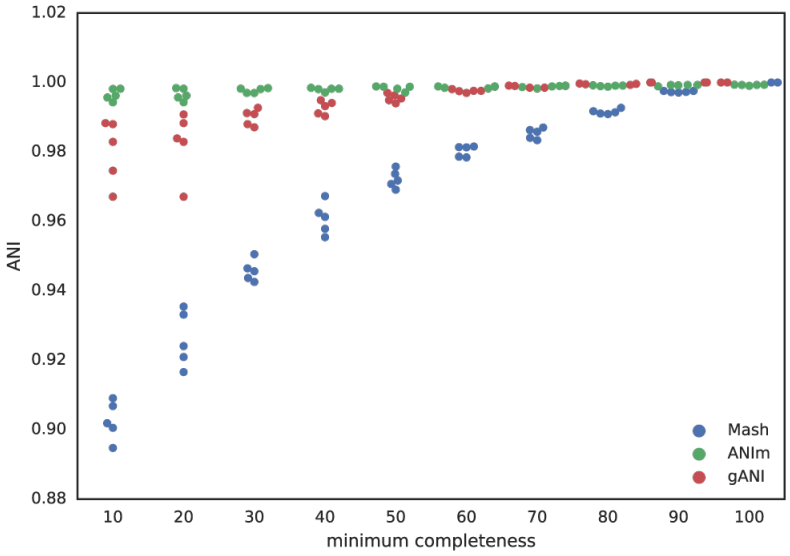


Fig. 2: **Figure B**: An identical *E. coli* genome is subset to fractions ranging from 10% - 100% and fractions are compared. When lower amounts of the genome align (due to incompleteness), Mash ANI is severely impacted

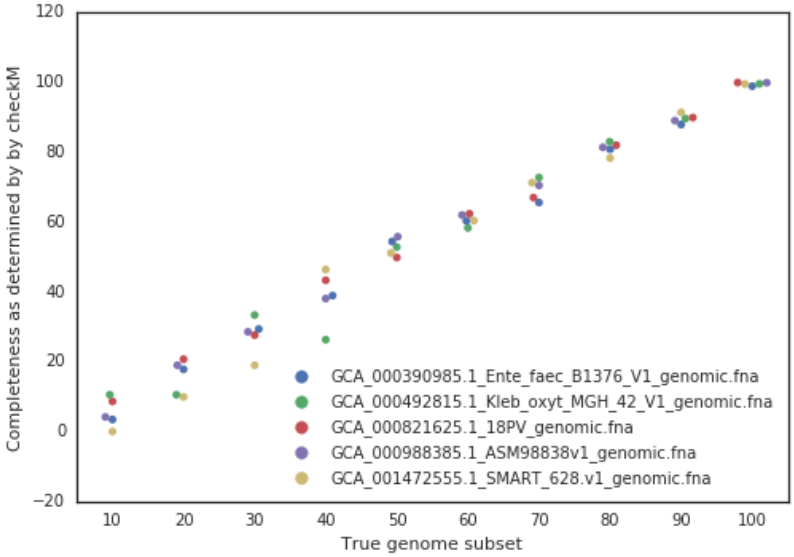
For example, say I set the minimum genome completeness to 50%. If I take an *E. coli* genome, subset it 50% 2 times, and compare those 2 subset genomes together, Mash will report an ANI of 96%. Therefore, the primary cluster threshold must be at least 96%, otherwise the two genomes could end up in different primary clusters, and thus would never have have the secondary algorithm run between them, and thus would not be de-replicated.

You don't want to set the primary cluster threshold super low, however, as this would result in more genomes being included in each primary cluster, and thus more secondary comparisons (which are slow), and thus a higher run-time.

Putting this altogether gives us a figure with the lowest reported ANI of identical genomes being subset to different fractions. This figure only takes into account 5 different genomes, but gives a rough idea of the limits.



A final piece to consider is that when running dRep for real, the user doesn't actually know how incomplete their genomes are. They have to rely on metrics like single copy gene inventories to tell them. This is the reason phage and plasmids are not currently supported by dRep- there is no way of knowing how complete they are, and thus no way of filtering out the bins that are too incomplete. In general though, checkM is pretty good at accessing genome completeness:



Note: Some general guidelines for picking genome completeness thresholds:

- Going below 50% completeness is not recommended. The resulting genomes will be very crappy anyways, and even the secondary algorithms break-down at this point.
- Lowering the secondary ANI should result in a consummate lowering in MASH ANI. This is because you want Mash to group non-similar *and* incomplete genomes.

2.5.6 6. Oddities of hierarchical clustering

The most common "bug" reported to dRep is that genome pairs with an ANI greater than the threshold end up in different clusters, or genome pairs with ANI less than the threshold end up in the same cluster. This almost always happens because of hierarchical clustering, the way that dRep transforms pair-wise ANI values into clusters.

By default, hierarchical clustering is performed in dRep using average linkage (https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html) which can result in scenarios like those described above. If you want all cases where two genomes are over your thresholds to be in the same cluster, you can run it in single mode (*–clusterAlg* single). The problem is that this can create big clusters- for example if A is similar to B, and B is similar to C, but A and C are not similar, what do you do? In single mode A, B, and C will be in the same cluster, and in average mode some averaging is done to try and handle this.

dRep can use any method of linkage listed at the following webpage by using the –clusterAlg argument: https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html .

Note: dRep also generates dendrogram figures to visualize the hierarchical clustering (though if you have too many genomes this can get too big to be visualized). This blog post was instrumental in my understanding of how hierarchical clustering works and how to implement it in python.

2.5.7 7. Overview of genome comparison algorithms

Primary clustering is always performed with Mash; an extremely fast but somewhat inaccurate algorithm.

There are several supported **secondary clustering algorithms**. These calculate the accurate Average Nucleotide Identity (ANI) between genomes that is used to cluster genomes into secondary clusters. The following algorithms are currently supported as of version 3:

- **ANIn** (Richter 2009). This aligns whole genomes with nucmer and compares the aligned regions.
- **ANImf** (DEFAULT). This is the same as ANIn, but filters the alignments such that each region of genome 1 and only align to a single region of genome 2. This takes slightly more time, but is much more accurate on genomes with repeat regions
- **gANI** (Varghese 2015). This aligns genes (ORFs) called by Prodigal instead of aligning whole genomes. This algorithm is a bit faster than ANIm-based algorithms, but only aligns coding regions.
- **goANI**. This is my own open-source implementation of gANI, which is not open source (and for which the authors would not share the source code when asked). I wrote this algorithm so that I could calculate dN/dS between aligned genes for this study (you can too using dnds_from_drep.py). Requires the program NSimScan.
- **FastANI** (Jain 2018). A really fast Mash-based algorithm that can also handle incomplete genomes. Seems to be just as accurate as alignment-based algorithms. **Should probably be the default algorithm when you care about runtime.***

Note: None of these algorithms are perfect, especially in repeat-prone genomes. Regions of the genome which are not homologous can align to each other and artificially decrease ANI. In fact, when a genome is compared to itself, the algorithms often reports values <100% for this reason.

2.5.8 8. Comparing and dereplicating non-bacterial genomes

dRep was developed for the use-case of bacterial dereplication, and there are some things to be aware of when running it on non-bacterial entities.

A major thing to be aware of is primary clustering. As described in 5. Importance of genome completeness, genomes need to be >50% complete for primary clustering to work. If you're comparing entities in which you cannot assess completeness or in which you want to compare genomes that share only a limited number of genes (e.g. phage or plasmids), this a problem. The easiest way to handle it is to avoid primary cluster altogether with the parameter *–SkipMash*, or lower the primary clustering threshold with *-pa*.

Also consider the effect of alignment coverage (2. Minimum alignment coverage) on hierarchical clustering (6. Oddities of hierarchical clustering). If your working with entities that are especially mosaic, like phage, this can be a bigger problem than with bacteria.

Genome filtering and scoring is also a major factor. If your genomes can't be assessed by checkM, you can turn off quality filtering and the use of completeness and contamination when picking genomes with the flag *–ignoreGenomeQuality*.

When considering these options for my own studies (Olm 2019 and Olm 2020), I landed on the following dRep command for clustering bacteriophages and plasmids. Please take this for what it is, one person's attempt to handle these parameters for their specific use-case, and don't be afraid to make additional adjustments as you see fit:

```
dRep dereplicate --S_algorithm ANImf -nc .5 -l 10000 -N50W 0 -sizeW 1 --
↪ignoreGenomeQuality --clusterAlg single
```

2.6 User manual

dRep has 3 commands: compare, dereplicate, and check dependencies. To see a list of these options check the help:

```
$ dRep -h

              ..::: dRep v3.0.0 :::...

Matt Olm. MIT License. Banfield Lab, UC Berkeley. 2017 (last updated 2020)

See https://drep.readthedocs.io/en/latest/index.html for documentation
Choose one of the operations below for more detailed help.

Example: dRep dereplicate -h

Commands:
  compare            -> Compare and cluster a set of genomes
  dereplicate        -> De-replicate a set of genomes
  check_dependencies -> Check which dependencies are properly installed
```

In previous versions of dRep (everything before v3) the user could run a number of additional modules separately, but now they can only be run as part of the larger workflows *compare* and *dereplicate*. Many of the modules are the same

for *compare* and *dereplicate*, however, and in cases where these is the same parameter in both it functions exactly the same in each.

dRep has descriptions in the program help for all the adjustable parameters. If any of these are particularly confusing, don't hesitate to send an email to ask what it does.

See also:

Important Concepts for theoretical thoughts about how to choose appropriate parameters and thresholds

Example Output for help interpreting the output from your run in the work directory

Advanced Use for access to the raw output data and the python API

2.6.1 Compare

This workflow compares a set of genomes. For a list of all parameters, check the help:

```
$ dRep compare -h
  usage: dRep compare [-p PROCESSORS] [-d] [-h] [-g [GENOMES [GENOMES ...]]]
                      [--S_algorithm {fastANI,gANI,goANI,ANIn,ANImf}]
                      [-ms MASH_SKETCH] [--SkipMash] [--SkipSecondary]
                      [--n_PRESET {normal,tight}] [-pa P_ANI] [-sa S_ANI]
                      [-nc COV_THRESH] [-cm {total,larger}]
                      [--clusterAlg {median,weighted,single,complete,average,ward,
↪centroid}]
                      [--multiround_primary_clustering]
                      [--primary_chunksize PRIMARY_CHUNKSIZE]
                      [--greedy_secondary_clustering]
                      [--run_tertiary_clustering] [--warn_dist WARN_DIST]
                      [--warn_sim WARN_SIM] [--warn_aln WARN_ALN]
                      work_directory

  positional arguments:
    work_directory        Directory where data and output are stored
                          *** USE THE SAME WORK DIRECTORY FOR ALL DREP OPERATIONS ***

  SYSTEM PARAMETERS:
    -p PROCESSORS, --processors PROCESSORS
                          threads (default: 6)
    -d, --debug           make extra debugging output (default: False)
    -h, --help            show this help message and exit

  GENOME INPUT:
    -g [GENOMES [GENOMES ...]], --genomes [GENOMES [GENOMES ...]]
                          genomes to filter in .fasta format. Not necessary if
                          Bdb or Wdb already exist. Can also input a text file
                          with paths to genomes, which results in fewer OS
                          issues than wildcard expansion (default: None)

  GENOME COMPARISON OPTIONS:
    --S_algorithm {fastANI,gANI,goANI,ANIn,ANImf}
                          Algorithm for secondary clustering comaprisons:
                          fastANI = Kmer-based approach; very fast
                          ANImf   = (DEFAULT) Align whole genomes with nucmer; filter␣
↪alignment; compare aligned regions
                          ANIn    = Align whole genomes with nucmer; compare aligned␣
↪regions
```

(continues on next page)

(continued from previous page)

```
                        gANI    = Identify and align ORFs; compare aligned ORFS
                        goANI   = Open source version of gANI; requires nsmimscan
                         (default: ANImf)
  -ms MASH_SKETCH, --MASH_sketch MASH_SKETCH
                        MASH sketch size (default: 1000)
  --SkipMash            Skip MASH clustering, just do secondary clustering on
                        all genomes (default: False)
  --SkipSecondary       Skip secondary clustering, just perform MASH
                        clustering (default: False)
  --n_PRESET {normal,tight}
                        Presets to pass to nucmer
                        tight   = only align highly conserved regions
                        normal  = default ANIn parameters (default: normal)

 GENOME CLUSTERING OPTIONS:
  -pa P_ANI, --P_ani P_ANI
                        ANI threshold to form primary (MASH) clusters
                        (default: 0.9)
  -sa S_ANI, --S_ani S_ANI
                        ANI threshold to form secondary clusters (default:
                        0.99)
  -nc COV_THRESH, --cov_thresh COV_THRESH
                        Minmum level of overlap between genomes when doing
                        secondary comparisons (default: 0.1)
  -cm {total,larger}, --coverage_method {total,larger}
                        Method to calculate coverage of an alignment
                        (for ANIn/ANImf only; gANI and fastANI can only do larger␣
↪method)
                        total   = 2*(aligned length) / (sum of total genome lengths)
                        larger  = max((aligned length / genome 1), (aligned_length /
↪ genome2))
                         (default: larger)
  --clusterAlg {median,weighted,single,complete,average,ward,centroid}
                        Algorithm used to cluster genomes (passed to
                        scipy.cluster.hierarchy.linkage (default: average)

 GREEDY CLUSTERING OPTIONS
 These decrease RAM use and runtime at the expense of a minor loss in accuracy.
 Recommended when clustering 5000+ genomes:
  --multiround_primary_clustering
                        Cluster each primary clunk separately and merge at the
                        end with single linkage. Decreases RAM usage and
                        increases speed, and the cost of a minor loss in
                        precision and the inability to plot
                        primary_clustering_dendrograms. Especially helpful
                        when clustering 5000+ genomes. Will be done with
                        single linkage clustering (default: False)
  --primary_chunksize PRIMARY_CHUNKSIZE
                        Impacts multiround_primary_clustering. If you have
                        more than this many genomes, process them in chunks of
                        this size. (default: 5000)
  --greedy_secondary_clustering
                        Use a heuristic to avoid pair-wise comparisons when
                        doing secondary clustering. Will be done with single
                        linkage clustering. Only works for fastANI S_algorithm
                        option at the moment (default: False)
  --run_tertiary_clustering
```

(continues on next page)

(continued from previous page)

```
                        Run an additional round of clustering on the final
                        genome set. This is especially useful when greedy
                        clustering is performed and/or to handle cases where
                        similar genomes end up in different primary clusters.
                        Only works with dereplicate, not compare. (default:
                        False)

  WARNINGS:
    --warn_dist WARN_DIST
                        How far from the threshold to throw cluster warnings
                        (default: 0.25)
    --warn_sim WARN_SIM   Similarity threshold for warnings between dereplicated
                        genomes (default: 0.98)
    --warn_aln WARN_ALN   Minimum aligned fraction for warnings between
                        dereplicated genomes (ANIn) (default: 0.25)

  Example: dRep compare output_dir/ -g /path/to/genomes/*.fasta
```

2.6.2 Dereplicate

This workflow dereplicates a set of genomes. For a list of all parameters, check the help:

```
$ dRep dereplicate -h
usage: dRep dereplicate [-p PROCESSORS] [-d] [-h] [-g [GENOMES [GENOMES ...]]]
                        [-l LENGTH] [-comp COMPLETENESS] [-con CONTAMINATION]
                        [--ignoreGenomeQuality] [--genomeInfo GENOMEINFO]
                        [--checkM_method {taxonomy_wf,lineage_wf}]
                        [--set_recursion SET_RECURSION]
                        [--S_algorithm {goANI,ANIn,gANI,ANImf,fastANI}]
                        [-ms MASH_SKETCH] [--SkipMash] [--SkipSecondary]
                        [--n_PRESET {normal,tight}] [-pa P_ANI] [-sa S_ANI]
                        [-nc COV_THRESH] [-cm {total,larger}]
                        [--clusterAlg {single,ward,complete,weighted,centroid,median,
↪average}]
                        [--multiround_primary_clustering]
                        [--primary_chunksize PRIMARY_CHUNKSIZE]
                        [--greedy_secondary_clustering]
                        [--run_tertiary_clustering]
                        [-comW COMPLETENESS_WEIGHT]
                        [-conW CONTAMINATION_WEIGHT]
                        [-strW STRAIN_HETEROGENEITY_WEIGHT] [-N50W N50_WEIGHT]
                        [-sizeW SIZE_WEIGHT] [-centW CENTRALITY_WEIGHT]
                        [--warn_dist WARN_DIST] [--warn_sim WARN_SIM]
                        [--warn_aln WARN_ALN]
                        work_directory

positional arguments:
  work_directory        Directory where data and output are stored
                        *** USE THE SAME WORK DIRECTORY FOR ALL DREP OPERATIONS ***

SYSTEM PARAMETERS:
  -p PROCESSORS, --processors PROCESSORS
                        threads (default: 6)
  -d, --debug           make extra debugging output (default: False)
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
GENOME INPUT:
  -g [GENOMES [GENOMES ...]], --genomes [GENOMES [GENOMES ...]]
                        genomes to filter in .fasta format. Not necessary if
                        Bdb or Wdb already exist. Can also input a text file
                        with paths to genomes, which results in fewer OS
                        issues than wildcard expansion (default: None)

GENOME FILTERING OPTIONS:
  -l LENGTH, --length LENGTH
                        Minimum genome length (default: 50000)
  -comp COMPLETENESS, --completeness COMPLETENESS
                        Minumum genome completeness (default: 75)
  -con CONTAMINATION, --contamination CONTAMINATION
                        Maximum genome contamination (default: 25)

GENOME QUALITY ASSESSMENT OPTIONS:
  --ignoreGenomeQuality
                        Don't run checkM or do any quality filtering. NOT
                        RECOMMENDED! This is useful for use with
                        bacteriophages or eukaryotes or things where checkM
                        scoring does not work. Will only choose genomes based
                        on length and N50 (default: False)
  --genomeInfo GENOMEINFO
                        location of .csv file containing quality information
                        on the genomes. Must contain: ["genome"(basename of
                        .fasta file of that genome), "completeness"(0-100
                        value for completeness of the genome),
                        "contamination"(0-100 value of the contamination of
                        the genome)] (default: None)
  --checkM_method {taxonomy_wf,lineage_wf}
                        Either lineage_wf (more accurate) or taxonomy_wf
                        (faster) (default: lineage_wf)
  --set_recursion SET_RECURSION
                        Increases the python recursion limit. NOT RECOMMENDED
                        unless checkM is crashing due to recursion issues.
                        Recommended to set to 2000 if needed, but setting this
                        could crash python (default: 0)

GENOME COMPARISON OPTIONS:
  --S_algorithm {goANI,ANIn,gANI,ANImf,fastANI}
                        Algorithm for secondary clustering comaprisons:
                        fastANI = Kmer-based approach; very fast
                        ANImf   = (DEFAULT) Align whole genomes with nucmer; filter␣
↪alignment; compare aligned regions
                        ANIn    = Align whole genomes with nucmer; compare aligned␣
↪regions
                        gANI    = Identify and align ORFs; compare aligned ORFS
                        goANI   = Open source version of gANI; requires nsmimscan
                         (default: ANImf)
  -ms MASH_SKETCH, --MASH_sketch MASH_SKETCH
                        MASH sketch size (default: 1000)
  --SkipMash            Skip MASH clustering, just do secondary clustering on
                        all genomes (default: False)
  --SkipSecondary       Skip secondary clustering, just perform MASH
                        clustering (default: False)
  --n_PRESET {normal,tight}
```

(continues on next page)

(continued from previous page)

```
                        Presets to pass to nucmer
                        tight   = only align highly conserved regions
                        normal  = default ANIn parameters (default: normal)

GENOME CLUSTERING OPTIONS:
  -pa P_ANI, --P_ani P_ANI
                        ANI threshold to form primary (MASH) clusters
                        (default: 0.9)
  -sa S_ANI, --S_ani S_ANI
                        ANI threshold to form secondary clusters (default:
                        0.99)
  -nc COV_THRESH, --cov_thresh COV_THRESH
                        Minmum level of overlap between genomes when doing
                        secondary comparisons (default: 0.1)
  -cm {total,larger}, --coverage_method {total,larger}
                        Method to calculate coverage of an alignment
                        (for ANIn/ANImf only; gANI and fastANI can only do larger␣
↪method)
                        total   = 2*(aligned length) / (sum of total genome lengths)
                        larger  = max((aligned length / genome 1), (aligned_length /␣
↪genome2))
                         (default: larger)
  --clusterAlg {single,ward,complete,weighted,centroid,median,average}
                        Algorithm used to cluster genomes (passed to
                        scipy.cluster.hierarchy.linkage (default: average)

GREEDY CLUSTERING OPTIONS
These decrease RAM use and runtime at the expense of a minor loss in accuracy.
Recommended when clustering 5000+ genomes:
  --multiround_primary_clustering
                        Cluster each primary clunk separately and merge at the
                        end with single linkage. Decreases RAM usage and
                        increases speed, and the cost of a minor loss in
                        precision and the inability to plot
                        primary_clustering_dendrograms. Especially helpful
                        when clustering 5000+ genomes. Will be done with
                        single linkage clustering (default: False)
  --primary_chunksize PRIMARY_CHUNKSIZE
                        Impacts multiround_primary_clustering. If you have
                        more than this many genomes, process them in chunks of
                        this size. (default: 5000)
  --greedy_secondary_clustering
                        Use a heuristic to avoid pair-wise comparisons when
                        doing secondary clustering. Will be done with single
                        linkage clustering. Only works for fastANI S_algorithm
                        option at the moment (default: False)
  --run_tertiary_clustering
                        Run an additional round of clustering on the final
                        genome set. This is especially useful when greedy
                        clustering is performed and/or to handle cases where
                        similar genomes end up in different primary clusters.
                        Only works with dereplicate, not compare. (default:
                        False)

SCORING CRITERIA
Based off of the formula:
A*Completeness - B*Contamination + C*(Contamination * (strain_heterogeneity/100)) +␣
↪D*log(N50) + E*log(size) + F*(centrality - S_ani)
```

(continues on next page)

(continued from previous page)

```
A = completeness_weight; B = contamination_weight; C = strain_heterogeneity_weight; D␣
↪= N50_weight; E = size_weight; F = cent_weight:
  -comW COMPLETENESS_WEIGHT, --completeness_weight COMPLETENESS_WEIGHT
                        completeness weight (default: 1)
  -conW CONTAMINATION_WEIGHT, --contamination_weight CONTAMINATION_WEIGHT
                        contamination weight (default: 5)
  -strW STRAIN_HETEROGENEITY_WEIGHT, --strain_heterogeneity_weight STRAIN_
↪HETEROGENEITY_WEIGHT
                        strain heterogeneity weight (default: 1)
  -N50W N50_WEIGHT, --N50_weight N50_WEIGHT
                        weight of log(genome N50) (default: 0.5)
  -sizeW SIZE_WEIGHT, --size_weight SIZE_WEIGHT
                        weight of log(genome size) (default: 0)
  -centW CENTRALITY_WEIGHT, --centrality_weight CENTRALITY_WEIGHT
                        Weight of (centrality - S_ani) (default: 1)

WARNINGS:
  --warn_dist WARN_DIST
                        How far from the threshold to throw cluster warnings
                        (default: 0.25)
  --warn_sim WARN_SIM   Similarity threshold for warnings between dereplicated
                        genomes (default: 0.98)
  --warn_aln WARN_ALN   Minimum aligned fraction for warnings between
                        dereplicated genomes (ANIn) (default: 0.25)

Example: dRep dereplicate output_dir/ -g /path/to/genomes/*.fasta
```

2.6.3 Work Directory

The work directory is where all of the program's internal workings, log files, cached data, and output is stored.

See also:

Example Output for help finding where the output from your run is located in the work directory

Advanced Use for access to the raw internal data (which can be very useful)

2.6.4 Genome filtering

In the *dereplicate* module, the genome set is quality filtered first (for why this is necessary, see *Important Concepts*). This is done using checkM. All genomes which don't pass the length threshold are filtered first to avoid running checkM unnecessarily. All genomes which don't pass checkM thresholds are filtered before comparisons are run to avoid running comparisons unnecessarily.

> **Warning:** All genomes must have at least one ORF called or else checkM will stall, so a length minimum of at least 10,000bp is recommended.

2.6.5 Warnings

A series of checks are preformed to alert the user to potential problems with de-replication. There are two things that it looks for:

de-replicated genome similarity- this is comparing all of the de-replicated genomes to each other and making sure they're not too similar. This is to try and catch cases where similar genomes were split into different primary clusters, and thus failed to be de-replicated. *Depending on the number of de-replicated genomes, this can take a while*

secondary clusters that were almost different- this alerts you to cases where genomes are on the edge between being considered "same" or "different", depending on the clustering parameters you used. *This module reads the parameters you used during clustering from the work directory, so you don't need to specify them again.*

Overall these warnings are a bit half-baked, however, and I personally don't pay attention to them when running dRep myself.

2.7 Advanced Use

2.7.1 Accessing Internal Information

All of internal information is stored in the work directory

work directory file-tree

```
workDirectory
./data
...../checkM/
...../Clustering_files/
...../gANI_files/
...../MASH_files/
...../ANIn_files/
...../prodigal/
./data_tables
...../Bdb.csv  # Sequence locations and filenames
...../Cdb.csv  # Genomes and cluster designations
...../Chdb.csv # CheckM results for Bdb
...../Mdb.csv  # Raw results of MASH comparisons
...../Ndb.csv  # Raw results of ANIn comparisons
...../Sdb.csv  # Scoring information
...../Wdb.csv  # Winning genomes
...../Widb.csv # Winning genomes' checkM information
./dereplicated_genomes
./figures
./log
...../cluster_arguments.json
...../logger.log
...../warnings.txt
```

Data Tables

Within the `data_tables` folder is where organized data lives. It's very easy to access this information, as it's all stored in .csv files.

Note: If you code in Python, I cannot recommend pandas enough for data-frame reading and manipulation. It's how all data is manipulated behind the scenes in dRep. See the API section below for easy access to these dataframes

Bdb Genome input locations, filenames, and lengths

Cdb Primary cluster, Secondary cluster, and information on clustering method for each genome

Chdb CheckM results for all genomes

Mdb Pair-wise Mash comparison results

Ndb Secondary comparison results

Tdb Taxonomy (as determined by centrifuge)

Sdb The score of each genome

Wdb The cluster and score of de-replicated genomes

Widb Useful checkM information on de-replicated genomes

Clustering files

These pickle files store information on both primary and secondary clusters. Loading the first value gives you the linkage, loading the second value gives you the db that was used to make the linkage, loading the third value give you a dictionary of the arguments that were used to make the linkage.

For example:

```
f = open(pickle, 'rb')
linkage = pickle.load(f)
db = pickle.load(f)
arguments = pickle.load(f)
```

Raw data

Refer to the above file structure to find the rest of the raw data. The data is kept from all program runs, so you can find the raw ANIm/gANI files, Mash sketches, prodigal gene predictions, centrifuge raw output, ect.

2.7.2 Using external genome quality information

If you already have your own genome quality information and would not like dRep to run checkM to generate it again, you can provide it using the *genomeInformation* flag.

The genomeInformation file must be in .csv format and have the columns "genome", "completeness", and "contamination". Columns "completeness" and "contamination" should be 0-100, and "genome" is the filename of the genome.

For example:

```
genome,completeness,contamination
Enterococcus_casseliflavus_EC20.fasta,98.28,0.0
Enterococcus_faecalis_T2.fna,98.28,0.0
Enterococcus_faecalis_TX0104.fa,96.55,0.0
Enterococcus_faecalis_YI6-1.fna,98.28,0.0
Escherichia_coli_Sakai.fna,100.0,0.0
```

2.7.3 Caching

The reason that dRep stores all of the raw information that it does is so that if future operations require the same file, it can just load the file that's already there instead of making it again. This goes for prodigal gene predictions, checkM,

centrifuge, all ANI algorithms, ect. The data-frame that uses the information **will** be remade, but the information itself will not.

The reason I mention this is because if you would like to run another dRep operation that's similar to one you've already run, you can use the results of the last run to speed up the second run.

For example, say you've already run the dereplicate_wf using gANI and want to run the same thing but with ANIm to compare. You can make a copy of the gANI work directory, and then run the dereplicate_wf on the copy specifying the new secondary algorithm. It will have to run all of the ANIm comparisons, but will **not** re-run checkM, prodigal, centrifuge, ect., as the data will already be cached in the work directory.

> **Warning:** Be warned, this is somewhat buggy and can easily get out of hand. While it does save time, sometimes it's just best to re-run the whole thing again with a clean start

Restarting a clustering job with already completed primary clustering

There are (rare) circumstances where you may way to run primary clustering on one machine and secondary clustering on another. Primary cluster needs more RAM, and secondary clustering needs more cores, so doing this could let you choose the optimal machine for each specific step. There's not a formal way of making dRep do this, but there is an hacky way. If you run dRep in debug mode (with `-d`), it'll make a file named "CdbF.csv" in the data_tables folder. This file contains the primary clustering information, and has a very simple format that looks like this:

```
genome,primary_cluster
Enterococcus_casseliflavus_EC20.fasta,0
Enterococcus_faecalis_T2.fna,0
Enterococcus_faecalis_TX0104.fa,0
Enterococcus_faecalis_YI6-1.fna,0
Escherichia_coli_Sakai.fna,1
```

If you run a dRep job on a work directort that already has this file in the `data_tables` folder, as well as a file named `Mdb.csv` (it doesn't actually matter what's in that file), as well as in debug mode (`-d`), dRep will load this file instead of running primary clustering. You can also make the CdbF.csv file yourself using the format above.

I know this is confusing- it's a hacky thing I threw together for my own research, but I figured I would let everyone else know about it as well. Feel free to shoot me an email if this is something you're interested in doing and can't get it to work.

2.7.4 API

See *dRep API* for the API to dRep. For example:

```
from drep.WorkDirectory import WorkDirectory

wd = WorkDirectory('path/to/workdirectory')
Mdb = wd.get_db('Mdb')
Cdb = wd.get_db('Cdb')
...
```

This will work for all datatables.

Be warned that the API is not very well maintained or documented, and you'll likely have to do a bit of digging into the source code if you want to use it extensively.

2.7.5 Troubleshooting checkM

One of the most common problems users have when running dRep are failures related to the program checkM. These errors can show up as either of the following:

```
New checkM db needs to be made

!!! checkM failed !!!
```

These errors can be caused by checkM crashing a variety of ways; here are some tips to figure out what's wrong.

Note: Sometimes the easier thing to do is just run checkM (or whatever genome assessment tool you prefer) yourself and provide the results to dRep, instead of making dRep run checkM. See the above section "Using external genome quality information" for more info.

---

1) Ensure that you have checkM installed. You can do this with the *check_dependencies option*:

```
With dRep version 3+

$ dRep check_dependencies
mash................................. all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/mash)
nucmer............................... all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/nucmer)
checkm............................... all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/checkm)
ANIcalculator........................ !!! ERROR !!!   (location = None)
prodigal............................. all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/prodigal)
centrifuge........................... all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/centrifuge)
nsimscan............................. !!! ERROR !!!   (location = None)
fastANI.............................. all good        (location = /home/
 ↪mattolm/bin/fastANI)`

With dRep version 2+

$ dRep bonus test --check_dependencies
mash................................. all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/mash)
nucmer............................... all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/nucmer)
checkm............................... all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/checkm)
ANIcalculator........................ !!! ERROR !!!   (location = None)
prodigal............................. all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/prodigal)
centrifuge........................... all good        (location = /home/
 ↪mattolm/miniconda3/envs/drep_testing/bin/centrifuge)
nsimscan............................. !!! ERROR !!!   (location = None)
fastANI.............................. all good        (location = /home/
 ↪mattolm/bin/fastANI)`
```

If checkM reports `!! ERROR !!`, try and re-install it.

2) Make sure you have have the checkM data installed and accessible, as described here: https://github.com/Ecogenomics/CheckM/wiki/Installation#how-to-install-checkm

3) To see the specific error that checkM is throwing, re-run dRep with the `-d` flag. This will produce a number of files in the `log/cmd_logs/` folder. Looking through these will be the commands that dRep gave to checkM, and the STDERR and STDOUT that checkM produced. Looking at the actual error code checkM is giving can be really helpful in figuring out what's wrong

4) If you're running lots of genomes, sometimes python will hit a recursion limit while running checkM and stall out. To fix this you can increase the recursion limit by setting the flag `--set_recursion` to some really big number. **Note: as of dRep version 3.2.0, dRep runs checkM in groups to prevent this problem. See ''checkm_group_size'' for more info.**

5) A newer problem is the error `New checkM db needs to be made`. This usually means that checkM worked on some, but not all of your genomes. Check `log/logger.log` to see which ones failed. I haven't quite figured out this problem yet. If you encounter it I would encourage you to just run checkM outside of dRep. If you think you know how to fix it, please send me an email.

6) You can check checkM's run log at ''data/checkM/checkM_outdir/". Sometimes there's helpful info there as to what's going wrong.

7) Check dRep's log file for lines with the words `Running CheckM with command`. This will tell you the exact checkM command dRep tried to run. Trying to run these commands yourself can be another way of seeing what the actual error is and troubleshooting.

8) If all else fails, please post a GitHub issue. I'm happy to help troubleshoot.

2.8 dRep API

This allows you to call the internal methods of dRep using your own python program

2.8.1 drep.d_filter

d_filter - a subset of drep

Filter genomes based on genome length or quality. Also can run prodigal and checkM

`drep.d_filter.`**`calc_fasta_length`**(*fasta_loc*)
Calculate the length of the .fasta file and retun length

Parameters **`fasta_loc`** – location of .fasta file

Returns total length of all .fasta files

Return type int

`drep.d_filter.`**`calc_genome_info`**(*genomes: list*)
Calculate the length and N50 of a list of genome locations

Parameters **`genomes`** – list of locations of genomes

Returns pandas dataframe with ["location", "length", "N50", "genome"]

Return type DataFrame

`drep.d_filter.`**`calc_n50`**(*loc*)
Calculate the N50 of a .fasta file

Parameters **`fasta_loc`** – location of .fasta file.

Returns N50 of .fasta file.

Return type int

drep.d_filter.**chdb_to_genomeInfo**(*chdb*)
Convert the output of checkM (chdb) into genomeInfo

Parameters **chdb** – dataframe of checkM

Returns genomeInfo

Return type DataFrame

drep.d_filter.**d_filter_wrapper**(*wd*, ***kwargs*)
Controller for the dRep filter operation

Parameters

- **wd** (WorkDirectory) – The current workDirectory
- ****kwargs** – Command line arguments

Keyword Arguments

- **genomes** – genomes to filter in .fasta format
- **genomeInfo** – location of .csv file with the columns: ["genome"(basename of .fasta file of that genome), "completeness"(0-100 value for completeness of the genome), "contamination"(0-100 value of the contamination of the genome)]
- **processors** – Threads to use with checkM / prodigal
- **overwrite** – Overwrite existing data in the work folder
- **debug** – If True, make extra output when running external scripts
- **length** – minimum genome length when filtering
- **completeness** – minimum genome completeness when filtering
- **contamination** – maximum genome contamination when filtering
- **ignoreGenomeQuality** – Don't run checkM or do any quality-based filtering (not recommended)
- **checkM_method** – Either lineage_wf (more accurate) or taxonomy_wf (faster)

Returns stores Bdb.csv, Chdb.csv, and GenomeInfo.csv in the work directory

Return type Nothing

drep.d_filter.**filter_bdb**(*bdb*, *Gdb*, ***kwargs*)
Filter bdb based on Gdb

Parameters

- **bdb** – DataFrame with ["genome"]
- **Gdb** – DataFrame with ["genome", "completeness", "contamination"]

Keyword Arguments

- **min_comp** – Minimum genome completeness (%)
- **max_con** – Maximum genome contamination (%)

Returns bdb filtered based on completeness and contamination

Return type DataFrame

drep.d_filter.**run_checkM**(*genome_folder_whole*, *checkm_outf_whole*, ***kwargs*)
Run checkM

WARNING- this will result in wrong genome length and genome N50 estimate, due to it being run on prodigal output

Parameters

- **genome_folder** – location of folder to run checkM on - should be full of files ending in .faa (result of prodigal)
- **checkm_outf** – location of folder to store checkM output

Keyword Arguments

- **processors** – number of threads
- **checkm_method** – either lineage_wf or taxonomy_wf
- **debug** – log all of the commands
- **wd** – if you want to log commands, you also need the wd
- **set_recursion** – if not 0, set the python recursion

drep.d_filter.**run_prodigal**(*genome_list*, *out_dir*, ***kwargs*)
Run prodigal on a set of genomes, store the output in the out_dir

Parameters

- **genome_list** – list of genomes to run prodigal on
- **out_dir** – output directory to store prodigal output

Keyword Arguments

- **processors** – number of processors to multithread with
- **exe_loc** – location of prodigal excutible (will try and find with shutil if not provided)
- **debug** – log all of the commands
- **wd** – if you want to log commands, you also need the wd

drep.d_filter.**sanity_check**(*bdb*, ***kwargs*)
Make sure there are no duplicate names or anything

Parameters **bdb** –

Returns:

drep.d_filter.**validate_chdb**(*Chdb*, *bdb*)
Make sure all genomes in bdb are in Chdb

Parameters

- **Chdb** – dataframe of checkM information
- **bdb** – dataframe with ['genome']

2.8.2 drep.d_cluster

2.8.3 drep.d_choose

d_choose - a subset of drep

Choose best genome from each cluster

drep.d_choose.**add_centrality**(*wd*, *Gdb*, ***kwargs*)
Add a columns named "centrality" to genome info

drep.d_choose.**calc_centrality_from_scratch**(*Bdb*, *Cdb*, *data_folder*)
Calculate centrality from scratch using Mash

drep.d_choose.**choose_winners**(*Cdb*, *Gdb*, ***kwargs*)
Make a scoring database and pick the winner of each cluster

Parameters

- **Cdb** – clustering database
- **Gdb** – genome information database

Keyword Arguments **wrapper** (*See*) –

Returns [Sdb (scoring database), Wdb (winner database)]

Return type List

drep.d_choose.**d_choose_wrapper**(*wd*, ***kwargs*)
Controller for the dRep choose operation

Based off of the formula: A*Completeness - B*Contamination + C*(Contamination * (strain_heterogeneity/100)) + D*log(N50) + E*log(size)

A = completeness_weight; B = contamination_weight; C = strain_heterogeneity_weight; D = N50_weight; E = size_weight

Parameters

- **wd** (WorkDirectory) – The current workDirectory
- ****kwargs** – Command line arguments

Keyword Arguments

- **genomeInfo** – .csv genomeInfo file
- **ignoreGenomeQuality** – Don't run checkM or do any quality-based filtering (not recommended)
- **checkM_method** – Either lineage_wf (more accurate) or taxonomy_wf (faster)
- **completeness_weight** – see formula
- **contamination_weight** – see formula
- **strain_heterogeneity_weight** – see formula
- **N50_weight** – see formula
- **size_weight** – see formula

Returns Makes Sdb (scoreDb) in the workDirectory

drep.d_choose.**load_extra_weight_table**(*loc*, *genomes*, ***kwargs*)

Parameters

- **loc** – location of extra weight table
- **genomes** – list of genomes you have in Cdb
- ****kwargs** – nothing realld

Returns dataframe with columns "genome" and "extra_weight"

drep.d_choose.**pick_winners**(*Sdb*, *Cdb*)
Based on clustering and scores, pick the best genome from every cluster

Parameters

- **Sdb** – score of every genome
- **Cdb** – clustering

Returns Wdb (winner database)

Return type DataFrame

drep.d_choose.**score_genomes**(*genomes*, *Gdb*, *Edb=None*, ***kwargs*)
Calculate the scores for a list of genomes

Parameters

- **genomes** – list of genomes
- **Gdb** – genome information database

Keyword Arguments **wrapper**(*See*) –

Returns Sdb (scoring database)

Return type DataFrame

drep.d_choose.**score_row**(*row*, *extra=0*, ***kwargs*)
Perform the scoring of a row based on kwargs

Parameters **row** – row of genome information

Keyword Arguments

- **ignoreGenomeQuality** – Don't run checkM or do any quality-based filtering (not recommended)
- **completeness_weight** – see formula
- **contamination_weight** – see formula
- **strain_heterogeneity_weight** – see formula
- **N50_weight** – see formula
- **size_weight** – see formula
- **= extra weight to apply**(*extra*) –

Returns score

Return type float

2.8.4 drep.d_analyze

d_analyze - a subset of drep

Make plots based on de-replication

drep.d_analyze.**calc_dist**(*x1*, *y1*, *x2*, *y2*)
Return distance from two points

Args: self explainatory

Returns distance

Return type int

drep.d_analyze.**cluster_test_wrapper**(*wd*, ***kwargs*)
DEPRICATED

drep.d_analyze.**d_analyze_wrapper**(*wd*, ***kwargs*)
Controller for the dRep analyze operation

Parameters

- **wd** – The current workDirectory
- ****kwargs** – Command line arguments

Keyword Arguments **plots** – List of plots to make [list of ints, 1-6]

Returns Makes some plots

drep.d_analyze.**fancy_dendrogram**(*linkage*, *names*, *name2color=False*, *threshold=False*, *self_thresh=False*)
Make a fancy dendrogram

drep.d_analyze.**gen_color_dictionary**(*names*, *name2cluster*)
Make the dictionary name2color

Parameters

- **names** – key in the returned dictionary
- **name2cluster** – a dictionary of name to it's cluster

Returns name -> color

Return type dict

drep.d_analyze.**gen_color_list**(*names*, *name2cluster*)
Make a list of colors the same length as names, based on their cluster

drep.d_analyze.**get_highest_self**(*db*, *genomes*, *min=0.0001*)
Return the highest ANI value resulting from comparing a genome to itself

drep.d_analyze.**mash_dendrogram_from_wd**(*wd*, *plot_dir=False*)
From the wd and kwargs, call plot_MASH_dendrogram

Parameters

- **wd** – WorkDirectory
- **plot_dir** (*optional*) – Location to store figure

Returns Shows plot, makes a plot in the plot_dir

drep.d_analyze.**normalize**(*df*)
Normalize all columns in df to 0-1 except 'genome' or 'location'

Parameters **df** – DataFrame

Returns Nomralized

Return type DataFrame

drep.d_analyze.**plot_ANIn_vs_ANIn_cov**(*Ndb*)
Makes plot and retuns plt.cgf()

All parameters are obvious

drep.d_analyze.**plot_ANIn_vs_len**(*Mdb*, *Ndb*, *exclude_zero_MASH=True*)
Makes plot and retuns plt.cgf()

All parameters are obvious

drep.d_analyze.**plot_MASH_dendrogram**(*Mdb*, *Cdb*, *linkage*, *threshold=False*, *plot_dir=False*)
Make a dendrogram of the primary clustering

Parameters

- **Mdb** – DataFrame of Mash comparison results; make sure loaded not as categories
- **Cdb** – DataFrame of Clustering results
- **linkage** – Result of scipy.cluster.hierarchy.linkage
- **threshold** (*optional*) – Line to plot on x-axis
- **plot_dir** (*optional*) – Location to store plot

Returns Makes and shows plot

drep.d_analyze.**plot_MASH_vs_ANIn_ani**(*Mdb*, *Ndb*, *exclude_zero_MASH=True*)
Makes plot and retuns plt.cgf()

All parameters are obvious

drep.d_analyze.**plot_MASH_vs_ANIn_cov**(*Mdb*, *Ndb*, *exclude_zero_MASH=True*)
Makes plot and retuns plt.cgf()

All parameters are obvious

drep.d_analyze.**plot_MASH_vs_len**(*Mdb*, *Ndb*, *exclude_zero_MASH=True*)
Makes plot and retuns plt.cgf()

All parameters are obvious

drep.d_analyze.**plot_MASH_vs_secondary_ani**(*Mdb*, *Ndb*, *Cdb*, *exclude_zero_MASH=True*)
Makes plot and retuns plt.cgf()

All parameters are obvious

drep.d_analyze.**plot_binscoring_from_wd**(*wd*, *plot_dir*, ***kwargs*)
From the wd and kwargs, call plot_winner_scoring_complex

Parameters

- **wd** – WorkDirectory
- **plot_dir** (*optional*) – Location to store figure

Returns Shows plot, makes a plot in the plot_dir

drep.d_analyze.**plot_clustertest**(*linkage*, *names*, *wd*, ***kwargs*)
DEPREICATED

names can be gotten like: db = db.pivot("reference","querry","ani") names = list(db.columns)

drep.d_analyze.**plot_scatterplots**(*Mdb*, *Ndb*, *Cdb*, *plot_dir=False*)
Make scatterplots comparing genome comparison algorithms

- plot_MASH_vs_ANIn_ani(Mdb, Ndb) - Plot MASH_ani vs. ANIn_ani (including correlation)
- plot_MASH_vs_ANIn_cov(Mdb, Ndb) - Plot MASH_ani vs. ANIn_cov (including correlation)
- plot_ANIn_vs_ANIn_cov(Mdb, Ndb) - Plot ANIn vs. ANIn_cov (including correlation)
- plot_MASH_vs_len(Mdb, Ndb) - Plot MASH_ani vs. length_difference (including correlation)

- plot_ANIn_vs_len(Ndb) - Plot ANIn vs. length_difference (including correlation)

Parameters

- **Mdb** – DataFrame of Mash comparison results
- **Ndb** – DataFrame of secondary clustering results
- **Cdb** – DataFrame of Clustering results
- **plot_dir** (*optional*) – Location to store plot

Returns Makes and shows plot

drep.d_analyze.**plot_scatterplots_from_wd**(*wd*, *plot_dir*, ***kwargs*)
From the wd and kwargs, call plot_scatterplots

Parameters

- **wd** – WorkDirectory
- **plot_dir** (*optional*) – Location to store figure

Returns Shows plot, makes a plot in the plot_dir

drep.d_analyze.**plot_secondary_dendrograms_from_wd**(*wd*, *plot_dir*, ***kwargs*)
From the wd and kwargs, make the secondary dendrograms

Parameters

- **wd** – WorkDirectory
- **plot_dir** (*optional*) – Location to store figure

Returns Makes plot

drep.d_analyze.**plot_secondary_mds_from_wd**(*wd*, *plot_dir*, ***kwargs*)
Make a .pdf of MDS of each cluster

Parameters

- **wd** – WorkDirectory
- **plot_dir** (*optional*) – Location to store figure

Returns Makes plot

drep.d_analyze.**plot_winner_scoring_complex**(*Wdb*, *Sdb*, *Cdb*, *Gdb*, *plot_dir=False*, ***kwargs*)
Make a plot showing the genome scoring for all genomes

Parameters

- **Wdb** – DataFrame of winning dereplicated genomes
- **Sdb** – Scores of all genomes
- **Cdb** – DataFrame of Clustering results
- **Gdb** – DataFrame of genome scoring information
- **plot_dir** (*optional*) – Location to store plot

Returns makes plot

drep.d_analyze.**plot_winners**(*Wdb*, *Gdb*, *Wndb*, *Wmdb*, *Widb*, *plot_dir=False*, ***kwargs*)
Make a bunch of plots about the de-replicated genomes

THIS REALLY NEEDS IMPROVED UPON

drep.d_analyze.**plot_winners_from_wd**(*wd*, *plot_dir*, ***kwargs*)
From the wd and kwargs, call plot_winners

Parameters

- **wd** – WorkDirectory
- **plot_dir** – Location to store figure

Returns Shows plot, makes a plot in the plot_dir

2.8.5 drep.WorkDirectory

This module provides access to the workDirectory

The directory layout:

```
workDirectory
./data
...../MASH_files/
...../ANIn_files/
...../gANI_files/
...../Clustering_files/
...../checkM/
........./genomes/
........./checkM_outdir/
...../prodigal/
./figures
./data_tables
...../Bdb.csv  # Sequence locations and filenames
...../Mdb.csv  # Raw results of MASH comparisons
...../Ndb.csv  # Raw results of ANIn comparisons
...../Cdb.csv  # Genomes and cluster designations
...../Chdb.csv # CheckM results for Bdb
...../Sdb.csv  # Scoring information
...../Wdb.csv  # Winning genomes
./dereplicated_genomes
./log
...../logger.log
...../cluster_arguments.json
```

class drep.WorkDirectory.**WorkDirectory**(*location*)
Bases: object

Object to interact with the workDirectory

Parameters **location** (*str*) – location to make the workDirectory

firstLevels = ['data', 'figures', 'data_tables', 'dereplicated_genomes', 'log']

get_cluster(*name*)
Get the cluster passed in

Parameters **name** – name of the cluster

Returns cluster

get_db(*name*, *return_none=True*, *forPlotting=False*)
Get database from self.data_tables

Parameters

- **name** – name of dataframe
- **return_none** – if True will return None if database not found; otherwise assert False
- **forPlotting** – if True don't do fancy dType loading; it messes with order of names for dendrograms

get_dir(*dir*)
Get the location of one of the named directory types

Parameters **dir** – Name of directory to find

Returns Location of requested directory

Return type string

get_loc(*what*)
Get the location of Things

Parameters **what** – string of what to get the location of

Returns location of what

Return type string

get_primary_linkage()
Get the primary linkage cluster

hasDb(*db*)
If db is in the data_tables, return True

import_arguments(*loc*)
Given the location of the log directory, load it

import_clusters(*loc*)
Given the location of the cluster files, load them

import_data_tables(*loc*)
Given the location of the datatables, load them

load_cached()
The wrapper to load everything it has into attributes

make_fileStructure()
Make the top level file structure

store_db(*db*, *name*, *overwrite=None*)
Store a dataframe in the workDirectory

Will make a physical copy in the datatables folder

Parameters

- **db** – pandas dataframe to store
- **name** – name to store it under (will add .csv automatically)
- **overwrite** – if True, overwrite if DataFrame with same name already exists

store_special(*name*, *thing*)
Store special items in the work directory

Parameters

- **name** – what to store
- **thing** – actual thing to store

Python Module Index

d

drep.d_analyze, 36
drep.d_choose, 34
drep.d_cluster, 34
drep.d_filter, 32
drep.WorkDirectory, 40

Index

A

add_centrality() (*in module drep.d_choose*), 35

C

calc_centrality_from_scratch() (*in module drep.d_choose*), 35
calc_dist() (*in module drep.d_analyze*), 36
calc_fasta_length() (*in module drep.d_filter*), 32
calc_genome_info() (*in module drep.d_filter*), 32
calc_n50() (*in module drep.d_filter*), 32
chdb_to_genomeInfo() (*in module drep.d_filter*), 32
choose_winners() (*in module drep.d_choose*), 35
cluster_test_wrapper() (*in module drep.d_analyze*), 37

D

d_analyze_wrapper() (*in module drep.d_analyze*), 37
d_choose_wrapper() (*in module drep.d_choose*), 35
d_filter_wrapper() (*in module drep.d_filter*), 33
drep.d_analyze (*module*), 36
drep.d_choose (*module*), 34
drep.d_cluster (*module*), 34
drep.d_filter (*module*), 32
drep.WorkDirectory (*module*), 40

F

fancy_dendrogram() (*in module drep.d_analyze*), 37
filter_bdb() (*in module drep.d_filter*), 33
firstLevels (*drep.WorkDirectory.WorkDirectory attribute*), 40

G

gen_color_dictionary() (*in module drep.d_analyze*), 37
gen_color_list() (*in module drep.d_analyze*), 37
get_cluster() (*drep.WorkDirectory.WorkDirectory method*), 40
get_db() (*drep.WorkDirectory.WorkDirectory method*), 40
get_dir() (*drep.WorkDirectory.WorkDirectory method*), 41
get_highest_self() (*in module drep.d_analyze*), 37
get_loc() (*drep.WorkDirectory.WorkDirectory method*), 41
get_primary_linkage() (*drep.WorkDirectory.WorkDirectory method*), 41

H

hasDb() (*drep.WorkDirectory.WorkDirectory method*), 41

I

import_arguments() (*drep.WorkDirectory.WorkDirectory method*), 41
import_clusters() (*drep.WorkDirectory.WorkDirectory method*), 41
import_data_tables() (*drep.WorkDirectory.WorkDirectory method*), 41

L

load_cached() (*drep.WorkDirectory.WorkDirectory method*), 41
load_extra_weight_table() (*in module drep.d_choose*), 35

M

make_fileStructure() (*drep.WorkDirectory.WorkDirectory method*), 41

mash_dendrogram_from_wd() (*in module drep.d_analyze*), 37

N

normalize() (*in module drep.d_analyze*), 37

P

pick_winners() (*in module drep.d_choose*), 36
plot_ANIn_vs_ANIn_cov() (*in module drep.d_analyze*), 37
plot_ANIn_vs_len() (*in module drep.d_analyze*), 37
plot_binscoring_from_wd() (*in module drep.d_analyze*), 38
plot_clustertest() (*in module drep.d_analyze*), 38
plot_MASH_dendrogram() (*in module drep.d_analyze*), 38
plot_MASH_vs_ANIn_ani() (*in module drep.d_analyze*), 38
plot_MASH_vs_ANIn_cov() (*in module drep.d_analyze*), 38
plot_MASH_vs_len() (*in module drep.d_analyze*), 38
plot_MASH_vs_secondary_ani() (*in module drep.d_analyze*), 38
plot_scatterplots() (*in module drep.d_analyze*), 38
plot_scatterplots_from_wd() (*in module drep.d_analyze*), 39
plot_secondary_dendrograms_from_wd() (*in module drep.d_analyze*), 39
plot_secondary_mds_from_wd() (*in module drep.d_analyze*), 39
plot_winner_scoring_complex() (*in module drep.d_analyze*), 39
plot_winners() (*in module drep.d_analyze*), 39
plot_winners_from_wd() (*in module drep.d_analyze*), 39

R

run_checkM() (*in module drep.d_filter*), 33
run_prodigal() (*in module drep.d_filter*), 34

S

sanity_check() (*in module drep.d_filter*), 34
score_genomes() (*in module drep.d_choose*), 36
score_row() (*in module drep.d_choose*), 36
store_db() (*drep.WorkDirectory.WorkDirectory method*), 41
store_special() (*drep.WorkDirectory.WorkDirectory method*), 41

V

validate_chdb() (*in module drep.d_filter*), 34

W

WorkDirectory (*class in drep.WorkDirectory*), 40