Software and documentation

http://drive5.com/usearch

robert@drive5.com

Version 4.2
June 1, 2011

Table of Contents

USEARCH

USEARCH is a program that implements several algorithms for fast sequence database search, clustering and related tasks.

License

USEARCH is copyrighted software and generally requires a paid license. Licenses to use the 32-bit binaries are available at no charge for approved non-commercial use.

Installation

USEARCH 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.

Executable filename

The file name of the binary file includes the platform and version number, e.g. usearch4.2.50_i86linux32 is the 32-bit binary for version 4.2.50 on Intel i86 architecture Linux. In this documentation, 'usearch' (lower-case) will be used for the command name. Where you see 'usearch', you should replace this with the appropriate command name for your system. For example, this is a generic command:

```
usearch --cluster input.fasta --uc results.uc --id 0.97
```

On your system, you would use something like this:

```
usearch4.2.50_i86linux32 --cluster input.fasta --uc results.uc --id 0.97
```

Or, if you like, you can change the name of your executable file to usearch, which is easier to remember and type.

USEARCH overview

The USEARCH program implements several algorithms with a rich set of options. This table gives a brief summary of its most popular capabilities.

Category	Common uses
Database search (UBLAST).	Capabilities similar to BLASTP, BLASTN and BLASTX. In addition. UBLAST offers fast search options that can achieve speed improvements over BLAST of 100 to 1000× with good sensitivity to distant relationships (down to around 75% id for nucleotides or 40% id for proteins). Used in a wide variety of sequence classification tasks. Supports both local and global alignments for search (BLAST is strictly local).
Clustering (UCLUST).	Create non-redundant ("NR") and reduced-redundancy ("RR") databases. Dereplication: removing identical (sub-)sequences. Denoising: removing sequencer error. Identify Operational Taxonomic Units (OTUs) from single-region environmental sequencing reads (e.g. 16S or ITS).
Database search with clustering of unmatched sequences (UCLUST).	Extend a previously clustered database by adding new sequences (saves compute resources compared with clustering again from scratch.)
Chimera detection (UCHIME).	Search for chimeric sequences *de novo* or using a trusted reference database that is believed to be chimera-free. Please see to the UCHIME documentation for details.

Basic usage: clustering with UCLUST

This section describes the most popular options for the UCLUST clustering algorithm. Many advanced options are also provided; these are described in later sections.

Seed sequences

UCLUST creates clusters defined by *seed* sequences. Each cluster has exactly one seed, which is a sequence from the input set. The user specifies an identity threshold using the --id command-line parameter. For example with --id 0.97, other members of the cluster must have identity at least 97% identity with the seed. Another term for a seed is a *representative sequence*.

Sorting the input sequences

Input sequences should be sorted in an appropriate order as a pre-processing step before clustering. UCLUST processes sequences in the order they appear in the input file. If a sequence matches an existing seed, then it is assigned to that cluster; otherwise it becomes the seed of a new cluster. This means that sequences should be sorted such that the most appropriate seed for a cluster tends to appear first. Suggested sort orders for two common applications follow.

Non-redundant database: sort by decreasing length

UCLUST can create a non-redundant or reduced-redundancy database from an input set containing families of similar sequences. In this case, the generally recommended sort order is by decreasing length. Sorting by length can be done using the following usearch command.

```
usearch --sort seqs.fasta --output seqs.sorted.fasta --log usearch.log
```

Sorting by length is effective when some sequences are exact or approximate fragments of other sequences. A full-length sequence is usually a better choice of seed than a fragment. If the database is too large to fit in memory, the --mergesort command can be used.

OTU identification

UCLUST is often used to identify Operational Taxonomic Units (OTUs) from environmental sequencing of a region such as the bacterial 16S ribosomal RNA gene or fungal ITS. While it is common practice to use a single clustering step for this task, typically with a 97% identity threshold, I do not recommend doing this. See the later section OTU clustering.

The --cluster command

Clustering is performed using --cluster. This is a typical command line.

```
usearch --cluster input.sorted.fasta --id 0.97 --seedsout seeds.fasta
  --uc results.uc --log usearch.log
```

The --id option specifies the identity threshold. In this example, sequences in a given cluster must have ≥ 97% identity with its seed sequence. Note that the --id parameter is specified as a fractional identity in the range 0.0 to 1.0, not a percentage.

Output files

The most popular output file options for clustering are --seedsout, which produces a FASTA file containing the seed sequences (i.e., a non-redundant or reduced-redundancy subset of the input), and --uc, which reports results in a custom file format designed to be easily parsed by a script or manipulated by standard Linux commands such as cut, grep and sort. The --uc file format is described in detail later. Other output file options include --userout, --blastout, --blast6out, --fastapairs and --consout (see next). For more details, see Output Files.

Consensus sequences

The --consout *fastafile* option produces consensus sequences for each cluster in the FASTA file. The consensus sequence is generated by creating a multiple alignment of the cluster and taking the majority symbol (letter or gap) from each column. In the majority symbol is a gap, the column in skipped. Terminal gaps are ignored unless the --cons_termgaps option is specified. In some applications, the consensus sequence is a better model of the cluster than the seed sequence. For example, if you are clustering next-generation reads, then the longest sequence in the cluster tends to have more errors, and taking the consensus tends to correct the errors.

The --usersort option

By default, the sort order is assumed to be by decreasing length, and a fatal error occurs if sequences are found to be out of order. The --usersort option specifies that another sort order was used, so is required for abundance sorting and other orders.

Basic usage: database search with UBLAST

This section describes the most popular options for the UBLAST database search algorithm. Many advanced options are also provided; these are described in later sections.

Database format

UBLAST uses FASTA files for both the query and database sequences. You don't have to use a program like formatdb or makeblastdb as for BLAST.

Search command

Here is an example of a typical database search command-line.

```
usearch --query query.fasta --db db.fasta --blastout results.blast
  --blast6out results.b6 --evalue 0.01 --log usearch.log
```

The query and database FASTA files are specified by the --query and --db parameters. The --evalue option specifies the maximum E-value for a hit.

U-sorting

UBLAST is designed to quickly find one strong hit, which will often be the best hit in the database. This differs from traditional search algorithms like BLAST, which are designed to find all possible hits in the database. UBLAST tests database sequences in order of decreasing number of short words in common with a given query sequence (*U-sorted order*). Similar sequences tend to have more words in common, so the first hit found is often the strongest hit in the database, or one of the best. By default, UBLAST terminates a search when the first hit is found, which can dramatically improve search speeds. This strategy can be disabled by using the --nousort option. If --nousort is specified, USEARCH tests all database sequences using an algorithm very similar to BLASTP or BLASTN. Depending on the database size and other options specified, this may be much slower than U-sorting.

Number of targets tested

UBLAST will abandon a search if too many database sequences (*targets*) have been tested without finding a hit. The maximum number of targets to be tested can be set using the --maxaccepts and --maxrejects parameters. Increasing these values so that more targets are tested tends to improve sensitivity, but reduces speed. If the database is very large, and / or if low-identity hits are desired, then sensitivity can be significantly improved by increasing these parameters. If you have, say, millions of proteins and are looking for hits that may have <50% identity, then you may get better results with --maxrejects values of 100 or 1000, or even larger. The --maxrejects option is ignored if U-sorting is disabled, i.e. if --nousort is specified.

Nucleotide, protein and translated searches

USEARCH can perform nucleotide searches (like BLASTN), protein searches (like BLASTP) and translated searches (like BLASTX). USEARCH automatically detects the alphabet of the query and database sequences. The type of search is determined per the following table.

Query sequences	Database sequences	Search
Nucleotide	Nucleotide	Nucleotide (like BLASTN)
Amino acid	Amino acid	Protein (like BLASTP)
Nucleotide	Amino acid	Translated ORFs (like BLASTX[1])
Amino acid	Nucleotide	*Not supported in this version.*

[1]USEARCH uses ORFs as queries, BLASTX does not.

Output files

Output file options include --blastout (human-readable, BLAST-like), --blast6out (tabbed format compatible with the -m8 or -outfmt 6 option to NCBI BLAST), --fastapairs (pair-wise alignments in FASTA format), --uc (tabbed format primarily designed for clustering results), --consout (consensus sequences for clusters) and --userout (tabbed format with fields specified by --userfields). For more details, see Output Files.

Search strategy

A central step in most USEARCH algorithms is to search a database, which is stored and indexed in memory. For more details, see Searching. The following tables summarize the most common ways in which a database can be constructed and searched. These variations are explained in more detail later.

Database

Option	Description
Search	The database is read from a FASTA file and does not change. This is done in a typical search, which is specified if the --query and --db options are both given.
Clustering	The database is initially empty. Each new seed sequence is added to the database, so the database contains one sequence per cluster. This strategy is used if the --cluster option is given but not --db.
Search + clustering	The database, which contains cluster seed sequences, is initialized from a FASTA file, and then grows as new seeds are identified. This strategy is used if the --cluster and --db options are both given.

Search order

Option	Description
U-sorted	Database is searched in order of decreasing number of words in common. This is the default.
S-sorted	Modified U-sorted order. Designed to increase sensitivity by improving the correlation of the search order with evolutionary distance. Specified by the --[no]ssort option. Amino acid databases only. S-sorting is the default if the --query and --evalue options are both given, otherwise it is disabled by default.
Entire database	All database sequences are tested. Specified by the --nousort option. In this case, the --[no]ssort option is ignored.

Search termination (U-sorted or S-sorted only)

Option	Description
Max accepts	Search stops after this number of accepts (matching targets) found. Set by --maxaccepts option, default value is 1. Increasing this value tends to find a better hit; if you do this you should generally also increase --maxrejects otherwise rejections will often terminate the search before more accepts are found.
Max rejects	Search stops after this number of failed attempts to match a target sequence. Set by --maxrejects option, default value is 32. Increasing this value tends to increase sensitivity and increase execution time.

Alignment

Feature	Variants	Description
Heuristic/optimal	Heuristic	By default, heuristics similar to those used in the BLAST program are used to speed up calculation of an alignment.
	Optimal	Heuristics can be turned off using --nofastalign, in which case the Needleman-Wunsch or Smith-Waterman algorithms are used, i.e. full O(L^2) dynamic programming. This is often much slower, and gives results similar to programs like SSEARCH and NEEDLE. Optimal alignments can be useful for benchmarking, e.g. to evaluate the effect of using heuristics, and for *ad hoc* searches of small databases where the best possible accuracy is desired; for such applications USEARCH may be convenient due to its ease of use, flexible options and rich set of output file formats.
Global/local	Global	All letters of both sequences are aligned. This is the default if --cluster is given, otherwise can be specified using the --global option.
	Local	Two segments (one substring from each sequence) are aligned. This is the default if --query is used, otherwise can be specified using the --local option.

Similarity measures

Variants	Description
Identity	Fraction of columns in the alignment that contain identical letters. Minimum identity is specified by the --id option, which ranges from 0.0 to 1.0, meaning 0% to 100% identity. Several different definitions of identity are supported, as specified by the --iddef option. Can be used for both local and global alignments.
E-value	Karlin-Altschul statistics are used to calculate bit scores and E-values. The maximum E-value is specified by the --evalue option. Applies to local alignments only.
Sequence coverage	Coverage is defined as the fraction of letters in one sequence that are aligned to letters in the other. The minimum coverage is specified by the --queryfract and --targetfract and options (range 0.0 to 1.0; default is 0.0, which effectively disables the option). Can be used for local or global alignments. For local alignments, --queryalnfract and --targetalnfract can also be used.

Word counting

Option	Description
Word length	Short words of fixed length (*k*-mers) are used for two purposes: U-sorting and seeding alignments (like in BLAST). The word length for U-sorting is set by the --w option, which defaults to 5 for amino acids and 8 for nucleotides. The seed word length is set by the --k option, which defaults to 3 for amino acids and 5 for nucleotides.
Word count rejection	By default, a target is rejected if it has too few words in common with the query. This improves speed by eliminating the expensive alignment step, but can result in some false negatives. In practice, this applies only to high identity thresholds because at lower identities the required number of words in common is one or zero. Word count rejection is disabled by --nowordcountreject.
Stepping	A subset of words in the query or target sequence may be used rather than all words. This improves speed by reducing the time required to find those words in the database. The subset is obtained by extracting words at intervals > 1 letter (stepping). Specified by the --stepwords and --dbstep options. Default value of --stepwords is 8, which means that the number of words extracted from the query is chosen so that a target sequence with the required identity is expected to have at least 8 of those words in common. The default for --dbstep is 1, meaning that all target words are indexed. Increasing --dbstep saves memory, but can reduce sensitivity at lower identity thresholds.
Bumping	This optimization reduces the time required for word counting and U-sorting in cases where many target sequences exceed the initial word count rejection threshold. Specified by the --bump option, which defaults to 50.

Sorting sequences for clustering

In most cases, sequences should be sorted prior to clustering. The order should be chosen so that an appropriate seed sequence for a cluster tends to appear first, before other members of the same cluster.

Don't sort if you're doing database search

There is no need to sort query sequences for database search, because no new seeds are created. You should sort if you are doing search and clustering in a single step (--cluster and --db options both specified).

Sorting by decreasing length

Sorting by decreasing length is effective when full-length sequences and fragments are present in the input. Fragments are usually not a good choice of seed, as shown by the following example.

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

Here the seed is a fragment. The two hits are both 100% matches to the seed except for terminal gaps and would therefore be assigned to the same cluster. However, the hits are extended with different terminal regions (red) and therefore have only about 50% identity to each other.

The --sort command sorts sequences by decreasing length.

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

The current implementation of --sort loads all sequences into memory for speed. Available memory (real or virtual) must be at least as big as the input file. Larger sets can be sorted using a slower merge sort, as in the following example.

```
usearch --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 memory needed for the sort will be a little more than this, and 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 of --split tend to be slower.

Sorting by decreasing cluster size (abundance)

The --sortsize option sorts sequences according to a *size=* field in the label. Usage is:

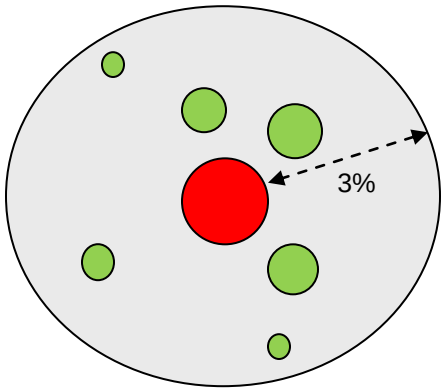
```
usearch --sortsize input.fasta --output input_sorted.fasta [--minsize N]
```

If the --minsize *N* option is given, sequences with a *size=n* label with $n < N$ are discarded. The *size=n* field should be delimited by semi-colons. If it is the last field in the label, the trailing semi-colon is optional.

Following are examples of valid labels with size fields.

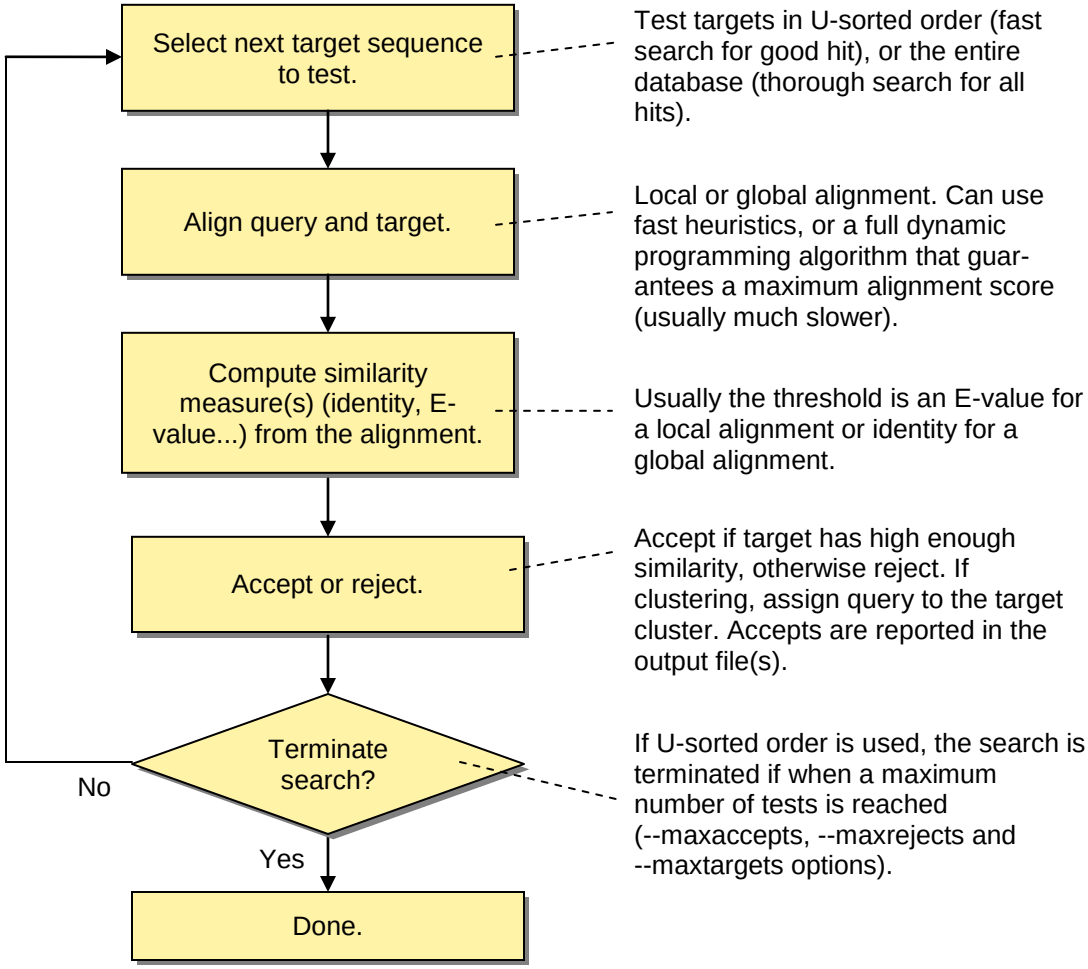
```
>FQ12GCZ34;size=12;qual=0.85
>FQ12GCZ35;size=4
```

Sorting by decreasing abundance is can be a useful strategy as a step in denoising or for identifying OTUs from single-region environmental sequencing reads such as 16S or ITS. 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. See later in this documentation for further discussion of denoising and OTU identification.



Searching

A fundamental step in most USEARCH algorithms is to search a sequence database. For example, UCLUST searches a database of seeds to find a matching cluster for an input sequence. Many different types of search can be specified via command-line options, allowing different trade-offs between speed and sensitivity, local or global alignments, and so on. The typical steps for a given query sequence are shown in the following flowchart.



Search order

Most sequence database search algorithms compare a query sequence with all database sequences (targets). By default, USEARCH algorithms test database sequences in U-sorted order and stop searching as soon as a strong enough hit ("accept") is found. This strategy is effective because U-sort order correlates well with sequence similarity, so the first hit found is often the best hit in the database, or one of the best. U-sorting can be disabled by specifying the --nousort

option, in which case all database sequences are aligned to the query and the target with the highest similarity is selected.

Query-target comparison

A query is tested against (compared with) a target sequence by first creating an alignment, then calculating a measure of similarity from the alignment. Many variants of these steps are supported, e.g. alignments can be global or local, and the measure of similarity can be identity or an E-value. The user can "mix and match" these variations as desired, e.g. the default for clustering is to use global alignment with identity as a similarity measure, but clustering can also be performed using local alignments with an E-value threshold.

Alignment parameters and heuristics

Given a pair of sequences (query and target), there are two sets of options that control the construction of an alignment: *scores* and *heuristics*. Scores include the substitution matrix and gap penalties. Changing scores will tend to change the optimal alignment of a given pair of sequences. Heuristics are approximations that reduce the time required to calculate an alignment. Ideally, changing the heuristic parameters or disabling the heuristics altogether would not change the alignment. However, by definition heuristics do not always produce an alignment with the best possible total score. They are introduced because they improve speed, at the expense of a possible reduction in accuracy. Here, accuracy should be understood in a computer science rather than a biological sense—the optimal alignment of a given pair of sequences may have biological errors despite having the best possible score.

USEARCH with --nofastalign, i.e. with heuristics disabled, is roughly equivalent to programs like SSEARCH and NEEDLE that are based on dynamic programming algorithms without approximations. This can be useful for benchmarking USEARCH, e.g. to evaluate the effect of using heuristics, and for searches with smaller datasets where maximum sensitivity is important. Here, USEARCH may be convenient compared with some other programs due to its ease of use, flexible options and its rich set of output file formats that are designed to be easily reviewed by a human reader or robustly parsed by scripts.

Similarity measure

One or more measures of similarity are computed from a query-target alignment. For clustering, the measure is usually identity computed from a global alignment. For database search, the measure is usually an E-value computed from a local alignment.

Search termination

By default, the database is searched in U-sorted order, and a search is terminated when either (i) a strong enough hit (accept) is found, or (ii) the maximum number of target sequences has been tested. If U-sorting is disabled (--nousort), the entire database is searched, and any termination options given on the command-line are ignored or cause an error.

Accepts and rejects

A target sequence that meets the threshold criteria is called an accept. Typically, this means the target sequence has a high enough identity (clustering with global alignment), or a low enough E-value (database search with local alignment). If the target sequence fails to match, it is called a reject. If the weak similarity parameters --weak_id or --weak_evalue are used, then weak matches are reported in the output files but do not count as accepts and will not cause a query to be assigned to a cluster.

Termination conditions

The following conditions terminate a search. If any condition is satisfied, the search stops. In other words, conditions are combined using a logical "OR".

Maximum accepts

Maximum number of accepts has been found. This is set by the --maxaccepts parameter, which defaults to 1. In other words, by default, a search terminates immediately when the first hit is found. If --maxaccepts is set to zero, there is no limit on the number of accepts (so zero means infinity). Sometimes the first hit found is not the best hit in the database; increasing --maxaccepts increases the probability that the best hit will be found, at the expense of slower execution time. If --maxaccepts is increased, you should probably increase --maxrejects also.

Maximum rejects

Maximum number of rejections has occurred. This is set by the --maxrejects parameter (default 32). If --maxrejects is set to zero, there is no limit on the number of rejections, so the search will continue until an accept is found or the entire database has been searched. Sometimes a hit is not found because the search is terminated too quickly; increasing --maxrejects increases the probability that a hit will be found, at the expense of slower execution time.

Search and clustering at high identities

At high identities, around 96% and above, compressed indexes are often more sensitive, faster and use less RAM. Compressed indexes are disabled by default, so I generally recommend that you specify the --slots and --w options when clustering at high identities.

The default termination parameters are --maxaccepts 1 --maxrejects 32. These are designed for high-identity clustering, which is one of the most common USEARCH applications, and also work well for database search when typical matches have high identity. When identity is high, word count correlates well with similarity, which means that the first accept found is usually the best, or close to it, and the probability of finding an accept falls rapidly with the number of rejects. A U-sorted search therefore quickly reaches a point of diminishing returns if a match is

not found in the first few attempts, so --maxrejects values larger than 32 typically give only small improvements in sensitivity.

Search and clustering at medium identities

When identity is lower, word count correlates less well with similarity, and sensitivity can therefore often be improved by testing more database sequences. Medium identity means, very roughly, 75% for nucleotides or 50% for proteins. Here, it may give better results to increase --maxaccepts over the default value of 1 because at medium identities, the first hit found is less likely to be the best hit and it may therefore be advantageous to test a few more targets. Typical parameters that might work well for medium identity applications are:

```
--maxaccepts 3 --maxrejects 128.
```

Search at low identities

When distant relationships are important, the default parameters will not work well because the number of words in common correlates poorly with similarity below around 50% identity for proteins or 80% for nucleotides. Clustering is rarely useful at such low identities, so this issue applies mainly to searching without clustering. The reduction in sensitivity can be mitigated by increasing the number of target sequences tested, which will be especially important when searching large databases, which tend to produce many spurious candidates (rejections) when tested in a U-sorted or S-sorted order. Some typical parameters are:

```
--maxrejects 1024 --maxaccepts 8 --evalue 1e-6 --weak_evalue 0.01.
```

Output files

The USEARCH database search and clustering algorithms support several output file formats. Most output file formats and features are supported by most of these algorithms.

Option	Format	Description
--uc	UCLUST	Tab-separated file designed primarily for clustering pipelines but can also be useful for search. There is one record for each input sequence giving its cluster assignment, identity and alignment; and one record for each cluster giving its size and average identity.
--blastout	BLAST-like	Verbose, human-readable format similar to BLAST.
--blast6out	Tab-separated	Tabbed format with one record per hit. Compatible with the -m8 or -outfmt 6 option of NCBI BLAST.
--userout	Tab-separated	Tabbed format with one record per hit, fields specified by the --userfields option.
--seedsout	FASTA	Seed sequences, i.e. the non-redundant or reduced redundancy set of sequences after clustering.
--consout	FASTA	Consensus sequence for each cluster. The consensus sequence is computed by taking the majority letter from each column in a multiple alignment of the cluster. If the majority symbol is an internal gap, the column is discarded.
--fastapairs	FASTA	Pair-wise alignments in FASTA format.

The UCLUST file format

UCLUST format (.uc) is a tab-separated text file. UCLUST output is supported by clustering and database search. Each line is either a comment (starts with #) or a record. Each query sequence generates at least one record; additional record types give information about clusters. The cluster number appears in every record type except R (reject). 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 in .uc format are show below.

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, as described in the following table.

Field	Name	Description
1	Type	See table below.
2	Cluster	Cluster number
3	Size	Sequence length or cluster size
4	Id	Identity to the seed (as a percentage), or * if this is a seed.
5	Strand	+ (plus strand), - (minus strand), or . (for amino acids).
6	Qlo	0-based coordinate of alignment start in the query sequence.
7	Tlo	0-based coordinate of alignment start in target (seed) sequence. If minus strand, Tlo is relative to start of reverse-complemented target.
8	Alignment	Compressed representation of query-seed alignment, or * if a seed.
9	Query	FASTA label of query sequence.
10	Target	FASTA label of target (seed / library / database) sequence, or * if a seed.

Record types are as follows.

Type	Description
S	Seed.
H	Hit, also known as an accept; i.e. a successful match.
C	Cluster (seed is a sequence in the --cluster file).
D	Library cluster (seed is a sequence in the --db file).
N	Not matched.
R	Reject (generated only if --output_rejects is specified).
L	Library seed. There is exactly one L record for every --db sequence that has one or more hits.

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 record (L) is output only if a hit is found to that database sequence. This saves writing a large number of records for database 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. Rejection records can be 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.

Class	Name	Query	Target
M	Match	Letter	Letter
D	Delete	Letter	Gap
I	Insert	Gap	Letter

Here, "match" simply means a letter-letter column; the letters may or may not be identical. Deletions and insertions are relative to the query. 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 by cluster number using the standard Linux sort command, as follows:

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

You can also sort by cluster number using USEARCH:

```
usearch --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.

UCLUST to FASTA conversion

You can convert UCLUST to FASTA using the --uc2fasta or --uc2fastax commands.

```
usearch --uc2fasta results.uc --input seqs.fasta --output results.fasta
usearch --uc2fastax results.uc --input seqs.fasta --output results.fasta
```

Here, seqs.fasta must be the same input file used when generating results.uc.

The --uc2fasta command outputs sequences with the same labels and sequences as found in the input file.

The --uc2fastax format reformats 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
```

BLAST6 format

The --blast6out filename option specifies output that is compatible with the NCBI BLAST -m8 or -outfmt 6 options. It is a tab-separated text file with one line per global alignment, or one line per HSP if local alignment is used. Only accepts and weak accepts are written; rejects are not written. By convention I use the .b6 extension for files in this format. There are twelve fields, as shown in the following table.

Field	Description
1	Query label
2	Target label
3	Percent identity
4	Alignment length
5	Number of mismatches
6	Number of gap-opens
7	1-based position of start in query
8	1-based position of end in query
9	1-based position of start in target
10	1-based position of end in target
11	E-value
12	Bit score

BLAST-like alignment 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.:

```
usearch --query seqs.fasta --db genes.fasta --blastout hits.blast
```

Since this format is rather verbose, the file size will be much larger than the corresponding .b6 or .uc file. The details of the formatting are subject to change between versions. It is therefore recommended that parsers use --userout or--blast6out. If full alignments are required, --fastpairs can also be used, though see also the qrow and trow fields for --userout. An example --blastout alignment is shown below.

```
Query    280aa >Q4HFD3_CAMCO
Target   337aa >A6CVA5_9VIBR

  1 LCLGVFGLISMELGVMGIIPLISEKFGVSVSDAGWSVSIFALIVMCCAPIAPMLCANFNPKKLM 64
    | |  | : . |  : ||:| :.    :.   ||: :| :|| :.   ||  .  | || | .:
  1 LTLAAFAIGTAEFIIAGILPQVATSLSITEGQAGYLISAYALAIVIGGPILTIYLARFNKKMVL 64

 65 LFCLAIFSLSSLASMFVNDFWLHLILRAIPAFFHPIYLALAFSTAANLADDKSKVPHIVAKIFM 128
    :  :|:| . .| | |   : :  | | |       :  :    | ||  :|      | ..|
 65 IGLMALFIVGNLMSAFSPSYDILFISRIISGLVQGPFYGIGAVVATNLVSEKM-AGRAVGQMFA 127

129 AISAGLVLGVPLSSYFGGNFSFEMAMAFYVVINSLAFFITLFFMPEFKKTSRIKVGKQLLSLRY 192
     :.   |||||  .: |  | .            .| |  |  .          |  || . :
128 GLTLANVLGVPGGTWIGVEFGWHTTFIVVAAFGVVALFAILAAIHSTGHGEAKNVKAQLAAFKN 191

193 ALLWISMLAVFCISTGYLGFYSYYSEFLFSVSKMSFTNISLALFIYGFASIIGNNIAGKTLVNH 256
      | ||:     : ||::  | | .     |.    . :.  | | |   |||| : | .
192 PKLLISLAITAVVWTGFMTLYGYIAPIAMHVAGYGESAVTWILVIVGLGLIIGNTLGGHSSDKD 255

257 SNQTLIFASIAMILIYALIFV 277
     |.. :| .||||    |: |
256 LNKSSLFWAIAMIASLVLVGV 276

277 cols, 69 ids (24.9%), 207 diffs (74.7%), 1 gaps (0.4%)
Score 256.0 (103.2 bits), Evalue 9.1e-023
```

FASTA alignment format

Alignments can be saved in FASTA format by using the --fastapairs option, e.g.:

```
usearch --query seqs.fasta --db greengenes.fasta --fastapairs hits.fasta
```

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:

```
usearch --uc2clstr results.uc --output results.clstr

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

User-defined output

Tabbed output in a user-defined format is produced by using the --userout and --userfields options. For example,

```
usearch --query query.fasta --db db.fasta --userout results.user
  --userfields query+target+evalue
```

The --userout option specifies the filename, and the --userfields option specifies one or more field names separated by +.

The output file is tab-separated. The first line contains the field names as specified by the --userfields option; each subsequent line contains one hit. Fields are output in the order given by --userfields. An example output file produced by --userfields query+target+evalue is as follows:

```
query     target    evalue
FQ76998   PF01023   1.2e-12
AZT77876  PF10922   6.7e-23
```
...etc...

Supported user fields are described in the following table.

User field	Description
query	Query sequence label.
target	Target (database, seed) sequence label.
evalue	E-value computed using Karlin-Altschul statistics.
id	%id as reported in other output files, i.e. calculated according to the --iddef option.
id0	%id as if --iddef 0 was specified.
id1	%id as if --iddef 1 was specified.
id2	%id as if --iddef 2 was specified.
id3	%id as if --iddef 3 was specified.
id4	%id as if --iddef 4 was specified.
pctpv	% alignment columns that contain a pair of letters with score > 0 per the substitution matrix.
pctpvz	% alignment columns that contain a pair of letters with score >= 0 per the

User field	Description
	substitution matrix.
pctgaps	% alignment columns that contain a gap.
pairs	Number of alignment columns containing a pair of letters.
gaps	Number of alignment columns that contain a gap.
ins	Number of alignment columns that contain an insertion (gap in query).
del	Number of alignment columns that contain a deletion (gap in target).
intgaps	Number of internal gaps.
tgaps	Number of terminal gaps.
ltgaps	Number of terminal gaps at the left of the alignment.
rtgaps	Number of terminal gaps at the right of the alignment.
qlo	Start coordinate in query (one-based relative to start of sequence).
qhi	End coordinate in query (one-based relative to start of sequence).
tlo	Start coordinate in target (one-based relative to start of sequence or reverse-complemented sequence).
thi	End coordinate in target (one-based relative to start of sequence or reverse-complemented sequence).
qloz	Start coordinate in query (zero-based relative to start of sequence).
qhiz	End coordinate in query (zero-based relative to start of sequence).
tloz	Start coordinate in target (zero-based relative to start of sequence or reverse-complemented sequence).
thiz	End coordinate in target (zero-based relative to start of sequence or reverse-complemented sequence).
pv	Number of alignment columns that contain a pair of letters with score > 0 per the substitution matrix.
pvz	Number of alignment columns that contain a pair of letters with score >= 0 per the substitution matrix.
ql	Full length of query sequence.
tl	Full length of target sequence.
qs	Length of query segment appearing in the alignment.
ts	Length of target segment appearing in the alignment.
cols	Number of alignment columns.
intcols	Internal columns, i.e. number of columns that are not terminal gaps.

User field	Description
opens	Number of gap opens.
exts	Number of gap extensions.
qi	Query sequence index, i.e. the zero-based number 0, 1, 2... of the sequence in the query file.
ti	Target sequence index, i.e. the zero-based number 0, 1, 2... of the sequence in the database.
raw	Raw score = sum of substitution scores minus gap penalties.
bits	Bit score computed from the raw score using Karlin-Altschul statistics.
strand	Strand: one letter '+' (forward strand), '-' (backward strand), or '.' for amino acid sequences.
frame	Signed integer -3, -2, -1, +1, +2 or +3 indicating the frame. For translated searches only, otherwise appears as ".".
aln	Alignment, coded as a string with one letter for each column: M is a pair of letters, D is a delete (gap in target), I is insert (gap in query).
caln	Alignment compressed using run-length encoding, exactly as in the .uc file format.
qrow	Query alignment row, i.e. the aligned segment of the query sequence with gap characters '-' inserted as appropriate.
trow	Target alignment row, i.e. the aligned segment of the target sequence with gap characters '-' inserted as appropriate.

Compressed indexes save memory

The default database index constructed by USEARCH requires a relatively large amount of memory, typically about 10× the size of a FASTA file containing the database sequences. In some situations, this can exceed the amount of installed RAM.

USEARCH also offers a compressed index option which can save a substantial amount of memory. For high-identity search or clustering, say at around 95% identity or above, compressed indexes can also be more sensitive and give faster execution times. A compressed index is used if the --slots option is given.

A compressed index uses a table with a fixed number of slots (--slots *slots* option). For the best possible performance, the number of slots should be set to a prime number. Using non-primes probably doesn't make much difference, but so far I haven't attempted to measure whether it matters or not so safest is to use a prime. You can use this web page to find a prime number close to a given integer: http://www.rsok.com/~jrm/printprimes.html.

Suggested parameter settings

For the best possible trade-off between sensitivity and memory use, the value of the *w* and *slots* parameters should be chosen based on the identity threshold and database size, as explained below. However, the values shown in the table below should be effective for typical nucleotide data. Bits refers to the usearch binary build (32- or 64-bit), Seq. length means the length of a typical short sequence, and Database is the database size in bytes.

Bits	Seq. length	Database	--id	--w	--slots	RAM
32	100 – 1000	≤ 1 Gb	0.97 – 0.99	32	40000003 ($4\times10^7+3$)	≤ 1.3 Gb
32	100 – 1000	≤ 1 Gb	0.90 – 0.96	16	40000003 ($4\times10^7+3$)	≤ 1.3 Gb
64	100 – 1000	≤ 10 Gb	0.97 – 0.99	32	400000009 ($4\times10^8+9$)	≤ 13 Gb
64	100 – 1000	≤ 10 Gb	0.90 – 0.96	16	400000009 ($4\times10^8+9$)	≤ 13 Gb
64	100 - 1000	≤ 32 Gb	0.97 – 0.99	32	1000000007 (10^9+7)	≤ 40 Gb
64	100 - 1000	≤ 32 Gb	0.90 – 0.96	16	1000000007 (10^9+7)	≤ 40 Gb

Choosing the best word length

A compressed index supports arbitrarily long word lengths (--w *w* option) in a fixed-size region of memory as specified by the --slots parameter, while the standard index requires memory that scales like A^w where A is the alphabet size, and therefore can support only short words in a reasonable amount of RAM. Long words can be effective at high identities, where similar sequences often have many shorter words in common but fall below the identity threshold.

In the case of clustering, the database size may not be known in advance, in which case a worst-case estimate should be made of the largest likely size. The word length should be approximately equal to the longest exact word match expected based on the query length and identity threshold. For example, if the shortest query sequences are ~100 letters and the identity threshold is 99%, then we expect an exact word match of at least 50 letters and we would set --w 50.

In general, an estimate of the word length can be made as follows:

$$d = (1 - t)\, L$$
$$w = L / (d + 1)$$

Here, L is a typical short query sequence length, t is the clustering threshold (--id option) and d is the maximum number of differences (substitutions and gapped positions). For example, if t=0.97 and L=250, we find d=7.5 and w=29.4, so a reasonable choice would be --w 30.

This estimate of the word length is quite conservative, and there is usually only a small loss in sensitivity using a larger value, say up to 2× larger than the estimate above, which can be useful if further savings in memory are needed.

Setting the best table size

An estimate of the minimum number of slots to use is:

$$slots = D / w$$

Here, D is the number of letters in the database and w is the word length. So, for example, if the database is a 1Gb FASTA file and w=30, we find $slots$=10^9 / 30 = 3.3×10^7.

If there is enough memory, it is better to use larger values of *slots*. See next for how to estimate the total memory requirement.

Memory requirements for a compressed index

The amount of memory needed for the index can be estimated as follows:

$$bytes\ RAM\ needed\ for\ index = 8 \times slots + D$$

So for our example of a 1Gb database file, the total memory is:

$$8 \times slots + D = 8 \times 3.3\times10^7 + 10^9 = 1.2\ \text{Gb}.$$

This is substantially less than the ~10 Gb that would be needed with an uncompressed index.

Dereplication: discarding identical sequences

Dereplication is the process of discarding sequences that are identical, leaving exactly one copy of each unique sequence. Dereplication can be defined in two different ways: (i) two sequences are identical over their full length, or (ii) one full-length sequence is a substring of another sequence. It is useful to consider these separately because more efficient algorithms are possible for finding full-length matches. It is harder to find identical substrings than identical full-length sequences.

Dereplication of full-length identical sequences

Full-length dereplication is specified by the --derep_fullseq option of the --cluster command. It is important to sort by decreasing length first. For example:

```
usearch --sort reads.fa --output reads.sorted.fa
usearch --derep_fullseq --cluster reads.sorted.fa --seedsout unique.fa
  [--uc results.uc] [--sizeoout] [--minsize n] [--slots n] [--w wordlength]
```

The --sizeout option causes the cluster size to be included in the label. By default, the label is >Cluster*N*, where *N* is 0, 1, 2... etc. If the --sizeout option is given, the label is >Cluster*N*;size=*M* where *M* is the number of identical sequences in the cluster, which will be ≥ 1.

The --uc option specifies a UCLUST format output file. This is useful if you need to know which pairs of sequences were matched.

The --minsize *n* option specifies the minimum size of a cluster. If fewer than *n* identical sequences are found, the cluster is discarded. By default, --minsize is set to zero, which means that this option is ignored.

The --slots *n* option specifies the number of table slots to allocate. Default is $1000003=10^6+3$, which is the first prime number $>10^6$. This number of slots should be prime and for best performance should be greater than *N*, the number of unique sequences in the input; preferably at least 2*N*.

Finding a prime number

You can use this web page to find a prime number close to a given integer: http://www.rsok.com/~jrm/printprimes.html.

Fast dereplication for huge datasets

The --bithash option provides a very fast option for full-length dereplication of large datasets, at the expense of possible false positives. The amount of memory needed is also usually much smaller. Typical usage is:

```
usearch --bithash --derep_fullseq --cluster reads.sorted.fa --seedsout unique.fa
```

For very large datasets, the --mergesort command can be used if --sort runs out of memory.

The --bithash option creates a hash table with 8*K* slots, where $K < 2\times10^9$. The value of *K* can be changed using the --slots option. By default, $K=1000000007=10^9+7$, which is the first prime number $>10^9$. The bithash algorithm saves time and memory by assuming that two sequences with the same hash value are identical, which is not guaranteed to be true. A failure of this assumption causes a false positive. A false positive is a sequence that is unique in the input set, but is incorrectly discarded because it has the same hash value as a different sequence by chance. False positives will be minimized if *K* is much larger than the number of unique sequences in the input.

The amount of memory needed to store the table is *K* bytes, so is 1 Gb by default.

The --uc, --sizeout and --minsize options are not supported if --bithash is used.

You can measure the number of false positives for benchmarking purposes by comparing the number of unique sequences generated with and without the --bithash option.

Dereplication of identical sub-sequences

Subsequence dereplication uses the --derep_subseq option of the --cluster command. Most of the usual options for clustering can be used, with obvious exceptions: e.g., --local and --evalue are not supported. It is important to sort by decreasing length prior to dereplication. For example:

```
usearch --sort reads.fa --output reads.sorted.fa
usearch --derep_subseq --cluster reads.sorted.fa --seedsout unique.fa \
  --slots 40000003 --minlen 64 --w 64 [--sizein] [--sizeout]
```

The --sizeout option appends size=*M* to the seed label in the --seedsout file, where *M* is the size of the cluster. If the --sizein option is also specified, the cluster size will be the sum of sizes specified in labels in the members of that cluster. This enables sizes to be accumulated over multiple clustering steps, e.g. using --derep_fullseq followed by --derep_subseq. This will produce equivalent results to a single --derep_subseq step, but may be more efficient. For example:

```
usearch --sort reads.fa --output reads.sorted.fa
usearch --derep_fullseq reads.sorted.fa --seedsout full.fa --sizeout
usearch --derep_subset full.fa --seedsout sub.fa --sizein --sizeout \
  --slots 1046527 --minlen 64 --w 32
```

Since the results are equivalent, there is no reason to use --derep_fullseq as a preprocessing step unless you find that using --derep_subseq directly on your input set is computationally expensive and you will be repeating these processing steps repeatedly with new data.

It is generally recommended to use --minlen and a compressed index (--slots option) with --derep_subseq.

Denoising: correcting sequencer error

Denoising is the process of correcting errors in reads from next-generation sequencing technologies, including Roche 454 and Illumina.

Denoising support in USEARCH is a work in progress

In this section, I will offer some suggestions, but these have not been well tested. I would like to provide more definitive solutions, but this is challenging for the following reasons (excuses):

- I don't know very much about these sequencers. If someone helps me understand the issues better, I may be able to offer better advice and provide appropriate new features in USEARCH.

- Denoising is a moving target as sequencing technologies change. A solution that works with well with one sequencer may not work so well with the next version of the hardware or with a sequencer from a different manufacturer.

When can you denoise by clustering?

Denoising can be accomplished by clustering when these conditions hold:

- Several reads will cover the same biological sequence, and

- Reads of one biological sequence are approximately globally alignable.

This conditions hold when sequences start from a fixed primer, as in single-region environmental sequencing experiments using the 16S rRNA gene or the fungal ITS region. It may also be true in other situations; I am not clear about this.

I would not expect these conditions to hold for shotgun sequencing, unless the sequencing machines generate several reads for each fragment (this is one of those things that I'm not clear about). With shotgun, you usually have to make contigs from fragments that overlap locally rather than globally. Denoising is then an assembler problem, as far as I know.

Using a consensus sequence

If these conditions hold, reads are approximately globally alignable to one biological sequence, then a multiple alignment of a biological sequence to its reads will look something like this. Read errors are highlighted.

```
G A T G A C G T C A - A G T C A T A G G    Biological sequence
G A T T A C G T C A - A G T C A A A G G    Read 1
G A T G A C G A C A - A G T C A T A G -    Read 2
G G T G A C G T C A A A G - C A T A G C    Read 3
```

The biological sequence can be estimated as the *consensus sequence* derived from the multiple alignment. In each column of the alignment, the most common letter is taken. If the column contains a gap, the column is discarded. In this example, the biological sequence is recovered correctly. In general, there might be some remaining errors but we expect the consensus sequence to be closer than the longest read or a randomly chosen read from the cluster.

The --consout option for cluster consensus sequences

The --consout option of the --cluster command constructs a consensus sequence by computing a multiple alignment of the cluster members. The method used is very fast compared with conventional multiple alignment methods like MUSCLE, so adds very little overhead in terms of execution time. Some additional memory is required, but this is typically not much more than around 20% in the case of nucleotides (more for amino acid sequences).

Strategy for denoising by clustering

Denoising by consensus sequence can be accomplished using the following steps.

1. Quality filter. Discard reads with unacceptably low average quality, and remove low-quality positions at the beginning or end of a read.

2. Create clusters in an attempt to bring together reads of the same biological sequence.

3. Create a multiple alignment of each cluster.

4. Extract a consensus sequence for each cluster.

5. Filter consensus sequences for chimeras, if appropriate.

Quality filtering must be done as a preprocessing step. Steps 2, 3, 4 and 5 can be done with USEARCH. I am not sure what parameters to recommend, but I would suggest something like the following.

Cluster at an identity threshold which corresponds to ~2× the expected maximum number of read errors. For example, suppose the average read error rate after quality filtering is believed to be around 1%. Then we might expect reads to contain up to 2% errors, allowing for some that have more errors than the average. A reasonable clustering threshold would then be 96% identity (4% difference), because the seed sequence might be 2% diverged from the correct sequence, and other cluster members may be diverged 2% at some other positions, for a total distance of 4% between the seed and the other cluster members. Use the --consout option in the final clustering step to generate the estimated biological sequences. Note that --consout uses *size=* fields in the sequence labels, if present, when computing the consensus sequences, providing that you specify the --sizein option. This ensures that the true number of observations (the underlying number of reads) is used to calculate the majority vote in each column if you use multiple clustering steps such as dereplication followed by clustering. Typical commands might be as follows.

Warning

I do not necessarily recommend using parameters such as the clustering threshold from this example because I am not familiar enough with the issues. You should choose parameters based on your best understanding of the sequencing technology.

```
usearch --sort reads.fa --output reads.sorted.fa
usearch --cluster reads.sorted.fa --id 0.96 --consout denoised.fa \
  --minlen 64 --w 20 --slots 16769023 --leftjust --idprefix 5 --minsize 4
```

Using --leftjust and --idprefix is recommended if the reads are expected to start from a fixed location in the biological sequence, such as an exact match to a primer sequence. This usually improves both speed and sensitivity.

Replacing CD-HIT-454 with USEARCH

The cd-hit software suite has a program cd-hit-454 designed to identify "natural and artificial duplicates from pyrosequencing reads" (I'm not clear exactly what that means), and a web server that uses CLUSTALW to construct consensus sequences from clusters found by the cd-hit-454 program. I don't really understand the motivation for the design of this program or web server, but I can recommend options for usearch that work in a similar way to cd-hit-454, but are significantly faster, smaller and more sensitive. Typical usage is:

```
usearch --sort reads.fa --output reads.sorted.fa
usearch --cluster reads.sorted.fa --id 0.96 --leftjust --idprefix 5 \
 --w 32 --slots 16769023 --seedsout seeds.fa --consout cons.fa
```

The seeds.fa file will contain the seed sequences, equivalent to those produced by the stand-alone cd-hit-454, and the cons.fa file will contain consensus sequences, similar to those produced by the web server. The clustering threshold --id 0.96 should be adjusted if needed according to the expected error rate after your quality filtering (the equivalent parameter in cd-hit-454 is -c, which defaults to 0.98). The --maxqgap and --maxtgap options to usearch provide similar functionality to the -D option of cd-hit-454.

Using the --minsize option to discard small clusters

The smallest clusters of reads, e.g. singletons, are likely to be sequencing artifacts. The --minsize option sets the minimum size of a cluster that will be output to the --seedsout or --consout file.

Chimera detection in single-region reads

For single-region experiments such as 16S and ITS, it is important to filter for chimeric sequences formed during the PCR amplification stage that is generally used prior to sequencing. In most such experiments, a substantial fraction of the unique sequences in the set of amplicons is chimeric. I would generally recommend denoising first, then using UCHIME to filter for chimeras. A fast version of UCHIME is implemented in USEARCH; usage is described in the separate UCHIME manual.

Amplicon and abundance estimation for UCHIME *de novo* mode

The *de novo* mode of UCHIME requires an input file that contains a set of estimated amplicon sequences with their abundances annotated using the *size*= attribute in the label, e.g.:

```
>FQ56TRV12;size=14
```

A reasonable method for computing abundance is to sum the number of raw reads corresponding to the estimated sequence, i.e. the total number of reads in the cluster (or hierarchy of clusters) for the sequence.

Warning

It is important that the reads come from *one* sequencer run (strictly, one PCR step), otherwise abundances may not be directly comparable.

Estimating amplicon sequences and abundances is a denoising problem, and I recommend that you read the previous section on denoising for more background. It is an open research problem to determine the best strategy for amplicon and abundance estimation. One challenge is that it is often a goal to determine operational taxonomic units (OTUs) by clustering at a threshold chosen to approximate species, typically 97%. In this case, clustering at 96% or lower may not be appropriate because multiple species could be merged during the denoising process. It might then be more effective to use a threshold of, say, 99% or 98% and extract consensus sequences like this:

```
usearch --cluster reads.sorted.fa --id 0.98 --consout denoised.fa --sizeout \
  --minlen 64 --w 20 --slots 16769023 --minsize 4
```

If you use the --sizeout option, then the denoised.fa file is suitable for input to UCHIME *de novo* mode. However, as noted, it is an open research problem to determine the most effective procedure for estimating amplicons.

OTU identification using USEARCH

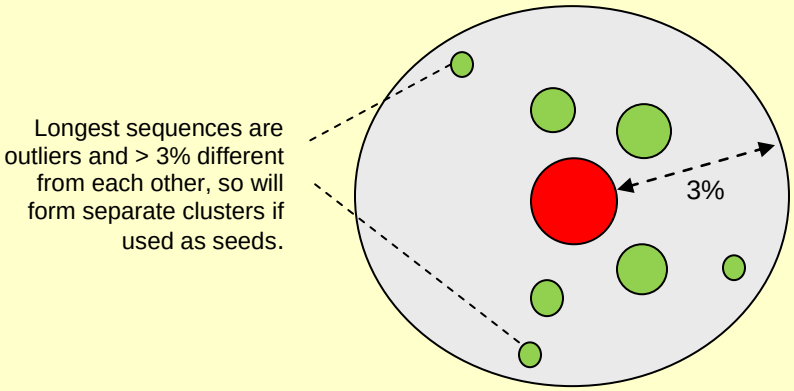
USEARCH is often used in experiments that sequence a single region such as the bacterial 16S rRNA gene or the fungal ITS region in order to identify Operational Taxonomic Units (OTUs). Often, reads are clustered at 97% in an attempt to identify species. From now on, I will assume a 97% threshold, though this could be changed, e.g. to a lower threshold in order to approximate genus instead of species.

In this section we assume that the reads start from a single primer so are globally alignable, as opposed to shotgun sequences that will in general only be locally alignable where they overlap.

The naive approach is to sort by decreasing length, then use UCLUST with --id 0.97 and declare that each cluster is an OTU. This approach has several potential problems.

Problems with naive 97% clustering of OTUs

Sorting by length causes longer reads to become seeds. However, longer reads tend to have more errors due to factors including (i) spurious insertions, especially in homopolymers, and (ii) a decline in quality towards the end of the read. A longer read will therefore tend to be an outlier relative to the biological sequence that would make a better seed for the cluster, and the effect of this is often to split a "natural" OTU into two or more clusters and to produce a seed that is diverged from the underlying biological sequence, as illustrated below.



Schematic illustration of a cluster of reads corresponding to a single species. Each dot indicates a unique sequence. There are several dots for one species because of read errors and / or biological variation within the species (strains). The size of a dot indicates the abundance of the sequence, i.e. how many identical (or very similar) reads have that sequence. The red dot indicates a biological sequence that is the best representative for the OTU since it is the most common in the sample. The longest sequences tend to be outliers, so are not optimal seeds.

Even if quality filtering has been applied, there will be a remaining fraction of the reads that have larger numbers of errors than their quality scores indicate. These reads tend to produce clusters with one, or a few, poor-quality sequences that should be assigned to some other OTU but fall outside the 97% threshold due to errors.

Finally, a substantial fraction of the reads are typically generated from chimeric amplicons, leading to clusters which do not correspond to biological sequences.

Strategies for improving OTU identification

USEARCH offers support for the following strategies for improving OTU identification, as follows.

Sorting by abundance (cluster size)

Sorting by cluster size is done by the --sortsize command. Larger clusters tend to indicate more reliable sequences due to two effects.

(i) Biological sequences should tend to be more abundant in the amplicon pool than PCR artifacts such as chimeras because the biological sequences will have undergone more rounds of amplification.

(ii) If there are many identical or very similar reads, then the most probable explanation is that these are accurate reads of a single biological sequence. It is not likely that they are bad reads that are similar by chance. However, the seed sequence for a larger cluster may be an outlier relative to the underlying biological sequence. This can be addressed by denoising (next).

Denoising by generating consensus sequences

Consensus sequences are generated by the --consout option to the --cluster command. For discussion, see the section on denoising.

Chimera filtering using the UCHIME algorithm

Chimera filtering is done using the --uchime command. This is discussed in detail in the UCHIME documentation, which is maintained separately; see this web page: http://www.drive5.com/usearch/usearch_docs.html.

Discarding small clusters

The --minsize option to the --sortsize command can be used to discard small clusters. Since small clusters tend to be spurious, this may give a significant improvement in specificity by reducing the number of spurious OTUs, with only a minimal reduction in sensitivity as only a small fraction of the discarded clusters are likely to be derived from valid biological sequences.

Suggested pipeline

A suggested pipeline for generating OTUs is described in the following table. This pipeline exploits these strategies to improve the quality of the final set of OTUs. Of course, you should understand the thinking behind the design and consider making adjustments for your particular situation.

Summary of suggested OTU pipeline

Step	Options	Comments
1. Dereplication.	--derep_subseq	This step is useful if the typical cluster size is > 1, enabling a useful estimate of abundance for step 2. Otherwise it might be better to skip steps 1 and 2, and start at step 3.
2. Abundance sort.	--sortsize	May give better seeds for step 3. If there is a lot of length variation in the reads, a decreasing length sort might be better.
3. 97% first pass.	--id 0.97	Cluster at 97% and generate consensus (not seed!) sequences.
4. Abundance sort.	--sortsize --minsize 4	To get better seeds for step 5, discarding small clusters which will tend to be spurious. Threshold of 4 is somewhat arbitrary; other values might work better depending on the data and how important it is to identify very rare species in the reads.
5. 97% second pass.	--id 0.97	Cluster at 97% and generate seed or consensus sequences. It is not clear whether creating consensus sequences will give better results at this stage. After this step, we have a set of denoised sequences.
6. UCHIME *de novo*.	--uchime	Filter for chimeras using *de novo* identification.
7. UCHIME ref. db.	--uchime --db	Filter for chimeras using reference database.

Example OTU pipeline commands

An example implementation of the suggested pipeline is given below. Note that reads should be quality filtered first. I would expect this to work well with up to about two million reads of lengths 200nt or more. For larger datasets or shorter reads, some adjustments may be needed. To maximize sensitivity, you could increase the --maxrejects parameter for the --cluster commands, e.g. use --maxrejects 256. The 32-bit version of USEARCH can run 10^6 reads through these commands on a laptop in a few minutes in about 1 Gb of RAM.

```
usearch--cluster reads.fa --derep_subseq --seedsout derep.fa \
  --minlen 200 --w 64 --slots 16769023 --sizeout

usearch --sortsize derep.fa --output derep_sorted.fa

usearch --cluster derep_sorted.fa --id 0.97 --consout cons.fa \
  --w 20 --slots 16769023 --sizein --sizeout --usersort

usearch --sortsize cons.fa --output cons_sorted.fas --minsize 4

usearch --cluster cons_sorted.fas --id 0.97 --seedsout cons97.fa \
  --usersort --sizein --sizeout --w 20 --slots 16769023

usearch --uchime cons97.fa --nonchimeras nonchimeras.fa

usearch --uchime nonchimeras.fa --db refdb.fa --rev --nonchimeras otus.fa
```

UBLASTX: Translated ORF search

UBLAST supports translated searches of nucleotide sequences against a protein database containing amino acid sequences. This is somewhat similar to BLASTX, except that ORFs are used as queries. This makes more effective use of the U-sort and S-sort heuristics.

Frame-shifts

The current UBLASTX implementation does not allow frame-shifts within an alignment. However, frame-shifts can be inferred from hits to ORFs in different frames on the same strand.

ORF identification

An ORF begins at the start of the sequence or with a START codon (ATG), and ends at a STOP codon (TAA, TAG or TGA) or the end of the sequence. Please let me know if you would like support for non-standard genetic codes. The minimum number of amino acids in the ORF is set by the --mincodons option (default 20).

The --orfstyle option controls how ORFs are defined. The value is created by adding up the following integers.

Value	Description
1	Allow an ORF to start at the beginning of a sequence, even if this is not a START codon (default cannot start before the first START codon).
2	Allow an ORF to start immediately following a STOP codon (default cannot start before the first START following a STOP).
4	Allow an ORF to end at the end of the nucleotide sequence (default must be terminated by a STOP codon).
8	Include the terminating STOP codon, if any, in the translated sequence (default do not include the STOP).

Default is --orfstyle 7. Since 7=1+2+4, ORFs are identified as subsequences that do not contain a STOP, which is appropriate for shotgun metagenomic reads that may only partially cover a gene and may contain errors such as frameshifts.

Search and output

The translated amino acid sequence for each ORF is used as a query sequence to search the target database. The --maxrejects option is especially important here if the database is large and / or if low-identity matches are designed, typically you will need to specify a larger value of --maxrejects (say, 100 or 1000) to achieve good sensitivity with low-identity proteins.

UHIRE: hierarchical clustering, clumping and large multiple alignments

The --uhire command performs hierarchical clustering with the goal of generating clusters of approximately a predetermined size (*clumps*). Sequences within a clump should be more similar to each other than to sequences in other clumps. This is intended to reduce the dataset size to be tractable for more expensive algorithms, such as multiple alignments. Basic usage is as follows.

```
usearch --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), unlike --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:

```
usearch --uhire reads.sorted.fasta --clumpfasta myclumpdir/ --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, which must exist. Sequences for each clump will be stored in these files:

```
myclumpdir/clump.0
myclumpdir/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.

Warning

This method was primarily designed to support clumping (see below). Clusters at levels below the first (highest identity) level will tend to be more diverse than clusters obtained in a single step. Say the first two levels are 99% and 98%. The 98% step uses seeds from the 99% step as input. Suppose a cluster at 99% includes two sequences S and A where S is the seed and A is another sequence such that pctid(A,S) ≥ 99%. A is discarded when the 98% clustering is done. Now suppose T is the seed at 98%, so pctid(S,T) ≥ 98%. There is no guarantee that A has ≥ 98% id with T, it may be less, and in fact we should expect such cases because A can be 'further' from T than S is. So clustering all sequences including A at 98% will tend to give different numbers of clusters than the hierarchical method.

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. This field is also 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.

Large multiple alignments

MUSCLE can create alignments of up to perhaps 10,000 to 20,000 sequences, depending on the available memory and sequence lengths. 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 myclumpdir
usearch --uhire seqs.sorted.fasta --clumpfasta myclumpdir/ --maxclump 5000
```

The clumps and the master set are then aligned using MUSCLE. For example (bash syntax):

```
mkdir clumpalns
cd myclumpdir
for filename in clump.* master
do
   muscle -in $filename -out ../clumpalns/$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.

```
usearch --mergeclumps clumpalns/ --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 28 must contain an identical sequence with label >GF2FOAC01AU7TA, this correspondence is used to merge the alignments of each clump into a single multiple alignment.

Parameter tuning

Where possible, I recommended that you tune parameters to obtain a good trade-off between speed and sensitivity. Following are some suggestions for how this can be achieved.

Improved sensitivity for distant proteins

Two options to try when clustering or searching with distantly-related proteins are --nb and --ssort. If --nousort is specified, then --nb is the default, otherwise it may give improved sensitivity with only a small cost in speed.

Choose suitable quality measures

Typical goals of tuning are find parameters that give high-quality results with the shortest possible execution times. This requires a measure of quality. The log file (--log option) reports execution time, memory use and some statistics on search and clustering which could be used as quality measures. Alternatively, you could write a script to parse one of the output files: the --uc, --blast6out and --userout files are well suited for this purpose.

Quality measures for clustering

For clustering, sensitivity can be measured by (i) the number of clusters or, equivalently, by the average cluster size, and (ii) the average identity of a cluster member to the seed. Fewer clusters (larger clusters) indicate higher sensitivity, and higher average identity with the seed indicates that a better cluster assignment is made in cases where more than one seed matches.

Quality measures for database search

For database searching, sensitivity can be measured by (i) the fraction of query sequences that are matched to the database at the given E-value or identity threshold, and (ii) the average similarity of a hit. It is generally better to measure similarity by identity even if an E-value threshold is used, because E-values range over many orders of magnitude so the mean or median is not very informative.

Construct a query set that is small enough for testing

If a typical query set is so large that repeated testing is unreasonably slow, then the size of the set can be reduced. For a database search application, this can be done by taking a random sample. For clustering, a random sample is not suitable because this tends to reduce the average size of a cluster but not the number of clusters, which increases the average number of rejections per query. A smaller set can be obtained by clustering the input sequences and taking a subset of the clusters. This should give a subset with similar redundancy to the original.

Test with increasing values of --maxrejects

First test with --nousort, if possible. This causes the entire database to be searched and thus guarantees the best possible hit for a given query, but may be unreasonably slow. Either way, set --maxaccepts 0 and try a range of values of --maxrejects, for example 8, 16, 32, 64, 128, etc. Increase the value until your quality measure(s) do not increase significantly. Regardless of whether you were able to use --nousort, you now have an estimate of the best possible results and the minimum value of --maxrejects that gives you good enough results.

Test with increasing values of --maxaccepts

Once you know the highest values of your quality measures that can be achieved on your test data, you can experiment with changing parameters and obtain an acceptable compromise: e.g., you might be satisfied with achieving 90% of the best possible sensitivity if the speed is improved by a factor of 100.

U-sort word length

The --w option sets the word length used for U-sorting. The default is 5 for amino acids and 8 for nucleotides. Try different values to check the effect on memory use and speed. Start by adding and subtracting one. If adding or subtracting one gives a better result, try changing by two, and so on.

Tune alignment heuristics

You can measure the impact of alignment heuristics by testing with and without --nofastalign. If --nofastalign is specified, heuristics are disabled, and the execution time may be tens or hundreds of times slower. Measures such as the number of clusters may not change much despite a significant reduction in biological accuracy. (This happens when over-aggressive heuristics produce many bad alignments without this causing a bias in the quality measure). Therefore, it is best to use biologically informed reference data if possible in order to test the effects of the heuristics. (Of course, biological reference data are preferred for tuning all parameters). The recommended heuristics to try are summarized in the following table. For all numerical parameters except seed word length, larger values tend to increase execution times and smaller values are faster but may degrade accuracy, though often the effect on accuracy is negligible. The effect of the seed word length is less predictable. Reducing the band radius is often an effective way to improve speed without a significant loss in quality.

Option	Heuristic
--wordcountreject, --nowordcountreject	Enables / disables word count rejection. For higher identities, tends to improve speed when enabled, but may induce false negatives.
--k	Seed word length.
--nb, --nonb	Use / don't use word neighborhoods (amino acids only). Using neighborhoods improves sensitivity; effect on speed varies.
--seedt	Seed score threshold (applies only for amino acids and if --nb is specified).
--xdrop_u, --xdrop_g, --xdtop_ug, --xdrop_nw	X-drop.
--band	Radius for banded dynamic programming.

USTAR: Fast multiple alignment of clusters

USEARCH can create a multiple alignment of each cluster found by UCLUST. This requires three steps: 1. clustering (--cluster), 2. extraction of S (seed) and H (hit) records, 3. conversion to FASTA (--uc2fastax) and 4. inserting additional gaps (--staralign).

```
usearch --cluster seqs_sorted.fasta --uc results.uc --id 0.97
grep "^[SH]" results.uc > sh.uc
usearch --uc2fastax sh.uc --input seqs_sorted.fasta --output sh.fasta
usearch --staralign sh.fasta --output aligned.fasta
```

The algorithm creates a 'star' alignment using pair-wise alignments to the seed, so the seed is the center of the star. This method emphasizes very high 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, e.g. for identifying highly conserved segments. 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 the UHIRE algorithm for a method that can create high-quality alignments of very large sets.

Gap penalties and substitution scores

USEARCH supports a comprehensive set of gap penalty and substitution score options. Different options apply to local vs. global alignments. All alignment scores and penalties in USEARCH can be specified as integer, floating point or real values.

E-value calculation

E-values are calculated by Karlin-Altschul statistics assuming default values for substitution scores and gap penalties. If you change the alignment scoring parameters, then E-value parameters must be adjusted accordingly. This is not a trivial exercise; the easiest way is usually to borrow parameters from some other program, such as BLAST. Contact me if you need more information.

Substitution scores for nucleotides

Two substitution scores are used for nucleotide sequences: match and mismatch. The match score must be positive and the mismatch score must be negative. For local alignments, the absolute value of the mismatch score should be greater than the match score. If you use non-default substitution scores, you should probably also specify appropriate gap penalties for those scores.

Score	Option	Default
Match	--match	1.0
Mismatch	--mismatch	–2.0

Substitution matrix for amino acids

By default, the BLOSUM62 matrix is used for amino acid sequences. The user can specify a different matrix by using the --maxtrix *filename* option. The matrix should be formatted as for NCBI BLAST. Integer or floating-point values may be used. If a different matrix is specified, you should probably also specify appropriate gap penalties for that matrix.

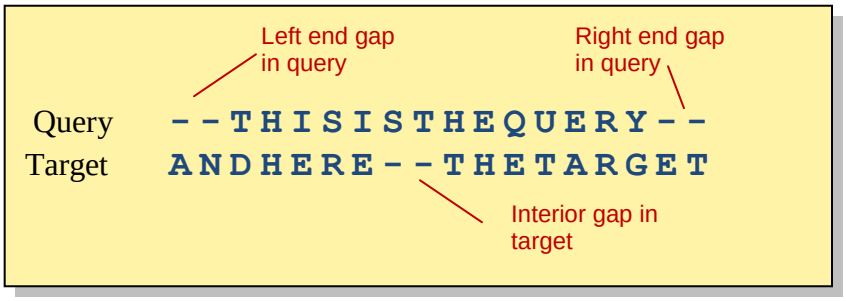
Gap penalties for local alignments

The --lopen and --lext options specify open and extend penalties for local alignments.

Penalty	Option	Default
Local gap open	--lopen	10.0
Local gap extend	--lext	1.0

Gap penalties for global alignments

Up to 12 separate penalties can be specified: all combinations of query / target, left / interior / terminal, and open / extend can be assigned different penalties.



Default penalties are as follows.

Penalty	Default
Interior gap open	10.0 nt, 17.0 aa
End gap open	1.0
Interior gap extend	1.0
End gap extend	0.5

End gaps (also called 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 gap=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 USEARCH to crash because no alignment is possible, and this condition is currently not handled gracefully (this is a really a bug; better would be to reject the target, but this is hard to implement).

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 a small subset and reviewing the --blastout file to make sure that the alignments look reasonable for your application.

Sequence identity

Sequence identity can be defined in many different ways; see for example this web page and its literature references: http://openwetware.org/wiki/Wikiomics:Percentage_identity. Identity is usually defined to be a ratio where the numerator is the number of identities (columns containing the same letter) in an alignment. Many choices are possible for the denominator, each of which has pros and cons in different applications. Common choices include:

- The number of columns in the alignment (terminal gaps may be included or excluded).
- The length of the shorter sequence.
- The length of the longer sequence.
- The average sequence length.
- The number of columns containing letter pairs (i.e., gaps are ignored).

Terminal gaps

Some definitions of identity treat terminal gaps as special cases. This can be important, e.g. if fragments are being aligned to full-length sequences, in which case terminal gaps are experimental artifacts rather than evidence of insertions or deletions. It should be noted that definitions of identity that count terminal gaps differently from internal gaps are more sensitive to details of the algorithm used to generate the alignment, and in particular to gap penalties. Problems may be caused if a short motif is misaligned close to a terminal, like this.

```
Query:    -XX------XXXXXXXXXXXXX------
Target:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Presumably, the correct alignment would look more like this:

```
Query:    ------- XXXXXXXXXXXXXXX------
Target:   XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

If gapped columns count as differences and terminal gaps are discarded, then the first alignment may have much lower identity.

The --iddef *n* option specifies how identity should be calculated, where *n* is 0, 1 ... etc. The default is --iddef 0. The definitions used are summarized in the following tables.

Variable	Description
Identities	Number of columns containing two identical letters. *(See notes below re. wildcards).*
Diffs	Number of columns that do not contain identities. In other words, the number of columns containing gaps or mismatches.
InternalDiffs	Number of columns that do not contain identities, excluding terminal gaps if any. In other words, the number of columns containing internal gaps or mismatches.
Columns	Number of columns in the alignment. Includes terminal gaps, if any.
InternalColumns	Number of columns, excluding any terminal gaps.
QueryLength	Length of the query sequence.
TargetLength	Length of the target (database or seed) sequence.

--iddef option	Name	Definition
0	Default	Identities / min(QueryLength, TargetLength) *(See notes below re. wildcards).*
1	All diffs	1 – (Diffs / Columns)
2	Internal diffs	1 – (InternalDiffs / InternalColumns)
3	MBL	1 – (Diffs / TargetLength)
4	BLAST	1 – (Identities / Columns)

Default definition of identity, --iddef 0

The default definition, --iddef 0, uses the length of the shorter sequence as the denominator. This definition is the one used by the CD-HIT program, and was originally used by UCLUST to facilitate comparison of the two programs.

Since all gap columns are discarded, this definition can report 100% identity despite gaps in the shorter sequence. Consider the following example.

```
Query:   SEQ-ENCE
Target:  SEQVENCE
```

Here, there are 7 identities and the length of the shorter sequence is also 7, giving Id = 7/7 = 100%.

All-diffs definition, --iddef 1

The all-diffs definition (--iddef 1) considers every gap column and every mismatch to be a difference, which is achieved by using the number of columns in the alignment as the denominator. This is the same as 1 – edit_distance / columns, where edit_distance is the smallest number of insertions, deletions and substitutions that transform one sequence into the other. In the above example, there are 8 columns in the alignment, so Id = 7/8 = 87.5%.

Internal diffs definition, --iddef 2

The internal diffs definition (--iddef 2) is similar to all-diffs, except that terminal gaps are not included in the number of columns. See above (*Terminal gaps*) for a discussion of a potential problem with this definition. This may be more appropriate if fragment sequences (e.g., partial 16S genes from a short-read sequencing experiment) are aligned to full-length sequences (complete genes in a reference database). Consider this example.

```
Query:    ---V-NC-
Target:   SEQVENCE
```

Here, there are 4 columns after terminal gaps are discarded, so the internal diffs Id = 3/4 = 75%, while the default Id = 3/3 = 100% and the all-diffs Id = 3/8 = 37.5%.

Marine Biological Laboratory definition, --iddef 3

The MBL definition (--iddef 3) is similar to all-diffs, except that a gap of any length (i.e., consecutive series of gap columns) counts as a single difference. Both internal and terminal gaps are counted. Identity is defined as:

1.0 – [(mismatches + gaps)/(longer_sequence_length)]

Notice that unlike other definitions, this does not use the number of identities as the numerator. Consider the following example.

```
Query:    --QVDNC-
Target:   SEQVENCE
```

Here, mismatches = 1 and gaps = 2 so Id = 1 – (1 + 2)/8 = 72.5%. In theory, this expression can be negative, in which case it is considered to be zero.

BLAST definition, --iddef 4

This definition is the one used by NCBI BLAST: the number of columns in the alignment that contain identical matches divided by the total number of columns. If the alignment is global, terminal gaps are included in the total number of columns, so this definition is probably not suitable if fragment sequences are aligned to full-length sequences using global alignments.

Wildcard letters

Wildcard (ambiguous or unresolved) letters include N (nucleotides) and X,B and Z (amino acids). Usearch treats any letter not in the standard 4- or 20-letter alphabets as a wildcard. There are two situations where wildcards may appear: (i) computing substitution scores when calculating the alignment, and (ii) computing identity from the alignment. A wildcard

substitution score is always zero. If the default definition is used (--iddef 0), then when computing identity, a column that contains a wildcard aligned to another letter is discarded; columns that align a wildcard letter to a gap are retained. With other values of --iddef, wildcards are treated exactly like other letters, so e.g. NN is an identity and NA is a mismatch (nucleotides).

UCHIME: Chimeric sequence detection

UCHIME is documented in a separate manual. Please visit this page:

http://www.drive5.com/usearch/usearch_docs.html

Memory requirements

The amount of memory needed for the database index with default options is approximately 10x the size of a FASTA file containing the database. When clustering, the database is the final set of seed sequences, which can be written to a FASTA file by using --seedsout. A more accurate 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.

Compressed indexes

Compressed indexes typically use much less memory, and can actually be more sensitive at high identities (say, 95% and above). See the compressed indexes section for more details.

Database stepping

With a default (non-compressed) index, the --dbstep *n* option reduces the memory required by a factor of roughly *n* for large databases (less for smaller database). However, sensitivity tends to be reduced when clustering or searching at lower identities (say, below 80%). Using --dbstep reduces the number of processor operations required to search the in-memory database index, which might be expected to improve speed, but in practice execution times are often slower due to a reduction in cache coherence.

Reducing redundancy

If you have very similar sequences in your database, 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 to pre-process your database. For sure, if you have 100% identical sequences these should be deleted since they can adversely affect sensitivity in a U-sorted search.

Trimming 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. This is true for example of the NCBI NR protein database, which has many very long labels. In such cases 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. You can use the --trunclabels option to trim labels by discarding any text after the first white space (blank or tab).

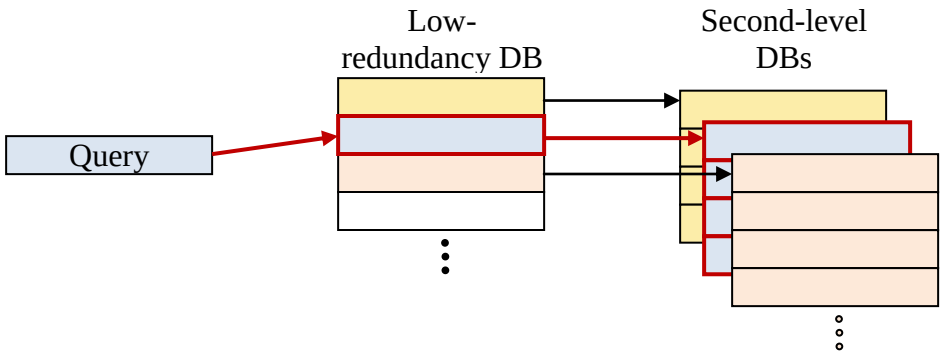
Splitting the database

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 machines 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 algorithms 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 reference

Algorithms

Algorithm	Description	Required options
UCLUST	*De novo* clustering	--cluster *fastafile*
UCLUST	Search + clustering	--cluster *fastafile* --db *fastaflle*
UBLAST	Database search	--query *fastafile* --db *fastafile*
UCHIME	*De novo* chimera detection	--uchime *fastafile*
UCHIME	Chimera detection with reference database	--uchime *fastafile*--db *fastafile*

Sorting

Command	Command line
Sort sequences by length	--sort *fastafile* --output *fastafile* --mergesort *fastafile* --output *fastafile* [--split *size*]
Sort UCLUST file by cluster nr.	--sortuc *ufile* --output *ucfile*

File format conversions

From	To	Command line
UCLUST (.uc)	FASTA	--uc2fasta *ucfile* --output *fastafile*
UCLUST (.uc)	Annotated FASTA	--uc2fastax *ucfile* --output *fastafile*
UCLUST (.uc)	CD-HIT (.clstr)	--uc2clstr *ucfile* --output *clstrfile*
CD-HIT (.clstr)	UCLUST (.uc)	--clstr2uc *clstrfile* --output *ucfile*

Output files

Option	Format	Description
--uc *filename*	UCLUST	Tab-separated file designed primarily for clustering pipelines but can also be useful for search. One record for each input sequence giving its cluster assignment, identity and alignment; and one record for each cluster giving its size and average identity. Supported by UCLUST and UBLAST.
--blastout *filename*	BLAST-like	Human-readable format similar to BLAST.Supported by UCLUST and UBLAST.
--blast6out *filename*	Tab-separated	Tabbed format with one record per hit. Compatible with the -m8 or -outfmt 6 option of NCBI BLAST. Supported by UCLUST and UBLAST.
--userout *filename*	Tab-separated	Tabbed format with one record per hit, fields specified by the --userfields option (see manual). Supported by UCLUST and UBLAST.
--seedsout *filename*	FASTA	Seed sequences, i.e. the non-redundant or reduced redundancy set of sequences after clustering. Supported by UCLUST only.
--consout *filename*	FASTA	Consensus sequences, one for each cluster. Computed from a multiple alignment of the cluster. The majority letter is taken from each column, or the column is discarded if the majority symbol is a gap. Terminal gaps are ignored unless the --cons_termgaps option is specified.
--fastapairs *filename*	FASTA	Pair-wise alignments in FASTA format. Supported by UCLUST and UBLAST.

Database search order

Option	Description
--[no]usort	[Do not] test database sequences in U-sorted order, i.e. in order of decreasing number of words in common. If --nousort is specified, the entire database is tested and search termination options are ignored or give an error. Default is --usort.
--[no]ssort	Change U-sort order to better correlate with evolutionary distance. Applies to amino acid databases only. If --query and --evalue are used, then --ssort is the default, otherwise --nossort is the default.
--stable_sort	Specifies that a stable algorithm should be used for U-sorting. This may be a little slower, but gives reproducible results when a given query has the same word count with more than one target sequence, which can cause the accepted target to change if a non-stable sort is used. Default is to use a non-stable sort.
--w *n*	Word length for U-sorting. Default 5 for amino acids, 8 for nucleotides. Changing the word length changes speed, sensitivity and memory requirements in different ways depending on the input data and the index type (compressed or standard).

Search termination

These options determine when a U-sorted search terminates. These options are ignored if --nousort is specified; the entire database is searched.

Option	Description
--maxaccepts *n*	Maximum number of accepted targets. Zero means that this option is ignored (i.e., zero means infinity). Default 1, unless --maxtargets is specified, in which case the default is zero. Increasing --maxaccepts improves sensitivity and also the probability that the best possible hit is found, at the expense of slower times. If --maxaccepts is increased, you should generally increase --maxrejects also.
--maxrejects *n*	Maximum number of rejected targets. Zero means that this option is ignored (i.e., zero means infinity). Default is 32. Increasing --maxrejects improves sensitivity by reducing the probability of a false negative, i.e. failing to find a possible hit, at the expense of slower times.

Accept / reject criteria

Accept / reject criteria specify one or more sequence similarity thresholds. At least one of --id or --evalue must be specified. Thresholds are combined with AND, so all must be satisfied for a query-target match to be accepted.

Option	Description
--id *f*	Minimum identity, as a value 0.0 to 1.0, meaning 0% to 100% identity. The --iddef option determines how identity is defined. There is no default value.
--evalue *E*	Maximum E-value. There is no default value.
--queryfract *f*	Minimum fraction of the query sequence that is covered by its alignment to the target, as a value 0.0 to 1.0, meaning 0% to 100% coverage. Coverage is defined as the number of letters in the query that are aligned to letters in the target, divided by the length of the query sequence. Default is 1.0, meaning that the option is effectively ignored.
--queryalnfract *f*	Minimum fraction of the query sequence that is covered by its alignment to the target, as a value 0.0 to 1.0, meaning 0% to 100% coverage. Coverage is defined as the number of letters in the query that appear in the alignment, divided by the length of the query sequence. Default is 1.0, meaning that the option is effectively ignored. This option is useful only for local alignments as all letters of the query sequence always appear in a global alignment, so the coverage by this definition is always 100%. The difference between --queryalnfract and --queryfract is the handling of gapped columns. With --queryalnfract, a letter is counted if it is aligned to a gap in the taget, with --queryfract it is not counted.
--targetfract *f* **--targetalnfract** *f*	Exactly as --queryfract and --queryalnfract, but for the target sequence.
--idprefix *n*	The first *n* letters of the query sequence must be identical to the first *n* letters of the target sequence. Default 0. This option is a special case because (i) it is applied before an alignment is constructed, which can save significant execution time, and (ii) failures to match do not count for --maxrejects.
--idsuffix *n*	The last *n* letters of the query sequence must be identical to the first *n* letters of the target sequence. Default 0. This option is a special case because (i) it is applied before an alignment is constructed, which can save significant execution time, and (ii) failures to match do not count for --maxrejects.
--leftjust	The alignment is rejected if there is a left terminal gap. If this option is given, it is recommended to also use the largest reasonable --idprefix value for improved efficiency.
--rightjust	The alignment is rejected if there is a right terminal gap. If this option is given, it is recommended to also use the largest reasonable --idsuffix value for improved efficiency.
--[no]wordcountreject	[Do not] reject a target sequence if it has too few words in common. The number of words in common is used to estimate identity, which is faster

	than calculating identity from an alignment, but can give some false negatives. Applies only to global alignments and on if --id is used as an accept threshold. Default is --wordcountreject.
--iddef *n*	Definition of sequence identity. See section on sequence identity for details. Default 0.

Weak match criteria in UBLAST

Weak matches are reported in output files but are not considered accepts, will not terminate a U-sorted search, and do not match a query to a cluster. Weak criteria are also combined with AND. Weak matches will also be reported by UCLUST, though this is rarely useful in practice.

Option	Description
--weak_id *f*	Minimum identity, as a value 0.0 to 1.0, meaning 0% to 100% identity. The --iddef option determines how identity is defined.
--weak_evalue *E*	Maximum E-value. There is no default value.

Alignment style options

Option	Description
--global	Global alignments. This is the default for UCLUST, i.e. if --cluster is specified.
--local	Local alignments. This is the default for UBLAST, i.e. if --query is specified.
--[no]gapped	[Do not] make gapped alignments. If --nogapped is specified, ungapped alignments will be created. The --nogapped option cannot be used if --global is specified. Default is --gapped.

Alignment scoring parameters

Note that if you change these parameters, then E-values will not be calculated correctly unless the K and Lambda parameters for E-value calculation are adjusted accordingly. If you don't need E-values, e.g. because you use global identity as your similarity measure, then you don't need to adjust K and Lambda.

Option	Description
--match *s*	Match score for nucleotides. Default 1.0. Must be > 0.
--mismatch *s*	Mismatch score for nucleotides. Default –2.0. Must be < 0. For local alignments, absolute value should be greater than --match.
--lopen *s*	Gap open penalty for local alignments. Default 10.0. Must be > 0.
--lext *s*	Gap extension penalty for local alignments. Default 1.0. Must be > 0.
--gapopen *spec*	Specifies gap open penalties for global alignments.
--gapext *spec*	Specifies gap extension penalties for global alignments.
--matrix *filename*	File name of amino acid substitution matrix in NCBI BLAST format. Default is BLOSUM62.

Alignment heuristics

Option	Description
--k *n*	Word length for alignment seeds. Default 3 for amino acids, 4 for nucleotides.
--minhsp *n*	Minimum length of HSP. Default 32.In versions up to 4.0.40, this could prevent short sequences from matching. In v4.0.41 and later, the minimum length of an HSP is automatically set to half the shorter sequence length if this is < *n*.
xdrop_u *s* **xdrop_g** *s* **xdrop_ug** *s* **xdrop_nw** *s*	X-drop parameters for extending alignments. If the value of (maximum alignment score found so far) – (current score) > X-drop, alignment extension is terminated. Smaller values are faster, but will miss more opportunities to find a higher scoring alignment by continuing to extend. With global alignments, smaller values are faster in the HSP-finding stage, but may result in slower overall times due to longer regions between HSPs that must be aligned by banding. **xdrop_u**: for ungapped local alignments, default 16.0. **xdrop_ug**: for ungapped local alignments used to trigger gapped extensions, default 16.0. **xdrop_g**: for gapped extensions of local alignments, default 32.0. **xdrop_nw**: for finding local HSPs in a global alignment, default 16.0.
--band *n*	Radius of band for banded dynamic programming, which is used to align regions between HSPs in a global alignment. Smaller values are faster, but may tend to produce less accurate alignments. Default 16.

--[no]twohit	[Do not] use two-hit word seeding. Two-hit seeding requires that two matching words are found on a single diagonal with maximum distance set by the --max2 option. With --notwohit, a single word match triggers an extension. Default is --notwohit.
--[no]nb	[Do not] use word neighborhoods for seeding alignments. Applies to amino acids only. If U-sorting is enabled, default is --nonb, otherwise the default is --nb. Using --nb is typically a little slower, but not by much and is more sensitive for low-identity matches.
--max2 *n*	Maximum distance between two word seeds on a diagonal. Ignored if --notwohit is set. Default 40.
--seedt1 *t*	Minimum score of a word seed. Used if single-hit seeding is specified (--notwohit) and word neighborhoods are enabled. Default 13.0.
--seedt2 *t*	Minimum score of a word seed. Used if word neighborhoods are enabled and two-hit seeding is used. Default 11.0.

Karlin-Altschul statistics and E-value calculation

Option	Description
--ka_dbsize	Effective database size, in letters. If the database has high redundancy, the effective size should be set to a value smaller than the actual size. Default is the actual size of the database.
--ka_gapped_k	K parameter for gapped local alignments. Default 0.041 for amino acids, 0.460 for nucleotides.
--ka_ungapped_k	K parameter for ungapped local alignments. Default 0.128 for amino acids, 0.621 for nucleotides.
--ka_gapped_lambda	Lambda parameter for gapped local alignments. Default 0.267 for amino acids, 0.128 for nucleotides.
--ka_ungapped_lambda	Lambda parameter for ungapped local alignments. Default 0.331 for amino acids, 1.330 for nucleotides.

Other options

Option	Description
--rev	Search reverse strand. Default search plus strand only. Applies to nucleotide databases only.
--[no]output_rejects	[Do not] output rejects to the --uc file. Useful for trouble-shooting cases where an expected match is not found. Default is --nooutput_rejects.
--mincodons *n*	Minimum number of amino acids in a predicted ORF. Default 20.
--usersort	Specifies that a user-defined sort order is used for the input sequences. Applies to clustering only. Default is to require that input sequences are sorted by decreasing length.
--stepwords *n*	Select a subset of words from query so that the expected number of words in common with an accepted target is *n*. Default 8. Zero means that query stepping is disabled, so all query words will be used. Larger values (or zero) tend to improve sensitivity at the expense of slower speed.
--dbstep *n*	Index every *n*th word in the database. Default is *n*=1, i.e. all words are indexed. Using this option reduces the memory required by a factor of roughly *n*. However, sensitivity tends to be reduced when clustering or searching at lower identities (say, below 80%). Using --dbstep reduces the number of processor operations required to search the in-memory database index, which might be expected to improve speed, but in practice execution times are often slower due to a reduction in cache coherence.
--bump *n*	Optimization for U-sorting, specified as integer percentage. Default 50. Zero means disabled. Larger values tend to improve speed at the expense of lower sensitivity.
--split *s*	Size of partition for --mergesort, in Mb. Default 1000.0, i.e. 1Gb.
--quiet	Do not show progress messages to standard error output while executing.
--log *logfile*	Log file name. Contains information about parameters and performance.
--version	Write program version number and exit.
--help	Write summary of command line options and exit.