USEARCH

http://www.drive5.com/usearch

robert@drive5.com

Version 2.1
July 7, 2010

Table of Contents

Introduction

USEARCH is a database search algorithm that is hundreds of times faster than BLAST in some applications.

UCLUST is a clustering algorithm based on USEARCH that is significantly better than CD-HIT: it is faster, uses less memory, is more sensitive, allows clustering at lower identities, can cluster much larger datasets and produces higher-quality clusters (higher average identity between member sequence and the 'seed' sequence that defines the cluster).

USEARCH, UCLUST and some other algorithms are implemented in a program called uclust. It combines functionality roughly equivalent to BLASTN, BLASTP, MEGABLAST, BLASTCLUST, CD-HIT, CD-HIT-EST, CD-HIT-2D and CD-HIT-EST-2D in a single program. The clustering method was implemented first, which is why the program name and manual tends to emphasize UCLUST rather than USEARCH. But the search algorithm is more fundamental and I believe it will probably turn out to be more widely used in practice, so I'm "re-positioning" UCLUST as USEARCH.

Installation

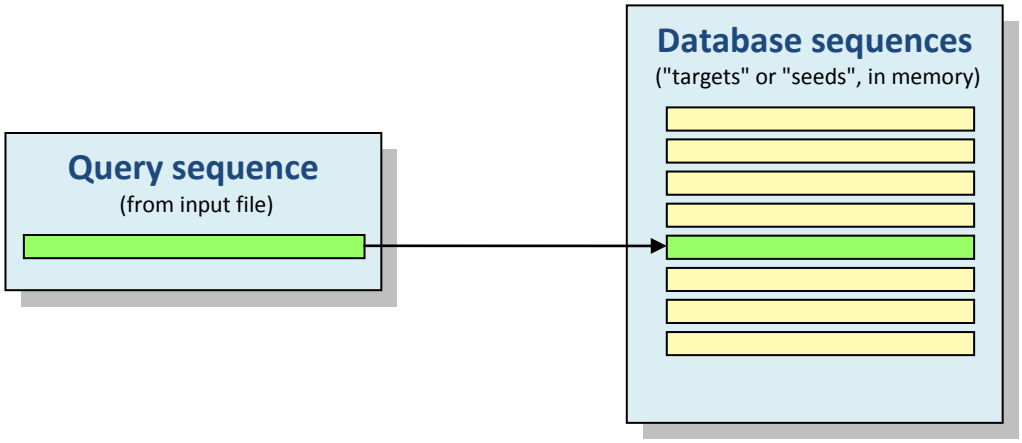
The uclust program is distributed as a stand-alone binary (executable file). The binary is self-contained: it does not require configuration files, environment variables, third-party libraries or other external dependencies. There is no setup script or installer because they're not needed. All you need to do is download or copy the binary file to a directory that is accessible from the computer where you want to run the code. For more information, please see http://drive5.com/cmdline.html.

UCLUST overview

UCLUST is a flexible program that can be used in many different ways, which can be confusing to new users. Here is an overview of how UCLUST works to help you get a picture of what's going on.

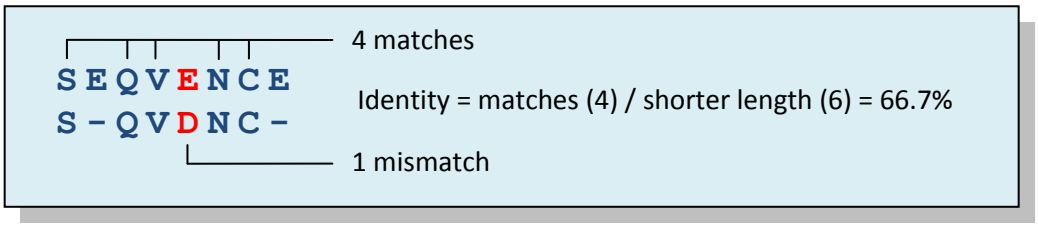
Searching

The core step is the USEARCH algorithm that searches a database stored in memory.



A query sequence matches a database sequence if the identity is high enough. Identity is calculated from a global alignment, i.e. an alignment that includes all letters from both sequences. This differs from BLAST and most other database search programs, which search for local matches. A database sequence is sometimes called a *target* or a *seed*, because a cluster of similar sequences can be grown from it.

The minimum identity is set by the --id option, e.g. --id 0.97 means that the global alignment must have at least 97% identity. Identity is computed as the number of matching (identical) letters divided by the length of the shorter sequence, as shown below.



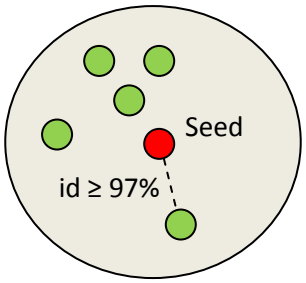
Other definitions of identity might be useful—if you would like additional options, let me know.

By default, USEARCH stops searching when it finds a match. Usually USEARCH finds the best match first, but this is not guaranteed. If it is important to find the best possible match (i.e., the database sequence with highest identity), then you can increase the --maxaccepts option, which defaults to 1. If you want all matches to be reported rather than the best match, then you can use the --allhits option, which will have no effect unless you also set maxaccepts > 1.

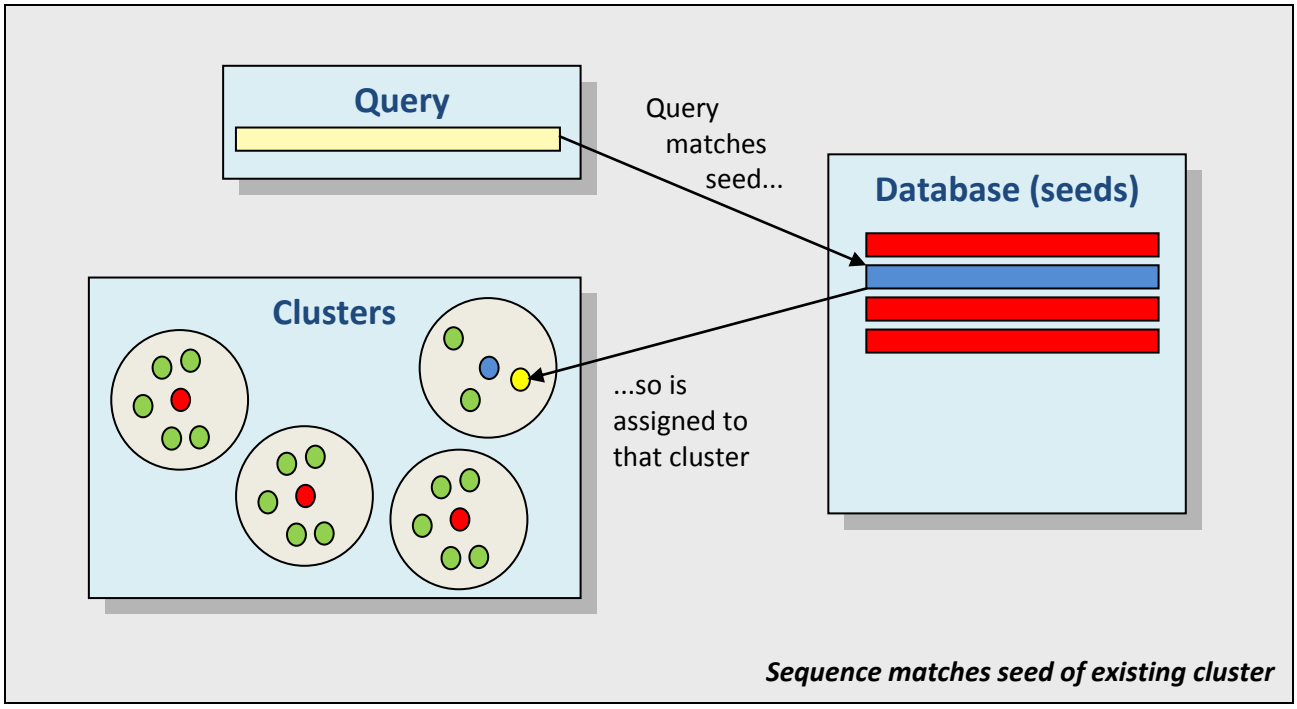
USEARCH also stops searching if it fails to find a match. By default, it gives up after eight failed attempts. Database sequences are tested in an order that correlates well (but not exactly) with decreasing identity. This means that the more sequences get tested, the less likely it is that a match will be found later, so giving up early doesn't miss a potential hit very often. You can set the maximum number to try using the --maxrejects option. With very high and very low identity thresholds, increasing maxrejects can significantly improve sensitivity. Here, a rule of thumb is that low identity is below 60% for amino acid sequences or 80% for nucleotides, high identity is 98% or more.

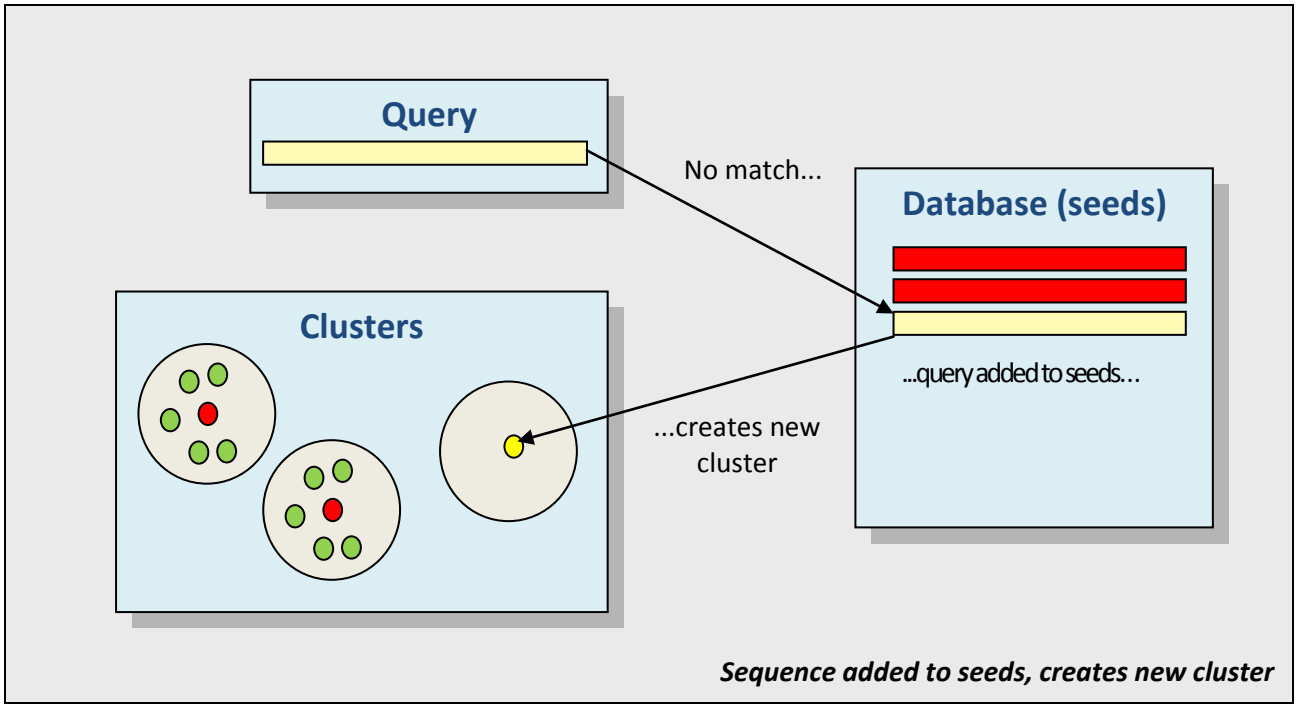
Clustering

The UCLUST algorithm uses USEARCH internally in order to assign sequences to clusters. Each cluster is defined by a representative sequence called a *seed*. Each sequence in a cluster matches the seed according to the identity threshold, e.g. 97%.

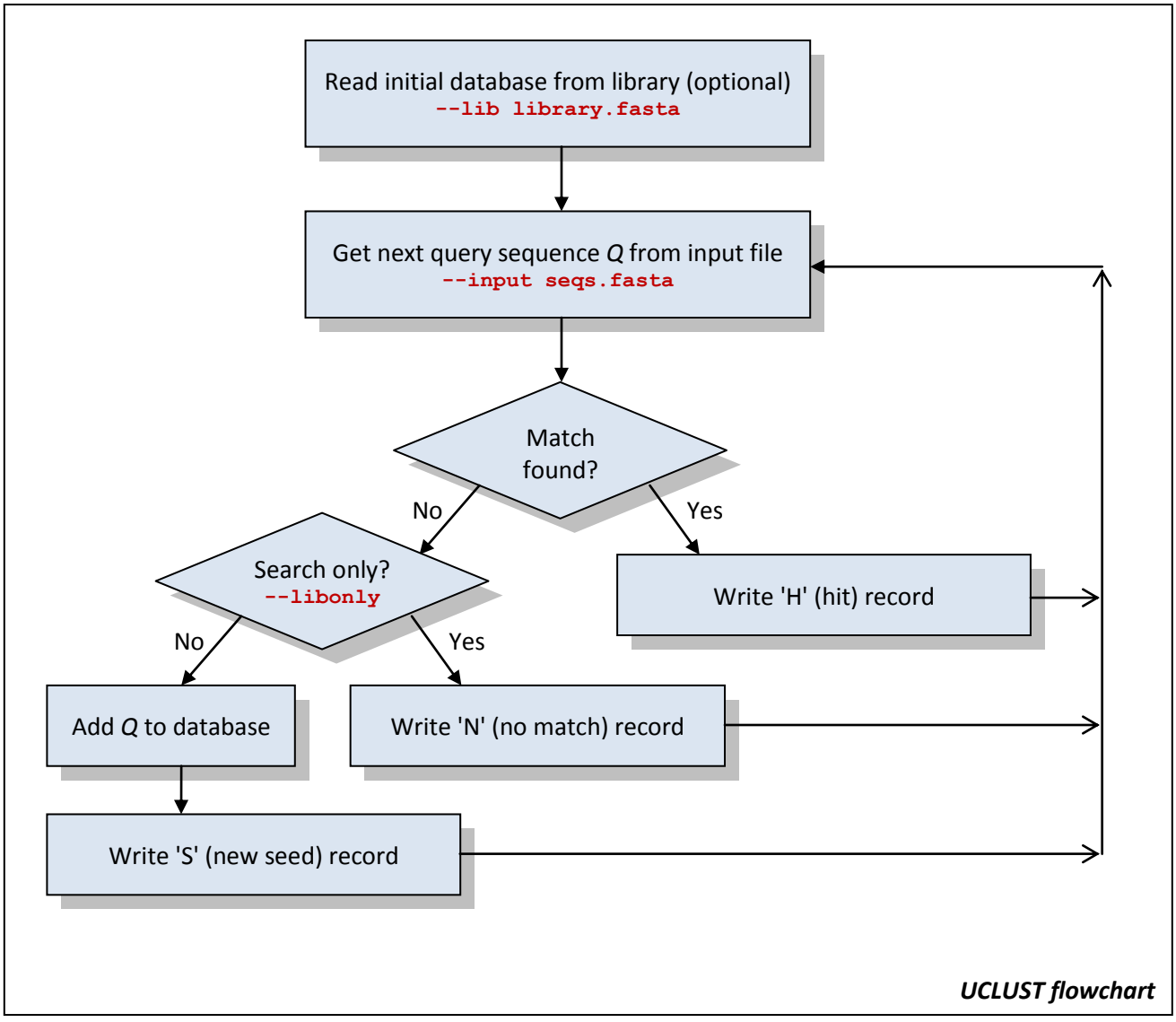


UCLUST performs *de novo* clustering by starting with an empty database in memory. Query sequences are processed in input order. If a match is found to a database sequence, then the query is assigned to that cluster (first figure below), otherwise the query becomes the seed of a new cluster (second figure below).





Only the seeds need to be stored in memory because other cluster members don't affect how new query sequences are processed. This is an advantage for large datasets because the amount of memory needed and the number of sequences to search are reduced. However, this design may not be ideal in some scenarios because it allows non-seed sequences in the same cluster to fall below the identity threshold. I plan to add other clustering methods to future versions of UCLUST: these will probably be slower for large datasets, but may be useful in some applications. Please let me know if there are new features you would like to have.



UCLUST flowchart

Basic usage

In the following, UCLUST refers to the program in general, and uclust stands for the binary file name. When typing a command, you should replace uclust with the binary file name on your system, e.g. uclust2.1.598_linuxi86_64.

Database search

In database search mode, there are two input files: the database and the query set. Each sequence in the query set is compared with the database, which is searched for a match exceeding a specified identity threshold. The database (also called a library) file is specified using the --db option, e.g.:

```
uclust --query query.fasta --db database.fasta --uc results.uc --id 0.90
```

In previous versions of UCLUST, database search mode was obtained by using this command line:

```
uclust --input query.fasta --lib database.fasta --uc results.uc --id 0.90 --libonly
```

The input set was specified by --input and the database by --lib, and the --libonly option specified that unmatched sequences do not extend the database. This usage is retained for backwards compatibility and will give identical results.

De novo clustering

UCLUST generates clusters containing similar sequences. A similarity threshold is specified, say --id 0.9 which means 90% identity. Each cluster has a representative sequence (its *seed*); all sequences in a cluster are required to have at least the given identity with the seed. Typical usage is:

```
uclust --sort seqs.fasta --output seqs_sorted.fasta
uclust --input seqs_sorted.fasta --uc results.uc --id 0.90
```

The first step is to sort the sequences in a way that is appropriate for your data. Decreasing length is often a suitable order, which is supported by uclust through the --sort option. If you sort by some other criteria, then you must specify the --usersort option to the clustering step. Input is a FASTA file, results are generated in a .uc (UCLUST format) file.

Simultaneous search and clustering

If --input is used instead of --query, then by default sequences that do not match the library become seeds for new clusters. Later sequences in the input file may be matched to these new seeds. This mode simultaneously matches sequences to a database and clusters those that do not match. Input sequences should be sorted appropriately as for *de novo* clustering.

```
uclust --input seqs_sorted.fasta --lib database.fasta --uc results.uc --id 0.90
```

The FASTA file for the database is not updated if new seeds are added. New seeds are indicated by 'S' records in the .uc file. See the discussion of the --uc2fasta option for how to create an updated database file.

Fast, approximatemultiple alignment of clusters

UCLUST can create a multiple alignment of each cluster. This requires three steps: 1. clustering, 2. conversion to FASTA (--uc2fasta), 3. inserting additional gaps (--staralign).

```
uclust --input seqs_sorted.fasta --uc results.uc --id 0.90
uclust --uc2fasta results.uc --input seqs_sorted.fasta --output results.fasta
uclust --staralign results.fasta --output aligned.fasta
```

This method emphasizes speed over alignment quality. It is not intended to replace slower but more accurate methods like MUSCLE. When sequence identity is reasonably high, the alignment will be good enough to be informative. Note that in addition to creating a multiple alignment, a consensus sequence is generated for each cluster. This can be useful for high-throughput evaluation of cluster quality. See below ("Clumping") for a method that can create high-quality alignments of very large sets.

Hierarchical clustering

Hierarchical clustering can be performed by repeatedly re-clustering at decreasing identities. Each time, the input sequences are the seeds found in the previous iteration. For example, the following commands cluster at 99, 97 and 90% identity.

```
uclust --input reads.sorted.fasta --id 0.99 --uc 99.uc
uclust --uc2fasta 99.uc --types S --input reads.sorted.fasta --output seeds99.fasta
uclust --input seeds99.fasta --id 0.97 --uc 97.uc
uclust --uc2fasta 97.uc --types S --input seeds99.fasta --output seeds97.fasta
uclust --input seeds97.fasta --id 0.90 --uc 90.uc
uclust --uc2fasta 90.uc --types S --input seeds97.fasta --output seeds90.fasta
```

Equivalent results can be obtained in a single step using the --uhire command, as follows.

```
uclust --uhire reads.sorted.fasta --hireout results.hire --ids 99,97,90
```

This command is explained in more detail in a later section. The --ids option specifies a series of percent identity thresholds to use. Since commands are usually significant to command shells, the --ids option should usually be quoted, e.g.:

```
uclust --uhire reads.sorted.fasta --hireout results.hire --ids "99,97,90"
```

Clumping

UCLUST supports a type of clustering I call *clumping*, which to the best of my knowledge has not previously been described, though the idea is simple and is probably not new. The goal is to create clusters ("clumps") of a given size. Members of a given clump should be more similar to each other than to members of other clumps. The motivation is to divide the input into subsets that are tractable for more expensive methods, say those requiring all-vs.-all comparisons. The subsets should be as large as possible to leverage the accuracy of the expensive method. One application of clumping is the creation of very large multiple alignments. More details are given in later sections.

Chimera detection

UCLUST implements algorithms for detecting chimeric sequences. This can be done de novo, or by using a reference database that is assumed to be free of chimeras. More details are given in a later section.

Input data

Input to UCLUST is generally in the form of FASTA files containing nucleotide or amino acid sequences. The library (database) is stored in memory. Input sequences are processed in the order they appear, allowing files of arbitrary size to be read sequentially with minimal use of memory. Input sequences for *de novo* clustering should therefore be ordered so that the most appropriate seed sequence for a cluster is likely to be found before other members. For example, ordering by decreasing length is desirable when both complete and fragmented sequences are present, in which case full-length sequences are generally preferred as seeds since a fragment may attract longer sequences that are dissimilar in terminal regions which do not align to the seed, as in the following example.

```
Seed:         THESEED
First hit:    THESEEDINSERTED
Second hit:   THESEEDTERMINAL
```

The two hits are both 100% identical to the seed in a pair-wise alignment (see later sections for a more detailed discussion of identity). However, the hits are extended with different terminal regions (red) and therefore have only about 50% identity to each other.

In other cases, long sequences may make poor seeds. For example, with some high-throughput sequencing technologies longer reads tend to have higher error rates, and in such cases sorting by decreasing read quality score may give better results. For 16S or 18S sequences, sorting by decreasing abundance may give significantly better results (see below for more on this topic).

If new seeds may be identified (de novo clustering or database clustering without --db or --libonly), then UCLUST checks that input sequences are sorted by decreasing length. This check can be disabled by specifying the --usersort option, which specifies that input sequences have been pre-sorted in a way that might not be decreasing length.

Sorting input by length

If required, sorting by decreasing length is done in a separate step as follows:

```
uclust --sort input.fasta --output input_sorted.fasta
```

The current implementation of --sort loads all sequences into memory for faster speed, so requires that the available memory be at least as big as the input file. Larger sets can be sorted using a merge sort:

```
uclust --mergesort input.fasta --output input_sorted.fasta --split 500.0
```

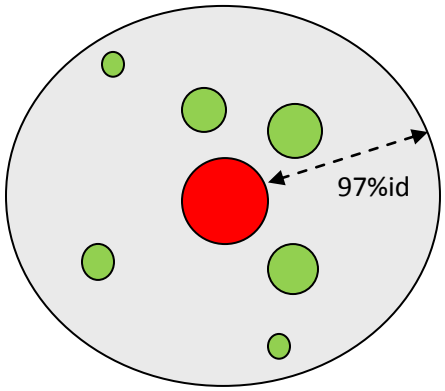
The --split option (default 1000.0) specifies the number of megabytes to use for each partition of the input file. Typically, the maximum RAM needed for the sort will be a bit more than this, but in a worst-case scenario can be closer to 2x the --split value, so a conservative choice is to use about half the physically available memory. Smaller values tend to give slower speeds.

There is no need to sort sequences if --db or --libonly is specified (because no new seeds are created).

If you want to sort by some other criteria, then you will need to write your own program or script to do this.

OTU identification and sorting input by abundance

The advantage of sorting by decreasing length is that this tends to prevent fragments from becoming seeds. However, in some applications this is not optimal because the longest sequences may have anomalous insertions, for example due to sequencing errors or a diverged gene that is not typical of its family. If the goal of clustering is to identify taxonomic units such as species ("OTUs") in a high-throughput sequencing experiment based on a single gene (e.g. 16S), then a better solution may be to sort by decreasing abundance. The most abundant sequence is likely to be a true biological sequence, while less common sequences may be artifacts due to sequencing error or PCR artifacts such as chimeras, as illustrated in the following figure. This shows the cluster for a single species; the red dot represents reads of the true sequence of the species. A dot indicates a unique sequence, the size of the dot indicates its abundance, i.e. the number of identical (or very similar) reads having that sequence. The longest sequence in the figure is likely to be one of the outliers, and will give a less accurate OTU—imagine drawing a circle of radius of size 97% around one of the outlying dots and you will see that some reads that belong to the species will be incorrectly excluded.



Sorting by abundance can be accomplished as follows. First cluster at a high identity, say --id 1.0 or 0.99. The size of the clusters can then be obtained from the C records in the .uc file (see below for more information on the .uc file format). This can be done using a series of commands like the following.

```
uclust --sort reads.fasta --output reads.sorted.fasta
uclust --input reads.sorted.fasta --id 0.99 --uc 99.uc
grep "^C" 99.uc | sort -nrk3 > c.uc # sort by decreasing cluster size
uclust --uc2fasta c.uc --types C --output c.fasta
```

Grepping on the first letter (grep "^C" above) selects a single type of record from the .uc file. For details of the .uc file format, see below. Finally, OTUs can be identified by clustering, e.g. for species 97% is typically used:

```
uclust --input c.fasta --id 0.97 --uc 97.uc --usersort
```

Note the --usersort option, which is required for de novo clustering when input sequences are not sorted by decreasing length.

UCLUST file format

The native UCLUST format (.uc) is a tab-separated text file. Each line is either a comment (starts with #) or a record. Each input sequence has at least one record; additional record types give information about clusters. The cluster number appears in every record. If an input sequence matched a target sequence, then the alignment and the identity computed from that alignment are also provided. A compressed representation of the alignment is used to save disk space. Records are appended to the output file as they are generated in order to minimize memory use, and sequences therefore appear in the same order as the input file.

Some example records:

Type	Cluster	Size	%Id	Strand	Qlo	Tlo	Alignment	Query	Target
S	0	292	*	*	*	*	*	AH70_12410	*
H	0	292	99.7	+	0	0	292M	EN70_12566	AH70_12410
S	1	292	*	*	*	*	*	EX70_12567	*
H	1	292	98.2	+	0	0	292M	AH70_12410	EX70_12567

Each record has ten fields, separated by tabs.

Type	Record type
Cluster	Cluster number
Size	Sequence length or cluster size
%Id	Identity to the seed (as a percentage), or * if this is a seed.
Strand	+ (plus strand), - (minus strand), or . (for amino acids).
Qlo	0-based coordinate of alignment start in the query sequence.
Tlo	0-based coordinate of alignment start in target (seed) sequence. If minus strand, Tlo is relative to start of reverse-complemented target.
Alignment	Compressed representation of alignment to the seed (see below), or * if a seed.
Query	FASTA label of query sequence.
Target	FASTA label of target (seed / library / database) sequence, or * if a seed.

Record types are:

- L Library seed (generated only if a match is found to this seed).
- S New seed.
- H Hit, also known as an accept; i.e. a successful match.
- D Library cluster.
- C New cluster.
- N Not matched (a sequence that didn't match library with --libonly specified).
- R Reject (generated only if --output_rejects is specified).

Records of type C and D are used when clustering. The Size field contains the cluster size, i.e. the number of sequences in the cluster including the seed, and %Id is the average identity of non-seed sequences to the seed. Otherwise, Size is the sequence length and %Id is the identity of the pair-wise alignment of this sequence to the seed. For Library clusters (D), records are only output if Size > 1, i.e. library sequences with no matches are not output. A library seed records (L) are output only if a hit is found to this seed. This saves writing a large number of records for library sequences that are not matched, but means that cluster numbers in the .uc file may not be consecutive (because UCLUST internally assigns a cluster number to every library seed, whether or not it is matched).

Rejections (R) are sequences that were aligned to a seed but found to have an identity below the threshold. Rejections are not output unless --output_rejects is specified. Using --output_rejects may increase the size of the .uc file significantly. Rejection records are mainly useful when trouble-shooting unexpected results.

The alignment is compressed using run-length encoding, as follows. Each column in the alignment is classified as M, D or I:

Code	Name	Query sequence	Seed sequence
M	Match	Letter	Letter
D	Delete	Gap	Letter
I	Insert	Letter	Gap

Here, "match" simply means a letter-letter column; the letters may or may not be identical. If there are *n* consecutive columns of type C, this is represented as *n*C. For example, 123M is 123 consecutive matches. As a special case, if *n*=1 then *n* is omitted. So for example, D5M2I3M represents an alignment of this form:

Query sequence	`-XXXXXXXXXX`
Seed sequence	`XXXXXX--XXX`
Column type	`DMMMMMIIMMM`

If a line in the output file starts with #, it is a comment and parser scripts should ignore it.

Records in the .uc file appear in the same order as the input sequences. You can sort the file using the standard Linux sort command, as follows:

```
sort -nk2 results.uc > results_sorted.uc
```

You can sort first by cluster number then by identity using:

```
sort -n -k2 -k4 results.uc > clusters.sorted.uc
```

UCLUST can also do the sort:

```
uclust --sortuc results.uc --output results_sorted.uc
```

However, the current implementation reads the entire file into memory, so may fail for very large sequence sets.

FASTA format

UCLUST re-formats both labels and sequences when generating FASTA format output.

Labels look like this:

```
>43|99.7%|AH70_12410
```

Here, 43 is the cluster number and 99.7% is the identity to the seed. The identity will be shown as * for the seed:

```
>43|*|AH70_12200
```

If a .uc record has an alignment, then the query sequence is re-formatted to indicate its pair-wise alignment to the seed. Gaps indicate deletions relative to the seed, lower-case indicates insertions relative to the seed. Here is an example:

```
>43|99.7%|TheSeed
SEQVENCE
>43|96.0%|NonSeed
S-QLENnCE
```

This represents the following pair-wise alignment:

```
TheSeed   SEQVEN-CE
NonSeed   S-QLENNCE
```

You can convert UCLUST to FASTA format as follows:

```
uclust --uc2fasta results.uc --input seqs.fasta --output results.fasta [--types XYZ...]
```

Here, seqs.fasta must be the same input file used when generating results.uc. The --types option specifies which record types to convert, default is SH (seeds and hits). It is not valid to use L or D in --types because these refer to library sequences and the current implementation extracts sequences from the input set only. Using --types enables some convenient idioms, as in the following examples.

Create non-redundant database

```
uclust --input seqs.fasta --uc results.uc --id 0.90
uclust --uc2fasta results.uc --types S --output nr.fasta
```

Assign to non-redundant database, cluster unmatched sequences and create new library

```
uclust --input seqs.fasta --lib nr.fasta --uc results.uc --id 0.90
uclust --uc2fasta results.uc --types S --output newlib.fasta
```

Multiple alignment FASTA format

(See Clumping for information on creating high-quality alignments). Alignments generated by --uc2fasta are saved in a specialized FASTA format. You can convert to a more conventional multiple alignment format by using --staralign:

```
uclust --staralign results.fasta --output star.fasta
```

The results.fasta file must be sorted by cluster number.

Gaps are added so that each sequence in a given cluster has the same length. Letters that are aligned to the same position in the seed appear in the same column and are in upper case. Deletions relative to the seed are indicated by dashes. Insertions relative to the seed are indicated in lower-case, and should not be considered aligned to each other. The seed sequence is the last sequence is a special case that

represents a consensus sequence. If the position is 100% conserved, i.e. if all letters in that column are identical, then an upper case letter is used. Otherwise, seed letters are lower-case.

BLAST-like format

Alignments generated during clustering or database search can be saved in a human-readable BLAST-like format by using the --blastout option, e.g.:

```
uclust --input seqs.fasta --lib greengenes.fasta --libonly --blastout hits.blast
```

Since this format is rather verbose, the file size will be much larger than the corresponding .uc file. This format may be changed in future versions, so it is recommended that parsers use the FASTA output generated by --fastapairs instead (see next).

FASTA alignment format

Alignments generated during clustering or database search can be saved in FASTA format by using the --blastout option, e.g.:

```
uclust --input seqs.fasta --lib greengenes.fasta --libonly --fastapairs hits.fasta
```

This format is probably the most convenient for parsers that need to derive information from explicit alignments. Pairs are separated by blank lines, to make the file easier to inspect visually. The query sequence is first, the target (seed, database) sequence is second. If the input sequences are nucleotides, then a + or - is appended to the label of the target sequence to indicate the strand. If the strand is - (reverse strand match), then the target sequence is reverse-complemented.

CD-HIT format

The CD-HIT .clstr format is supported for the benefit of code already written for that format. You can convert UCLUST format to and from .clstr as follows:

```
uclust --uc2clstr results.uc --output results.clstr [--amino]

uclust --clstr2uc results.clstr --output results.uc
```

The --amino option to --uc2clstr specifies that sequence lengths in the .clstr file should be written with the 'aa' suffix, e.g.:

```
>Cluster 83
0       297aa, >DTS1061058149802r_891_1_1... *
```

By default, the 'nt' suffix is used, so this would appear as:

```
>Cluster 83
0       297nt, >DTS1061058149802r_891_1_1... *
```

"Exact" and "optimal" clustering

An "optimal" variant of the algorithm is specified by --optimal, which is equivalent to these options:

```
--maxaccepts 0 --maxrejects 0 --nowordcountreject
```

This guarantees that every seed will be aligned to the query, and that every sequence will therefore be assigned to the highest-identity seed that passes the identity threshold (*t*). All pairs of seeds are guaranteed to have identity < *t*. The number of seeds is guaranteed to be the minimum that can be discovered by greedy list removal, though it is possible that the number of clusters could be reduced by using a different set of seeds.

An "exact" variant of the algorithm is selected by --exact, which is equivalent to:

```
--maxaccepts 1 --maxrejects 0 --nowordcountreject
```

This guarantees that a match will be found if one exists, but not that the best match will be found.

The exact and optimal variants are guaranteed to find the minimum possible of clusters and both guarantee that all pairs of seeds have identities < *t*. Exact clustering will be faster, but may have lower average identity of non-seeds to seeds. "Optimal" and "exact" are misleading names that ideally should be changed. They are retained for backwards compatibility.

Searching with reverse-complement (minus) strand

By default, UCLUST seeks nucleotide matches in the same orientation (i.e., plus strand only). You can enable both plus and minus strand matching by using --rev. In the current implementation, using --rev approximately doubles memory use but results in only small increases in execution time. Optimizations are possible that would avoid most of the increase in memory, but would be a fair amount of work to implement and so far do not appear to be worth the effort.

Hierarchical clustering

The --uhire command performs hierarchical clustering, as follows.

```
uclust --uhire reads.sorted.fasta --hireout results.hire --clumpout results.clump
   --clumpfasta filenameprefix --maxclump 1000 --ids id1,id2...,idN
```

At least one output option must be given, i.e. at least one of --hireout, --clumpout or --clumpfasta. The --maxclump option gives the maximum number of sequences in a clump (default 1000). The --ids option gives the percent identities of each level in the hierarchy. The default is 99,98,95,90,85,80,70,50,35. Note that --ids uses percentages (0 to 100), compared to --id which uses fractional identities (0.0 to 1.0). Values are separated by commas. Since commas are significant to most command shells, the value of the --ids argument should usually be quoted. If the --clumpfasta option is given, each clump is written to a file named clump.0, clump.1, clump.2 etc., prefixed by the --filenameprefix option. This will typically be a directory name. E.g., you might do this:

```
uclust --uhire reads.sorted.fasta --clumpfasta myclumps/ --maxclump 256
```

Note the '/' at the end of the prefix. This is not required, but if present specifies that clump files are to be stored in the given directory. Sequences for each clump will be stored in these files:

```
myclumps/clump.0
myclumps/clump.1
..etc..
```

In addition to the clumps, a file named 'master' will also be written. This contains the longest sequence in each clump. It can be used for creating large multiple alignments, as explained shortly below.

The .hire file format

The .hire format is designed to be easily parsed by a scripting language and to avoid very long lines as found in mothur files. If you would like a script to convert .hire to mothur format, please let me know.

A .hire file is a text file.

The first line is the number of levels (K), i.e. the number of ids specified in the --ids option.

The second line is the number of sequences (N).

The following N lines specify sequences. Each line contains three tab-separated fields, for example:

    37261   167 GF2FOAC01BL6E9

    The first field is the sequence ID, an integer 0, 1 ... (N-1). This is redundant, but should be used by parsers to check that they are in synch with the file.

    The second field is the sequence length in letters.

    The third field is the sequence label from the FASTA file.

Following the last sequence (ID=N-1) will be K levels. Each level is specified as follows.

The first line in a LEVEL is a record with four fields, for example:

    LEVEL   6   9 70.0

The first field is always the text "LEVEL". The remaining fields are:

6	Level number, a zero-based level number 0, 1 ... K-1.
9	Number of input sequences at this level.
70.0	Percent identity for this level.

This is followed by one line per sequence. Each line has three fields. Here is a complete example of a level.

```
LEVEL     6    9   70.0
6    0      *
6    61     *
6    565    0
6    726    *
6    1542   *
6    4408   61
6    4858   0
6    4879   *
6    9366   *
```

In the lines following the LEVEL record, the first field is the level number. As for sequence records, this field is redundant but should be used by parsers to verify consistency with the file. The second field is the sequence ID, referring back to the sequence records at the beginning of the file. Sequence IDs are

the same for all levels. A given level will have only the subset of IDs that correspond to seeds discovered in the previous level. The third field is either a second sequence ID, indicating a match, or an asterisk '*', indicating no match. A match means that the sequence was assigned to a cluster, an asterisk means that this sequence becomes a new seed at this level. So the above example has six seeds that would be passed down to the next level and three matches, two to seed ID=0 and one to seed ID=61. If there is a 7th level, it will have six input sequences which are the seeds identified at level 6.

Creating large, high-quality multiple alignments

MUSCLE can create alignments of up to perhaps 10,000 to 20,000 sequences. Larger sets can be aligned using a divide and conquer strategy based on clumping. This may be advantageous even in cases where MUSCLE can align the complete set as the resulting alignments tend to be more compact, having fewer columns and thus fewer gaps, which may be preferred for some types of analysis.

In outline, the strategy is as follows.

1. Create clumps, i.e. clusters that are small enough for MUSCLE to align.
2. Create a 'master' set containing the longest sequence from each clump.
3. Align each clump.
4. Align the master set.
5. Merge the clumps into a final alignment, using the master alignment as a guide.

The first step is to create clumps. Anecdotally, I have found that a clump size of around 5000 gives good results, but this may vary depending on your data. I recommend experimenting with different clump sizes and examining the results. Typical commands would be:

```
mkdir myclumps
uclust --uhire seqs.sorted.fasta --clumpfasta myclumps/ --maxclump 5000
```

The clumps and the master set are then aligned using MUSCLE. For example, this

```
mkdir alns
cd alns
for filename in clump.* master
do
   muscle -in $filename -out ../alns/$filename -maxiters 2
done
cd ..
```

I recommend the -maxiters 2 option to MUSCLE as a good compromise between speed and accuracy for larger sets. Any multiple alignment method can be used in place of MUSCLE if desired.

The alignments are combined using the --mergeclumps command, as follows.

```
uclust --mergeclumps alns/ --output aligned.fasta
```

Sequences in the master file are required to have their labels formatted to indicate the clump number. This is done automatically if the --clumpfasta option is used; if you use some other method to select the master set then you must take care to follow the label formatting convention. The clump ID (0, 1... N-1) is indicated by a prefix like >M123| where 123 is the clump ID. For example, this is a valid FASTA label for the master sequence of clump 28:

```
>M28|GF2FOAC01AU7TA
```

Clump 8 must contain an identical sequence with label >GF2FOAC01AU7TA, this correspondence is used to merge the alignments of each clump into a single multiple alignment.

The UCHIME algorithm: chimeric sequence detection

UCHIME searches for chimeric sequences. Like UCLUST, UCHIME leverages the high speed of the underlying USEARCH algorithm. The basic command-line is:

```
uclust --uchime myseqs.fasta --report myseqs.rep --reportx myseqs.repx
  [--lib ref.fasta] [--minh 0.6]
```

The --lib option specifies a trusted reference database. If not specified, de novo detection is performed. In de novo mode, chimeric triples are reported and additional evidence is required to determine which of the three is the true chimera. A good heuristic is to designate the least abundant sequence as the chimera. Input and reference sequences should be in unaligned FASTA format (no gaps). The --minh option specifies the minimum score to be considered a hit. Default is --minh 0.6, which gives an error rate < 1% on full-length simulated sequences.

SNPs and voting

UCHIME constructs a three-way alignment of two putative step-parents (A and B) to the query sequence (Q). A is always the first step-parent that models the left side of the query, B is the second (right side). UCHIME counts columns in the three-way alignment containing differences (SNPs). SNPs in which two sequences agree and the third differs either support or contradict the model. SNPs in which all three sequences differ do not provide direct evidence but could reduce confidence by indicating noise (sequence errors, inaccurate alignment or divergence between the true parent and the step-parent).

Summary report format (--report option)

In the summary report produced by the --report <filename> option, the top hit (T) is also reported. The top hit is the most similar sequence to the query, and acts as a control: the closer the top hit is to the model, the lower the confidence in the prediction. Often, but not always, the top hit is one of the step-parents in the model. In the Labels column, there are either three or four labels: Q(A,B) or Q(A,B,T). The top hit is indicated by a + prefix on the label. Here is some sample output.

>	H	Div	L.SNPs y/n/?	R.SNPs y/n/?	IdP	IdT	IdAB	IdM	Labels
Y	1.172	1.010	A14/b0/N0/?0	B187/a0/N0/?	99.0%	99.1%	86.2%	100.0%	7000004131498723 (S000129426,+7000004131498722)
Y	1.100	1.023	A35/b0/N2/?0	B32/a0/N1/?0	97.6%	99.0%	95.4%	99.8%	7000004131499276 (7000004131499535,S000004313,+7000004131499279)
Y	1.058	1.009	A17/b0/N0/?0	B13/a0/N0/?0	99.2%	99.2%	98.0%	100.0%	7000004131500055 (+7000004131500053,7000004131500110)
Y	0.966	1.008	A136/b0/N0/?	B12/a0/N0/?0	99.2%	99.4%	90.5%	100.0%	7000004131313504 (7000004131313502,S000546730,+7000004131319294)
Y	0.933	1.008	A55/b0/N0/?0	B11/a0/N0/?0	99.2%	99.2%	95.2%	100.0%	S000439503 (+S000439506,S000364348)
Y	0.791	1.007	A76/b0/N1/?0	B10/a0/N0/?0	99.3%	99.3%	94.3%	99.9%	7000004130820864 (+7000004130820865,S000436035)
Y	0.790	1.007	A54/b1/N0/?0	B10/a0/N0/?0	99.3%	99.3%	95.7%	99.9%	7000004130820865 (+7000004130820864,S000390059)
Y	0.748	1.007	A10/b0/N0/?0	B191/a2/N0/?	99.0%	98.4%	85.9%	99.7%	S000474146 (S000541573,+S000539440)
Y	0.736	1.006	A52/b0/N0/?0	B9/a0/N0/?0	99.4%	99.4%	95.8%	100.0%	7000004128190036 (S000391339,S000388296,+S000391451)
Y	0.691	1.019	A30/b0/N1/?0	B28/a2/N0/?0	97.9%	97.8%	95.7%	99.8%	7000004131499453 (S000470254,7000004131499640,+7000004131499276)
Y	0.651	1.006	A21/b0/N0/?0	B8/a0/N0/?0	99.5%	98.4%	98.0%	100.0%	S000008957 (S000544247,+S000005612)
Y	0.641	1.005	A23/b0/N0/?0	B8/a0/N0/?0	99.5%	99.2%	98.0%	100.0%	7000004131499449 (+7000004131499640,7000004129457926)
Y	0.628	1.005	A8/b0/N0/?0	B75/a0/N1/?0	99.4%	99.4%	94.5%	99.9%	7000004131503051 (7000004129457926,7000004131503121,+7000004131503117)
Y	0.604	1.006	A8/b0/N0/?0	B10/a0/N0/?0	99.2%	98.8%	98.5%	99.7%	S000544231 (S000142900,+S000003569)

Detailed report format (--reportx option)

The detailed report shows three-way alignments and summary statistics for each model. Here is an example alignment.

```
Query   (  209 nt) FW1022_2nd_NO3Lac_F3SGFHA01AFHGD
ParentA (  207 nt) FW1023_2nd_NO3Lac_F3SGFHA01DNBN3
ParentB (  208 nt) FW1022_2nd_NO3Lac_F3SGFHA01CZS9S

A     1 tgcgcaggcggttatataagacagatgtgaaat-ccccgggctcaacctgggaactgcattagtgactgtatagctagag 79
Q     1 tgcgcaggcggttatataagacagatgtgaaatcccccgggctcaacctgggaactgcattagtgactgtatagctagag 80
B     1 tgcgcaggcggttGtGtaagacagGCgtgaaat-ccccgggctcaacctgggaactgcGCtTgtgactgCaCagctagag 79
SNPs                A A        AA                                  AA A       A A
Model   .............AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

A    80 tgcggcagaggggatggaattccGcgtgtagcagtgaaatgcgtGgaTatgCggaggaacaccgatggcgaaggca-AT 158
Q    81 tgcggcagaggggatggaattccacgtgtagcagtgaaatgcgtagagatgtggaggaacaccgatggcgaaggcagcc 160
B    80 tAcggcagaggggGtggaattccacgtgtagcagtgaaatgcgtagagatgtggaggaacaccgatggcgaaggcagcc 159
SNPs     A           A        B                   B  B   B                          BB
Model   AAAAAAAAAAAAAAAxxxxxxxxxxBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

A   159 cccctgggccTGCactgacgctcatgcacgaaagcgtgggGagcaaaca 207
Q   161 cccctgggccgatactgacgctcatgcacgaaagcgtgggtagcaaaca 209
B   160 cccctgggccgatactgacgctcatgcacgaaagcgtgggtagcaaaca 208
SNPs             BBB                           B
Model   BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB........

Score   2.838
Ids.    QA 95.2%, QB 94.7%, AB 89.9%, QModel 100.0%, Div. +5.1%
SNPs    Left A=11(100%)/b=0/N=0/?=0, Right B=10(100%)/a=0/N=0/?=0
```

SNP classification

Letters that agree are shown in lower-case, disagreements are shown in upper-case (in order to stand out -- I think upper case stands out better against lower-case than vice versa). SNPs are classified as follows. Here, a column is written as AQB, and upper case indicates disagreement. These are the possible types of column.

aqb	Q=A=B	Not a SNP.
aqB	Q=A, Q!=B	Votes for model if on the left (symbol 'A') or against model if on right (symbol 'a').
Aqb	Q=B, Q!=A	Votes for model if on the right (symbol 'B') or against model if on left (symbol 'b').
aQb	A=B, Q!=A, Q!=B	Indicates SNP in query vs. (step-)parents. Could be a read error, or a change in one of the true parents vs. the closest step-parent. Symbol 'N'.
AQB	All different.	Not directly informative, indicates noise. Symbol '?'.

Pros and cons of UCHIME

Please note that UCHIME is a work in progress. I am hoping to improve the algorithm in the near future. This is a brief discussion of the current version.

Advantages of UCHIME

Compared with previous methods, UCHIME is very fast: it can process hundreds or thousands of sequences per minute, depending on length. Also, unlike some other methods it does not require a trusted (i.e., chimera-free) reference database or a pre-built multiple alignment of trusted sequences. Preliminary tests suggest that on full-length 16S rRNA sequences, UCHIME has better sensitivity and

lower error rates than any other method. UCHIME provides a score for each predicted chimera which allows the user to set a score threshold. Adjusting the threshold increases sensitivity or reduces the false-positive rate, similar to a BLAST e-value. At reasonable thresholds, the false-positive rate is consistently low.

Disadvantages of UCHIME

Like most other methods, UCHIME currently only searches for "bimeras" (two parent sequences), but "multimeras" can also be important in practice [doi 10.2144/000113219]. Preliminary tests suggest that ChimeraSlayer has significantly better sensitivity than UCHIME in two scenarios: (i) with short reads, and (ii) when parent sequences are not available in the reference database so that related sequences ("step-parents") are needed to construct a model of the query sequence. Here, "short" means less than about 500 nucleotides.

While several avenues for improving detection accuracy remain to be explored, I believe UCHIME can, and therefore should, at least match the performance of ChimeraSlayer on bimeras before moving on. This is because UCHIME currently uses a similar strategy to ChimeraSlayer that constructs a three-way alignment of putative step-parents to a query sequence. If UCHIME is less sensitive than ChimeraSlayer, then one or more phases of UCHIME must be sub-optimal (search for candidate step-parents, alignment construction or model scoring). Optimizing the "three-way / bimera" algorithm will inform attempts to develop a better "multi-way / multimera" algorithm, which is my long-term goal.

If you are working with short reads, then a practical compromise could be to use UCHIME to find candidate chimeras using a threshold designed to maximize sensitivity at the expense of a higher error rate. The candidates could then be passed to the much slower ChimeraSlayer for final discrimination. (Note that the complete CMCS pipeline should be used, not the ChimeraSlayer perl script alone).

Search parameter tuning

UCLUST offers a number of parameters for adjusting speed and sensitivity. The characteristics of datasets found in practice are highly variable, and it is therefore challenging to set universally appropriate defaults or to develop simple guidelines to assist users in setting the best parameter values for particular applications. With these considerations in mind, the following procedure is suggested for tuning parameters. (See also the section below *Evaluating fast alignment performance*).

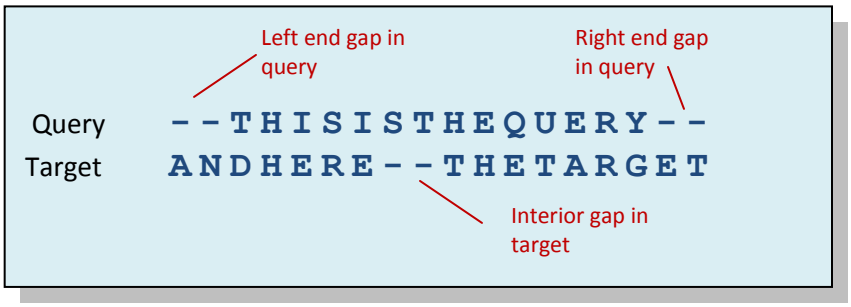
A dataset is first clustered at low identity (say, 50% for proteins or 80% for nucleotides) using default parameters, giving an initial set *S* of clusters. A subset *F* is then extracted from *S* for performance tuning analysis. The redundancy of *F* at higher identities, e.g. 90%, will typically be similar to *S*, meaning that the average cluster size will be similar. If *F* is small enough, the exact or optimal variants can be used as a reference against which faster but potentially less sensitive parameters can be compared. Redundancy is the most important factor in determining elapsed time (which scales roughly linearly in the number of sequences *N* and the number of clusters *M*) and memory use (which scales roughly linearly in *M*), unless *M* becomes very large. Time or memory use on *S* with a given set of parameters can therefore be estimated as $|S|/|F| \times$ (time or memory on *F*). If the exact variant is prohibitively expensive on *F*, an alternative is to use parameters designed for high sensitivity while retaining some speed heuristics, e.g.

```
--maxaccepts 1 --maxrejects 128 --step 0 --bump 0.
```

See also the --check_fast option, described next.

Gap penalties

UCLUST supports a rich set of gap penalty options. Up to 12 separate penalties can be specified: all combinations of query / target, left / interior /end, and open / extend.



The following table gives the penalties that UCLUST uses by default.

Penalty	Default
Interior gap open	10.0 (nucleotide) 17.0 (amino acid)
Terminal gap open	1.0
Interior gap extend	1
Terminal gap extend	0.5

Terminal gaps are penalized much less than interior gaps, which is typically appropriate when fragments are aligned to full-length sequences. These defaults can be changed using the --gapopen and --gapext options. The nucleotide defaults would be set using these options:

```
--gapopen 10.0I/1.0E --gapext 0.5
```

A numerical value for a penalty is optionally followed by one or more letters that specify particular types of gap. Here, "10.0I" means "Interior gap=10.0", and "1.0E" means "End gap=1.0". If no letters are given after the numerical value, then the penalty applies to all gaps. More than one letter can be specified, so for example "0.5IE" means "Interior and End gaps=0.5", which is the same as all gaps. Following are valid letters: I=Interior, E=End, L=Left, R=Right, Q=Query and T=Target. If more than one numerical value is specified, then they must be separated by a slash character '/'. White space is not allowed. If a star ('*') is used as the numerical value, then the gap is forbidden. Using * in an open penalty means that the gap will never be allowed, using * in an extension penalty means that gaps longer than one will be forbidden. So, for example, *LQ in --gapopen means "left end-gaps in the query are not allowed". A sign (plus or minus) is not allowed in the numerical value, which can be integer or floating-point (in which case a period '.' must be used for the decimal point). The --gapopen and --gapext options are interpreted first by setting the defaults, then by scanning the string left-to-right. Later values override previous values.

The final settings are written to the --log file, and I strongly recommend that you use this information to check that your options are correctly formatted. Here is another set of example options.

```
--gapopen 10.0QL/*QL/2.0TE/1.0QR --gapext 0.5I/0.1E
```

The resulting penalties appear as follows in the log file.

```
    10.00  Open penalty (query, internal)
        *  Open penalty (query, left end)
     1.00  Open penalty (query, right end)
    10.00  Open penalty (target, internal)
     2.00  Open penalty (target, left end)
     2.00  Open penalty (target, right end)
     0.50  Ext. penalty (query, internal)
     0.10  Ext. penalty (query, left end)
     0.10  Ext. penalty (query, right end)
     0.50  Ext. penalty (target, internal)
     0.10  Ext. penalty (target, left end)
     0.10  Ext. penalty (target, right end)
```

Considerations when using non-standard gap penalties

The --gapopen and --gapext options do not always work well with the fast alignment heuristics that are enabled by default. In some cases, especially if some gap types are forbidden, then this can cause UCLUST to crash.

If possible, the best thing to do is to disable the heuristics by using --nofastalign. Then the gap penalties should work well. If you have very large datasets and heuristics are needed, then I recommend testing on small datasets and reviewing the --blastout file to make sure that the alignments look reasonable for your application.

A compromise that often works well is to disable HSPs by using --hsp 0. In typical applications, banding will still give improvements in speed without significantly degrading alignment accuracy or estimates of identity.

Evaluating fast alignment performance

The --check_fast option performs an automated analysis of the --fastalign heuristics. With --check_fast, whenever a pair-wise alignment is constructed, UCLUST compares the alignment with and without fast heuristics (HSPs and banding). Results are written to the --log file. Here is an example.

```
Pair-wise alignment statistics
   1183407  Alignments
   1113873  Hits (94.1%)
    718023  Fast alignments same as slow (hits) (64.5%)
    774833  Fast alignments same as slow (all) (65.5%)
      4302  No HSPs found (0.4%)
       188  Alignments with HSPs not in slow (0.0%)
      3131  Rejected by low HSP id (0.3%), 0 are FPs (0.0%), 1 are FNs (0.0%)
      6319  Heuristic %id > 1% error vs. slow (0.5%)
         0  Heuristic false-positive hits (0.0%)
      4163  Heuristic false-negative hits (0.4%)
 2.65e+012  CPU time for slow alignments
 1.45e+011  CPU time for fast alignments
      18.3  Alignment time speedup by using heuristics
```

Here, 'slow' means without fast heuristics, i.e. using full dynamic programming as with --nofastalign. 1.1M alignments were made, and 94% of these were hits. This shows the effectiveness of the UCLUST index and filtering algorithm: almost all alignments confirmed putative hits. About 2/3 of alignments

were the same with and without the --fastalign heuristics: 64.5% of alignments for hits, and 65.5% overall. So about 1/3 of alignments were sub-optimal in terms of maximizing the objective score. However, the statisitcs show that these sub-optimal alignments make little difference in estimating the query-target identity. There was just one false negative (FN) due to rejection of a target sequence based on the low identity of its HSP(s), and no false positives (FPs). In only 6.3k/1.1M cases (0.5%) was the identity estimated using a fast heuristic alignment more than 1% different from the identity computed from the full dynamic programming alignment. There were no false positive hits and only 4.2k/1.1M false negative hits, i.e. 0.4%. In most cases, this low "error" rate is well worth the 18x speed improvement achieved by using --fastalign. And in general, it is not certain that the full dynamic programming alignments are really better than the heuristic alignments, so it is not clear whether these are true biological "errors".

Definition of identity

UCLUST computes identity from a global alignment as:

> (number of letter-letter columns containing identical letters) / (length of shorter sequence).

It is straightforward to modify UCLUST to support other definitions of identity—please let me know if another measure would be useful for you. Using the --blastout option is useful for visual review of alignments and identities.

Computing all-vs-all pair-wise identities

You can compute all pair-wise identities for a set of input sequences as follows:

```
uclust --input seqs.fasta --lib seqs.fasta --usersort --allhits
  --libonly --maxaccepts 0 --maxrejects 0 --id 0 --uc allpairs.uc
```

Why did (or didn't) UCLUST assign sequence Q to target S?

This is the most common question I get about UCLUST results. The following notes walk you through the process of figuring out why UCLUST didn't do what you expected.

Check UCLUST's idea of the identity of Q and S

The first thing to check is the identity of Q and S according to the UCLUST alignment. If this is above (below) the threshold, this would explain why it was (was not) assigned to the target. First make a file qs.fa containing just Q and S, then:

```
uclust --input qs.fa --usersort --id 0.0 --blastout qs.blast --uc qs.uc
```

You can check the identity in the .uc file, and see the alignment and identity calculation in the .blast file.

The alignment looks bad

If the .blast alignment looks bad, this may be caused by the heuristics (HSPs and banding) used by UCLUST to make fast alignments. You can check this manually by repeating with --nofastalign:

```
uclust --input qs.fa --usersort --id 0.0 --blastout qs.blast --uc qs.uc --nofastalign
```

UCLUST can do this automatically by using --check_fast, in which case you should use the same --id that you used for clustering, say:

```
uclust --input qs.fa --usersort --id 0.90 --check_fast --log qs.log
```

A report will be written to the log file. See the section "Evaluating fast alignment heuristics" for more information.

You can reduce the number of bad alignments by using --nofastalign, which does full global dynamic programming alignments with no heuristics. This will be slower, but will produce alignments of comparable quality pair-wise alignments by tools like MUSCLE and CLUSTALW.

If --nofastalign is unacceptably slow, you can compromise by tweaking the heuristics. E.g., you can increase the band radius by using the --band option.

The identity is above the threshold, but Q wasn't assigned to S

There are two cases: Q was assigned to a different target (next section below), or no match was found.

If no match to S was found, there are three possible explanations:

1. S was rejected because the word count was too low.
2. S was rejected because the HSP identity was too low.
3. S was not tested because the search was abandoned before reaching S.

Cases 1 and 2 can be checked by clustering a file containing just S and Q. Make sure S comes first so that it becomes the seed. Then:

```
uclust --input qs.fa --usersort --id ...
```

Use the same --id and other search options that you used in your original run. If Q is assigned to S, this means that in your original run, the search must have been abandoned before reaching S. If Q is not assigned to S, but has a high enough identity in your first test, then S must have been rejected because the word count or HSP identity was too low. You can check whether it was the word count by setting the --nowordcountreject option. If S was rejected because of a low word count, now Q should be assigned to S.

You can check rejections by repeating your original run with --output_rejects. If repeating your original run is time-consuming, a later section below explains how to reproduce what happened to Q more quickly. With output_rejects, Reject records (R in column 1) are written to the .uc file. If there is a reject record for S, this is due to case 1 or 2 above, or because the identity calculated from an alignment is too low (as discussed earlier). If there is no reject record for S, then S was never tested and the search was abandoned too early. This is expected to happen in rare cases because target sequences are tested in an order that correlates approximately with identity, but not exactly. It can therefore happen that UCLUST gives up the search when in fact there is match lower in the list. The frequency of this type of false negative can be reduced by increasing the --maxrejects option.

Q was assigned to T which has lower identity than S

This also happens occasionally because of the order in which UCLUST checks target sequences. Since the order does not exactly correlate with identity, there may be a better match lower in the list, but is not found because the first match found is accepted by default. The frequency of these sub-optimal hits can be reduced by increasing --maxaccepts, which defaults to 1. Generally you should increase --maxrejects

also, otherwise UCLUST will give up after only 8 rejections, which limits the total number of sequences tested and prevents UCLUST from getting deep into the list of potential hits.

Quickly reproducing the state when Q was processed

If you are using --libonly, you can make an input file containing just Q. The database in memory doesn't change, so Q should be processed in exactly the same way as in your full run. It should then be very fast to try the following techniques with the entire database instead of just one or two targets.

Use --blastout to review the alignment and identity calculation.

Use --nofastalign or --check_fast to compare alignments with and without heuristics.

Use --output_rejects to review rejections.

If you are not using --libonly, then the database grows over time and you need to know exactly which sequences were in the database when Q was processed. The can be determined from the .uc file. Each new seed is indicated by an S record. So extract all S records in the .uc file that appear before Q, and call this file seeds.uc. Convert seeds.uc to a FASTA library:

```
uclust --uc2fasta seeds.uc --input seqs.fa --output seeds.fa --types S
```

If you used --lib, add that in too:

```
cat lib.fa >> seeds.fa
```

Now you can process Q in exactly the same way as your original run without waiting for other sequences to be processed first. Put Q into a file q.fa, then:

```
uclust --input q.fa --lib seeds.fa ...other options as in original run...
```

Memory requirements

The amount of memory needed is approximately 10x the size of a FASTA file containing the database (seeds). A better estimate is

(9x the number of letters in all sequences) + (1x the number of letters in all labels)

The amount of memory required can be reduced in a number of ways, as follows.

Reduce redundancy

If you have very similar sequences in your database, say at least 98% identical, then it could pay to reduce redundancy by clustering at a high identity, say 98% or 99%. This, of course, can be done using uclust itself to pre-process your database.

Trim sequence labels

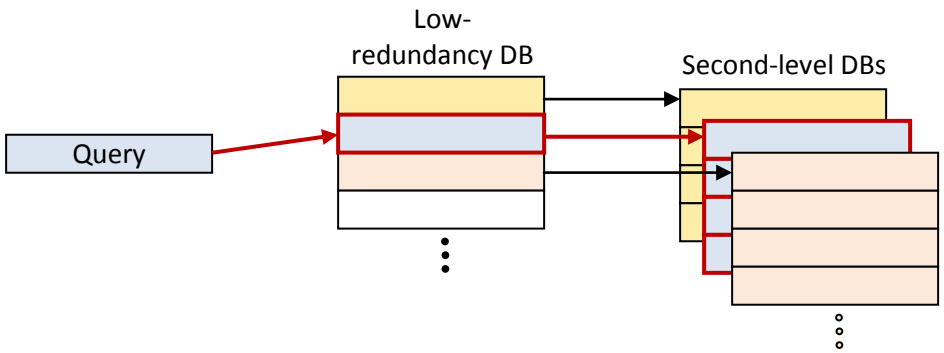
Sequence labels, i.e. the characters following '>' in a FASTA file, are stored as-is in memory. If your labels are long and your sequences are short, then the amount of memory required for labels may be a significant fraction of the total memory requirement. If so, it pays to reduce the label size. For example, you could label your sequences with an integer or some other short string that can be used as a key for retrieving longer annotations in a post-processing step.

Split the database

You can split the database into smaller pieces. This allows you to parallelize a search (e.g. by running the query against N pieces in parallel on N machine in a cluster, or to serialize (by running one piece after the other on a single machine). Splitting the database may also have the advantageous side-effect of improving sensitivity. The very high speed of the USEARCH algorithm is achieved by limiting explicit sequence comparisons to a small subset of the database having the most unique words in common with the query sequence. As the database size grows, more spurious sequences will tend to appear in this subset and sensitivity may be reduced as a result.

Two-level search

If finding the closest possible match to a very large database is important in your application, then you can combine the "reduce redundancy" and "split" strategies to achieve improved speed, reduced memory use and (usually, but not always) higher sensitivity. The idea is to search first in a low-redundancy database (LRD). Sequences in the LRD are annotated with the name of a second-level database (SLD) which has more closely-related sequences. There are several SLDs that, when combined, contain the full set of sequences. In the second pass, the query is searched against the SLD identified in the first search.



This picture is over-simplified: we don't want a separate SLD for every sequence in the low-redundancy DB. There are two reasons for this: if the SLD is too small, we lose the advantage of the high search speed of USEARCH because there will be too much overhead setting up each query. Also, we want to group related families into a single SLD because otherwise the hit to the LRD may not correctly identify the SLD with the closest possible match.

To create the databases, I suggest the following approach.

1. Cluster at a fairly low identity; say 50% for proteins or 80% for nucleotides.

2. Pick a desired size for an SLD, say 1/N of the full database. If a cluster from step 1 is larger than this, you can split it by clustering at a higher identity, or go back and re-cluster the entire database at a higher identity.

3. Merge clusters from step 1 to create the SLDs (SLD1, SLD2 ... SLDN). This can be done by a simple greedy algorithm which can be implemented in a script, let me know if you'd like help with this. Label each sequence with the name of its SLD (this is so that the SLD name is available in step 5 below where the LRD is created).

4. Cluster each SLD at a high identity; say 98% for nucleotides or 90% for proteins.

5. Combine all the seeds from step 4 above, this produces the LRD.

To run a two-pass query, first search the query sequences against the LRD. Then divide them according to the SLD identified by the LRD hit and run each subset against its SLD; this of course can be done serially or in parallel.

Command-line options

Input options

Option	Description
--input filename	Input file containing query sequences. FASTA format. By default, must be sorted by decreasing sequence length (see --usersort).
--query filename	Same as --input, except implies --libonly.
--db filename	Same as --lib if used with --query.
--lib filename	Library file. FASTA format. This is used to initialize the search database in memory. By default, the initial database is empty.
--usersort	By default, if new seeds may be created, then UCLUST requires that the input file is sorted by decreasing sequence length, and will fail if it is not. Specify --usersort to indicate that file is sorted by some other criteria (or is not sorted at all but you think this is OK). For *de novo* clustering, input should be sorted so that a suitable seed sequence will appear before other members of the cluster. If the input includes both full-length sequences and fragments, then sorting by decreasing length is usually the best approach.
--maxlen L	Ignore query sequences that are longer than L. Default 10000. UCLUST is currently not designed to handle very long sequences. If you increase this value significantly, then UCLUST may fail due to lack of memory or for other reasons. Let me know if you have applications that need longer sequence lengths.
--minlen L	Ignore query sequences that are shorter than L. Default 16.
--amino	Specifies that input sequences use the 20-letter amino acid alphabet. By default, UCLUST 'guesses' this from the frequencies of AGCTU in the first few sequences.
--nucleo	Specifies that the input sequence use a nucleotide alphabet. By default, UCLUST 'guesses' this from the frequencies of AGCTU in the first few sequences.

Output options

Option	Description
--trunclabels	Truncate FASTA labels at the first whitespace character.
--allhits	Write all hits to the .uc file. By default, only the best hit found is written. To get more than one hit, you must specify --allhits and set --maxaccepts

Option	Description
	to a value > 1.
--[no]output_rejects	Write reject records to the .uc file. By default, rejects are not written (--nooutputrejects). This is mainly useful for trouble-shooting unexpected results.
--log filename	Write a log file with miscellaneous information. I recommend that you try this and take a look at the output, you might find some of it helpful.
--blastout filename	Output file for human-readable alignments in a BLAST-like format. This format may be changed in future versions, so it is recommended that parsers use the FASTA output generated by --fastapairs instead.
--fastapairs filename	Output file for pair-wise alignments in FASTA format. Pairs are separated by blank lines, to make the file easier to inspect visually. The query sequence is first, the target (seed, database) sequence is second. If the input sequences are nucleotides, then a + or - is appended to the label of the target sequence to indicate the strand. If the strand is - (reverse strand match), then the target sequence is reverse-complemented.
--rowlen n	Row length for --blastout file. Default 64.
--idchar c	Character annotating identities in --blastout file. Default '|'.
--diffchar c	Character annotating differences in --blastout file. Default blank .
--[no]blast_termgaps	[Don't] output terminal gaps to --blastout file. Default --noblast_termgaps.

Search options

Option	Description
--id f	Minimum identity to accept a hit. Floating point number in range 0.0 to 1.0. Default 0.9.
--maxaccepts n	Keep searching until n hits have been found, then report the best. Default 1. Zero means infinity, i.e. don't stop however many matches have been found (but will still stop if the maximum number of rejects has occurred). Use --maxaccepts 0 --maxrejects 0 to force a search of the entire database with every query, this guarantees that the best hit will be found, if one exists.
--maxrejects n	Keep searching until n rejects have occurred, then report a failure to find a hit. Default 8. Zero means infinity, i.e. keep searching until all a hit is found or database sequences have been tested.

Option	Description
--w n	Word length for unique word index. Default is 8 for nucleotides, 5 for amino acids. It is not clear whether these defaults are good in all applications; further research is needed to understand this better.
--[no]wordcountreject	By default, --wordcountreject is enabled so that target sequences are rejected if they have too few unique words in common with the query sequence. The threshold is estimated using heuristics. This improves speed, but may also reduce sensitivity. Using --nowordcountreject disables word count rejection.
--bump n	By default, an optimization called "threshold bumping" is used to reduce the search space when many target sequences are found to pass the word count threshold. This may reduce sensitivity slightly, and may increase the probability that the top hit is not found, but often improves speed significantly when the database is large. Default is --bump 50. Use --bump 0 to disable bumping.
--stepwords n	By default, an optimization called "stepping" is used to speed up database searching. This is effective when the number of words in common between the query and target is expected to be large. Then it is expensive to check all words, and stepping selects a subset of words in the query. By default, --stepwords is 8. This means that the number of query words is chosen so that approximately 8 words are expected to be found in the target sequence. Use --stepwords 0 to disable stepping. As with bumping, stepping may reduce sensitivity and may reduce the probability that the best hit is found first.
--rev	By default, UCLUST searches only the plus strand for nucleotide sequences. If --rev is specified, then UCLUST will also search the reverse-complemented sequence.
--libonly	By default, if no hit is found, UCLUST will add the query sequence as a new seed. If --libonly is specified, this does not happen. Using --lib and --libonly is appropriate for database search applications.
--self	Used for searching a database against itself, e.g. to reduce redundancy. If the target and query labels are identical, the target is ignored. Typical use is uclust --input lib.fa --lib lib.fa --libonly --self --uc results.uc.
--idprefix n	Require that the first n letters of query and target are identical. Default zero.
--exact	Same as --maxrejects 0 --nowordfilter. Guarantees that a hit will be found if one exists.
--optimal	Same as --maxrejects 0 --maxaccepts 0 --nowordfilter. Similar to --exact

Option	Description
	but also guarantees that the best hit will be found.

Alignment options

Option	Description
--match s	Match score for nucleotides. Default 2.0.
--mismatch s	Mismatch score for nucleotides. Default -1.0.
--gapopen s	Gap open penalty specification. Format is described elsewhere in this manual.
--gapext s	Gap extension penalty specification. Format is described elsewhere in this manual.
--[no]fastalign	Default is --fastalign. Specify --nofastalign to disable fast alignment heuristics (HSPs and banding).
--hsp	Minimum length for an HSP. Default 32. Specify zero to disable HSPs.
--hspscore s	Minimum score/column for an HSP. Default 1.0.
--band n	Radius of band for banded dynamic programming between HSPs. Default 16. Specify zero to disable banding.
--check_fast	Compare results using alignments with and without fast heuristics and generate a report in the --log file.

Miscellaneous options

Option	Description
--quiet	Don't write progress messages to standard error.
--version	Write version to standard output and exit.
--help	Write command-line help to standard output and exit.