OXFORD

Sequence analysis

Starcode: sequence clustering based on all-pairs search

Eduard Zorita[1,2], Pol Cuscó[1,2] and Guillaume J. Filion[1,2],*

[1]Genome Architecture, Gene Regulation, Stem Cells and Cancer Programme, Centre for Genomic Regulation (CRG), Dr. Aiguader 88, 08003 Barcelona and [2]Universitat Pompeu Fabra (UPF), 08002 Barcelona, Spain

*To whom correspondence should be addressed.
Associate Editor: Inanc Birol

Abstract

Motivation: The increasing throughput of sequencing technologies offers new applications and challenges for computational biology. In many of those applications, sequencing errors need to be corrected. This is particularly important when sequencing reads from an unknown reference such as random DNA barcodes. In this case, error correction can be done by performing a pairwise comparison of all the barcodes, which is a computationally complex problem.
Results: Here, we address this challenge and describe an exact algorithm to determine which pairs of sequences lie within a given Levenshtein distance. For error correction or redundancy reduction purposes, matched pairs are then merged into clusters of similar sequences. The efficiency of starcode is attributable to the poucet search, a novel implementation of the Needleman–Wunsch algorithm performed on the nodes of a trie. On the task of matching random barcodes, starcode outperforms sequence clustering algorithms in both speed and precision.
Availability and implementation: The C source code is available at http://github.com/gui11aume/starcode.
Contact: guillaume.filion@gmail.com

1 Introduction

All sequencing technologies have a certain degree of imprecision. For instance, the Illumina platform (Margulies *et al.*, 2005) has a 1–2% error rate consisting of substitutions (Dohm *et al.*, 2008; Nakamura *et al.*, 2011) and the PacBio platform has a 15% error rate consisting of insertions and deletions (Eid *et al.*, 2009). The enormous throughput of such technologies has recently created additional needs for developing efficient error correction algorithms.

Sequencing errors can be discovered by comparing the reads to a reference genome. However, such a reference is not always available. When the sequences are random or taken from an unknown source, clustering is the main strategy to correct the errors. For instance, this situation arises when using random barcodes to track cells or transcripts (Akhtar *et al.*, 2013; Schepers *et al.*, 2008). Sequencing errors will create erroneous (nonexistent) barcodes that have to be removed.

Sequence clustering can be viewed as a community detection problem on graphs, where nodes represent sequences and edges represent matches between related sequences. The process consists of a matching phase (the most computationally intensive), where the graph is constructed, and a clustering phase where communities are identified.

Here, we describe a sequence clustering algorithm called 'starcode' in reference to clusters of random barcodes, which typically have a star shape. Starcode is based on all-pairs search, i.e. all the pairs of sequences below a given Levenshtein distance are identified during the graph construction phase. Matching is carried out by lossless filtration, followed by an exhaustive search on the branches of a prefix trie. The novelty of the algorithm is the poucet strategy, which uses the redundancy of alphabetically sorted sequences to avoid unnecessary recomputations and gain speed.

In this article, we present and benchmark starcode. We show that on real biological datasets, starcode is orders of magnitude faster than existing sequence clustering software. Even though starcode was designed for error correction, we also show that it can be

used for other problems. As an illustration, we use it to identify enriched motifs in a bacterial genome and in protein–RNA interaction experiments.

2 Methods

2.1 Inexact string matching using tries

The matching method of starcode is based on a variation of the Needleman–Wunsch (NW) algorithm (Needleman and Wunsch, 1970). In the original algorithm (Fig. 1a), the Levenshtein distance between two sequences is found by applying a recurrence relation throughout a matrix of mn terms (the edit matrix), where m and n are the respective sequence lengths. The complexity of this dynamic programming approach is $O(mn)$.

In many instances, the information of interest is to find out whether the sequences are τ-matches (i.e. their distance is less than or equal to a fixed threshold τ). In that case, the complexity can be reduced to $O(\tau \min (m, n))$. Instead of computing all the terms of the edit matrix, it is initialized as shown on Figure 1b and only the terms around the diagonal are computed. If a diagonal term has a value greater than τ, the process is halted because the sequences are not τ-matches.

This method can be used to match sequences against a prefix tree, also known as a trie (Ukkonen, 1995). The terms of the edit matrix are updated row-wise, while a depth-first search traverses the trie (Fig. 2). Every time a node is visited, a row is computed and every time the search backtracks, a row is erased. If the threshold value τ is exceeded for a diagonal term, the Levenshtein distance for all the downstream sequences is also necessarily greater than τ. Therefore, no more hits are to be discovered in this path and the depth-first search backtracks to the parent node. When the process halts, every tail node (corresponding to a sequence of the database) on the path of this search is a τ-match of the query. This method is efficient because it eliminates large areas of the search space and

because the NW comparison of the query with each prefix of the database is computed only once.

2.2 The poucet search algorithm

The search strategy can be further improved. If two consecutive queries share a prefix of length k, the succession of computations up to the kth row of the edit matrix will be exactly the same for both queries. Therefore, computation intermediates can be stored in the nodes of the trie, so that the next trie search can start at depth k. However, storing the rows of the edit matrix in the nodes meets some difficulty. Indeed, on the kth row, the terms on the right side of the diagonal depend on characters that are not shared between the two queries. This issue is solved by storing in each node a combination of row and column terms that form an angle shape, looking like a horizontally flipped L (Fig. 3). Using this structure, the computation intermediates stored in a node at depth k depend only on the first k characters of the query.

To take full advantage of this property, the input sequences are sorted alphabetically, which maximize prefix sharing between consecutive queries. In the fairy tale 'Le Petit Poucet', the hero seeds white pebbles for his older brothers to find their way home, which is reminiscent of the way a smaller query (in alphabetical order) paves the way for the next. We therefore called this search algorithm 'poucet'.

2.3 Lossless filtration

When a query has no match, it is advantageous to omit the trie search. To this end, starcode uses a partition approach similar to that described by Wu and Manber (1992). The query is initially partitioned in $\tau + 1$ segments. Assuming that all the segments have length at least τ, then every τ-match present in the database will contain at least a verbatim copy of one of the query segments. Indeed,
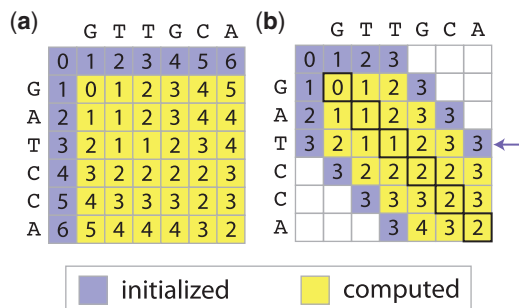

Fig. 1. NW sequence comparison. (**a**) Comparison of GTTGCA and GATCCA. The margins of the edit matrix are initialized and the cells are computed from left to right and from top to bottom by the NW dynamic programming algorithm. $E[i,j]$, the term of coordinates (i, j) is computed as $\min (E[i-1, j] + 1, E[i, j-1] + 1, E[i-1, j-1] + \Delta(i,j))$, where $\Delta(i,j) = 0$ if the ith symbol from the first sequence is the same as the jth symbol from the second and $\Delta(i,j) = 1$ otherwise. The Levenshtein distance between the two sequences is the value of the bottom right cell. (**b**) Lower complexity algorithm to determine whether GTTGCA and GATCCA are 2-matches. The values in the outer cells are set during initialization. The dynamic programming algorithm proceeds as above, with the difference that it is aborted if the value of a diagonal cell (bold borders) is larger than 2. The values in the initialized cells may differ from the original NW scheme (arrow), but the values in the computed cells are nevertheless identical. The values of the empty cells are never computed, which contributes to reducing the complexity
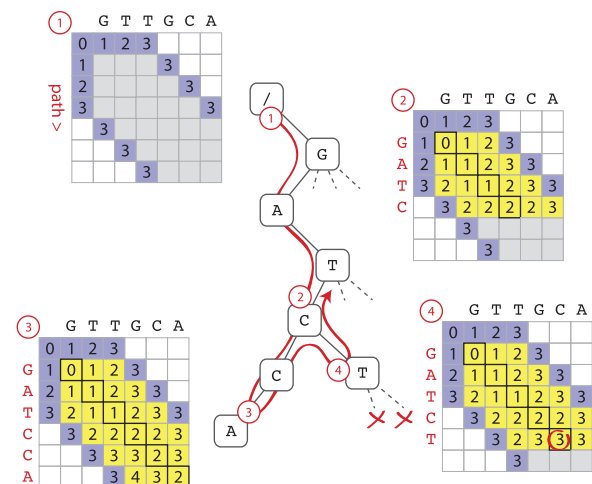
Fig. 2. NW algorithm on tries. Each sequence of the index is a path in the trie. The query GTTGCA is written at the top of the matrix, which is initialized as shown on Figure 1b. The trie is traversed by a depth-first search (path ending with an arrow). At each depth, the node added to the path is written on the left of the edit matrix and the row is computed. Checkpoints from 1 to 4 (circled numbers) show the state of the edit matrix as the search proceeds. The node labeled 3 is a leaf and thus corresponds to a 2-match of the query. After discovering the hit, the search path backtracks to the node labeled 2 and the last rows of the edit matrix are erased. The search path then goes to the node labeled 4, in which case the newly computed diagonal cell exceeds the threshold (circled). Even if this node has children, they are not visited (crosses) because there is no 2-match to discover

there are at most τ editions between the query and the match to be distributed in $\tau + 1$ regions, so at least one segment is unmodified. Because of potential insertions and deletions in the preceding segments, the shared segment may be shifted up to τ nucleotides on the left (all insertions) or on the right (all deletions) from its original position in the query.

These observations are the basis of a filtration method with 100% sensitivity. More precisely, the segments are defined as follows: the first τ nucleotides of the sequence are removed, and the rest of the sequence is partitioned in $\tau + 1$ segments of sizes differing by at most 1 (the longer segments always in 3' for consistency). Every time a sequence is added to the trie, it is partitioned and its segments are added to $\tau + 1$ different indexes. The first fragments are added to the first index, the second fragments to the second index, etc. Before the search, the query is partitioned in the same way and its segments are looked up in the indexes. In case no match is found, this query has no τ-match in the current database, therefore the trie search is omitted. Conversely, if at least one segment is found, the trie search must be performed.

As mentioned above, segments shared between the query and a τ-match may be found shifted up to τ nucleotides. For this reason, shifted segments of the query are looked up in the indexes according to the scheme of Figure 4, which ensures that no match can be missed: the rightmost segment is looked up in the $\tau + 1$st index, the second rightmost segment and the contiguous segments shifted by one nucleotide are looked up in the τth index and so on, until the first segment and its contiguous segments shifted by up to τ nucleotides are looked up in the first index.

2.4 Seek and construct

To reduce the size of the search space, starcode uses a dynamic 'seek and construct' approach whereby queries are processed, meanwhile the trie is built. In other words, each sequence is matched against the trie before it is inserted. If A and B are mutual τ-matches, either A will be queried when B is in the trie or the converse. Either way, the match A-B is discovered. This guarantees that every τ-match is discovered, while maintaining the trie as 'thin' as possible, thereby

reducing the search time. The whole matching process is summarized in the pseudocode shown in Algorithms 1 and 2.

Algorithm 1 Starcode algorithm

1: **Define:** τ
2: **Variables:** *seed*, *start* = 0, *height*, *seq*, *trie*, *lastseq*, *k*
3: **Containers:** *hits*, *pebbles*
4: READ sequence file
5: *height* ← DETERMINE maximum sequence length
6: PAD sequences up to *height*
7: SORT sequences alphabetically
8: *k* ← COMPUTE filter segment lengths
9: *trie* ← CREATE an empty trie of height *height*
10: INSERT root node of *trie* in *pebbles* at depth 0
11: **for all** sequences **do**
12:     *seq* ← GET next sequence
13:     **if** at least one *k*-mer of *seq* is in the filter index **then**
14:         *seed* ← LENGTH of shared prefix between current and next sequence
15:         *start* ← LENGTH of shared prefix between *seq* and *lastseq*
16:         CLEAR *hits*
17:         CLEAR *pebbles* at depth > *start*
18:         **for all** *pebbles* at depth *start* **do**
19:             *node* ← GET next node from *pebbles*
20:             **call** POUCET(*seq*, *node*, *seed*, *hits*, *pebbles*)
21:         **end for**
22:         PROCESS *hits* and LINK matches to *seq*
23:         *lastseq* ← *seq*
24:     **end if**
25:     INSERT *seq* path in *trie*
26:     INSERT *seq* *k*-mers into the filter index
27: **end for**

Algorithm 2 Poucet search algorithm

1: **procedure** POUCET(*query*, *node*, *seed*, *hits*, *pebbles*):
2:     COMPUTE *node*-specific column following NW ◁ Figure 1
3:     **for all** *child* nodes in *node* **do**
4:         COMPUTE *child*-specific row following NW ◁ Figure 1
5:         COMPUTE center value using row and column ◁ Figure 1
6:         **if** center value > τ **then** ◁ Mismatches exceeded.
7:             **continue** with next *child*
8:         **end if**
9:         **if** *node* depth = *height* **then** ◁ Hit found.
10:            SAVE *node* sequence in *hits*
11:            **continue** with next *child*
12:        **end if**
13:        **if** *node* depth ≤ *seed* **then**
14:            SAVE *node* in *pebbles* at current depth
15:        **end if**
16:        **call** poucet(*query*, *child*, *seed*, *hits*, *pebbles*)
17:    **end for**
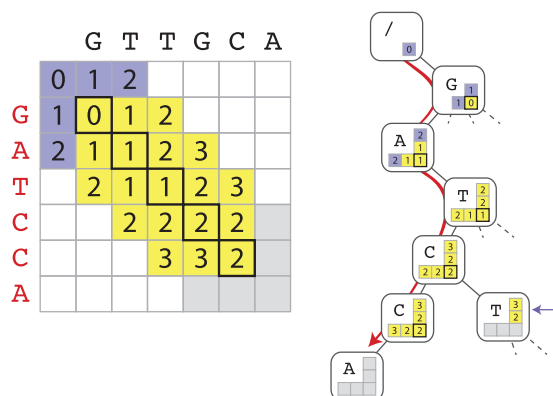18: **end procedure**

Fig. 3. Poucet search algorithm. The algorithm proceeds with the same principles as shown on Figure 2 with the difference that the edit matrix is not updated row-wise but along a horizontally flipped L. As the depth-first search proceeds, these values are stored in the nodes of the trie. As the values in the vertical part of the flipped L are the same for every child of a node, they are computed only once (arrow). The values in the gray cells will be computed as the search path visits the node. Storing the intermediates in the nodes allows the next query to restart at depth *k* if it shares a common prefix of length *k* with the current query

2.5 Parallelization

Queries are sorted and partitioned in contiguous blocks. The matching step then proceeds in two phases. In the build phase, a distinct trie is built from the sequences of each block according to the algorithm described above. In the second, all the sequence blocks are queried against all the other tries. If the queries are partitioned in N blocks, the first phase consists of N seek and construct jobs, whereas
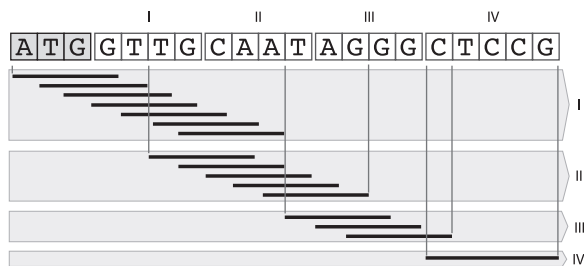
Fig. 4. Lossless filtration illustrated by an example sequence of length 20 with $\tau = 3$. The last τ nucleotides of the query are removed and the rest is divided into four series of contiguous segments. Each series is queried against a different index numbered I–IV. For instance, the only segment queried against index I is GTTG, whereas those queried against index II are GCAA, CAAT and AATA. If any of the segments is found in the appropriate index, the trie search is performed, otherwise it is omitted as there can be no τ-match. Regardless of the result, segments labeled I–IV are then added to the corresponding respective index (i.e. only one segment is added to each index)

Table 1. Software execution options used in simulation benchmark

Software	Command-line options
Starcode-1.0	`starcode -d3`
Slidesort-2	`slidesort_v2 -d 3 -t E -c DNA`
Cd-hit-est-4.6.1	`cd-hit-est -n 9 -c 0.9 -M 0 -r 0`
Seed-1.4.1	`SEED –mismatch 3`

Table 2. Software execution options used in real data benchmark

Software	Command-line options
Starcode-1.0	`starcode -d3`
Slidesort-2	`slidesort_v2 -d 3 -u -t E -c DNA`
Cd-hit-est-4.6.1	`cd-hit-est -n 8 -c 0.94 -M 0`
Seed-1.4.1	`SEED –mismatch 3 –shift 3`
Rainbow-2.0.3	`rainbow cluster -m 3`

the second consists of $N(N − 1)/2$ query jobs. In each phase, the jobs show no dependency on each other, so the matching algorithm can be efficiently parallelized provided N is larger than the number of independent threads.

2.6 Clustering

The default clustering algorithm of starcode is designed to correct sequencing error. This method uses message passing (MacKay, 2002) to identify and count 'canonical' sequences (also referred to as centroids in the clustering terminology). By default, each sequence transfers its read count to its closest τ-match provided the latter has at least five times more counts. If the condition is not met, the transfer does not take place. If the sequence has several equally close τ-matches, the counts are split equally among them. The process is repeated recursively, starting from sequences with lowest read count. The sequences with a positive read count at the end of the process are considered canonical. Clusters consist of all the sequences transferring their read counts to the same canonical sequence (sequence transferring their read counts to different canonicals are discarded). Note that the radius of the clusters can be higher than the maximum distance used for matching.

As no sequencing technology has an error rate higher than 20%, it is expected that sequences appearing from sequencing errors will always have five times or lower read count than the canonical sequence. Otherwise, sequences are more likely unrelated or both are derived from the same canonical sequence. This behavior can be modified with the command-line option *cluster-ratio* to allow for a more flexible or more strict clustering, e.g. to cluster unique input sequences together, *cluster-ratio* must be set to 1.

For other sequence clustering problems, starcode implements a multi-purpose algorithm called 'sphere clustering'. In sphere clustering, sequences are sorted by frequency of occurrence. Starting from the most frequent, each sequence becomes canonical and claims all its τ-matches, which forms a cluster of radius τ (hence the name). Claimed sequences are immediately removed, so that they can belong to only one cluster.

2.7 Benchmark conditions

All the tests were performed on a 16-core dual-processor Intel Xeon E5-2687W v2 system with 256 GB of DDR3-RAM at 1866 Mhz. Command-line parameters were set equivalently in all softwares to run in single-core mode allowing up to three mismatches for input

sequences of length 50. Tables 1 and 2 summarize the execution options used in simulation and real datasets, respectively.

3 Results

3.1 Presentation and basic performance

Starcode is a general purpose DNA sequence clustering tool with a strong focus on error correction. Errors are assumed to be mismatches, insertions or deletions (the implementation presented here matches sequences with up to eight errors). The input sequences can be single or paired-end reads, with an upper limit of 1024 nucleotides (512 for paired end). Sequences may be of variable length, they may be trimmed and filtered for quality or not. File formats compatible with starcode are raw sequence, raw sequence with count, FASTA or FASTQ (in which case starcode ignores the quality). Starcode either returns detailed information of the clustering results, i.e. canonical sequences, cluster sizes and the complete list of their constituent sequences. Alternatively, only the canonical sequences are printed, which is useful to filter out redundant sequences from input files. By default, clustering is performed under the assumption that divergence occurs from experimental errors (sequencing errors, polymerase chain reaction mutations, etc.) and a more general algorithm is also available for other clustering problems (an example of which is given in Section 3.3).

We show the basic performance and scalability of starcode on a dataset of pseudorandom sequences (Fig. 5). The standard configuration is a set of 1 000 000 sequences of length 40, running on one thread and with a maximum Levenshtein distance of 3. In each test, only one parameter is modified, while the others are kept constant. As the clustering step does not require additional memory allocation and is significantly faster than all-pairs search, the performance results presented in Sections 3.1 and 3.2 apply for both message-passing and spheres clustering algorithms.

Figure 5a shows the running time of starcode as a function of the number of input sequences n. In double logarithmic scale, the trend is a straight line with slope 1.5, suggesting that the running time complexity of starcode is lower than quadratic (the naive implementation of all-pairs search). Note that the sequences of this dataset have no match, see Section 3.2 for an evaluation of the performance on more realistic datasets. Figure 5b shows that the running time grows exponentially as a function of the maximum Levenshtein distance used for clustering. The reason is that the trie fans out
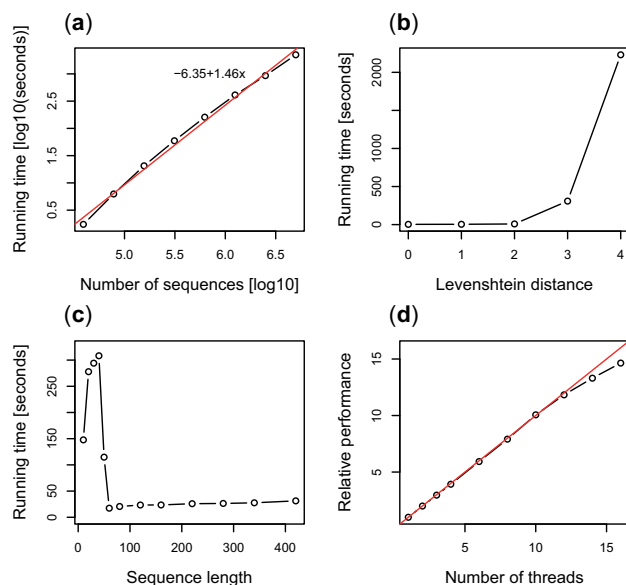
Fig. 5. Scalability. (**a**) Logarithm of the running time versus the logarithm of the number of sequences to be clustered. (**b**) Running time as a function of the clustering distance. (**c**) Running time versus length of the input sequences. (**d**) Relative performance increase for different number of parallel threads

Fig. 6. Benchmark results on artificial datasets of known cluster structure (see main text). (**a**) Accuracy measured by the number of identified clusters. Starcode identifies the correct number of clusters, whereas seed and cd-hit identify about 40 false positives per true positive. The first bisector is plotted and indicates perfect results. (**b**) Accuracy measured by the number of identified pairs. Slidesort identifies 5–10% less pairs than starcode. The horizontal line indicates a ratio of 1. (**c**) Running time of the different tools. As the size of the clusters, the running time of starcode increases but remains competitive. (**d**) Memory usage of the different tools. The memory usage of starcode decreases as the size of the cluster increases.

exponentially and the search bails out at a greater depth as the maximum distance increases. As a function of the sequence length, the running time first increases but then plummets and stays low (Fig. 5c). Beyond a threshold length, the filtering algorithm starts to be efficient, and most of the queries are resolved without searching the trie. Finally, we show the scalability of starcode with increasing number of threads in Figure 5d. The search algorithm is fully parallel and the relative performance increases linearly up to 12 threads. The bending observed thereafter has two sources; the first is that the input reading and clustering steps are brief but not parallel, the second is due to hardware limitations, i.e. there is insufficient memory bandwidth to satisfy the increased demand of parallel memory accesses.

3.2 Benchmark

We benchmarked starcode against the sequence clustering algorithms slidesort (Shimizu and Tsuda, 2011), seed (Bao *et al.*, 2011), rainbow (Chong *et al.*, 2012) and cd-hit (Fu *et al.*, 2012). Even though slidesort is an all-pairs search algorithm, it was included in the benchmark because sequence comparison is the most computationally intensive step of the sequence clustering problem. Rainbow runs exclusively on paired-end reads, whereas the other tools run on single reads, for this reason all the tools could not be run on the same dataset.

The performance of sequence clustering algorithms can be sensitive to the size of the clusters in the dataset, which in many applications is not known a priori. We therefore set up a benchmark on artificial datasets to test the accuracy and the scaling of the tools on a known cluster structure. We generated four datasets of 1 million 50-mers arranged in 1 to 1000 clusters. Each cluster consisted of 100 repeats of the same centroid sequence, plus satellites derived from the centroid by incorporating three errors including at most one indel. The number of satellites per cluster ranged from 999 900 to 900. Rainbow was not tested on this benchmark because the generated data are single reads. In addition, evaluating the exactness of slidesort was problematic because the number of 3-matches in each
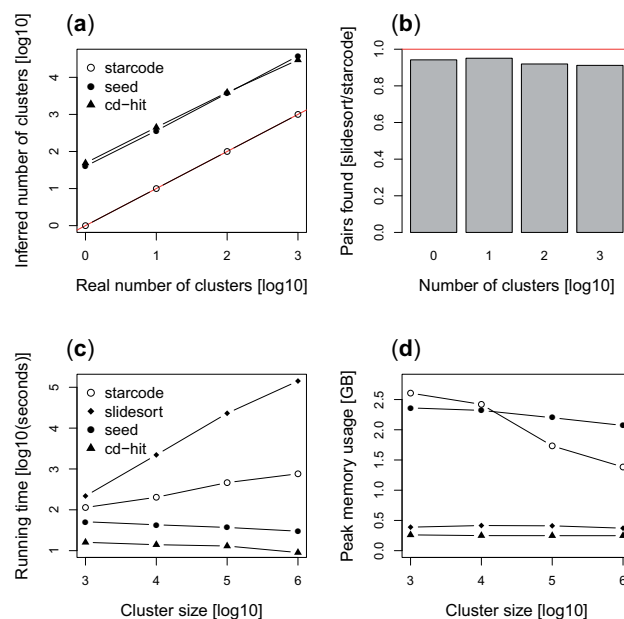
dataset is not known (pairs of satellites in the same cluster may be 3-matches or not). For this reason, we only compared the number of pairs found by starcode with the number of pairs found by slidesort. The outcome of the test is summarized in Figure 6.

Although starcode achieves perfect clustering on all four datasets, the clustering achieved by seed and cd-hit is incomplete. Both tools identify approximately 40 false clusters per true cluster on all the datasets (Fig. 6a). We also observed that slidesort found 5–10% less 3-matches than starcode on all the datasets (Fig. 6b). We were surprised by this result because slidesort is claimed to be an exact algorithm. However, this was clearly not the case when we ran additional tests on smaller datasets where naive pairwise comparisons are feasible (more information is available on the starcode repository, see http://github.com/gui11aume/starcode/tree/master/misc).

The running time of the different tools as a function of the size of the clusters is shown on Figure 6c. The running time of slidesort and starcode show linear and sub-linear trends, respectively. Seed and cd-hit run approximately in constant time regardless of the cluster size. In spite of this result, the performance of starcode remains competitive, even for clusters of 1 million sequences. The memory usage is shown in Figure 6c. The smallest memory footprint is achieved by slidesort and cd-hit, with a maximum difference of an order of magnitude with respect to the other tools. Note that the comparison with slidesort is not completely fair since it does not hold in memory the full graph necessary for clustering. The memory usage of starcode is the highest for clusters of size 1000, but it decreases and becomes lower than the memory usage of seed as the size of the cluster increases. In conclusion, starcode was the only tool to achieve perfect precision on these datasets at a price of increased memory footprint. Considering the exactness of the

Table 3. Summary of the biological datasets used for benchmarking

Dataset	Read count	Read length	Type
SRR950457	6 542 309	16 ± 1	Single
PRJEB7686	127 675 537	50	Single
SRR950477	2 460 226	100 + 100	Paired end

All the datasets are Illumina reads.

Table 4. Running time (in seconds) of the software on three biological datasets

Software	SRR950457	PRJEB7686	SRR950477
Starcode	5	2898	44
Seed	—	60 374	—
Slidesort	4055	>10 days	—
Rainbow	—	—	306
Cd-hit-est	170	512 591	—

Exact running time was not recorded past 10 days. A dash indicates that the software cannot be used for this dataset.

Table 5. Memory usage (in GB)

Software	SRR950457	PRJEB7686	SRR950477
Starcode	0.65	30.9	5.2
Seed	—	53.9	—
Slidesort	1.30	13.9	—
Rainbow	—	—	0.5
Cd-hit-est	0.80	28.5	—

output, starcode maintains a competitive performance in terms of running time.

The performance on artificial data is not always in agreement with the performance on experimental datasets. Typical experiments present additional difficulties. For instance, the sizes of the clusters may be uneven and the reads may contain near constant regions that usually degrade the performance of filter-based algorithms. We benchmarked sequence clustering algorithms on the problem of clustering thousands of reporters integrated in parallel (TRIP) barcodes (Akhtar *et al.*, 2013). Briefly, the principle of TRIP is to tag reporter transcripts with random barcodes and measure the abundance of barcodes in the RNA as a proxy for gene expression. There is no reference to match aberrant barcodes against, because the tagging sequences are unknown.

The basic properties of the datasets used for benchmarking are summarized in Table 3. Dataset 1 (SRR950457) has been pre-processed to extract the barcode and remove the constant part of the reads. Only barcodes between 15 and 17 nucleotides were included in the file. Dataset 2 (PRJEB7686) consists of raw Illumina single reads. These datasets differ by the read size, the total read count and the empirical cluster sizes. According to the output of starcode, the largest clusters of dataset 1 contain approximately 70 000 sequences, whereas dataset 2 contains four clusters with more than 1 million sequences. Dataset 3 (SRR950477) has been included to benchmark starcode against rainbow in paired-end clustering mode.

The running times of starcode, seed, slidesort, rainbow and cd-hit are summarized in Table 4. We accommodated the distance threshold for the first dataset to compensate for the reduced sequence length. Both starcode and slidesort were executed with the option '-d 2' and the identity for cd-hit was set to '-c 0.85'. We were not able to run seed on dataset 1 due to limitations on the minimum sequence length. Starcode was significantly faster than the other tools on all the datasets. Seed and cd-hit came in second position with a running time approximately 35 and 20 times greater on datasets 1 and 2, respectively. Rainbow was nearly an order of magnitude slower in the job of clustering paired-end reads. We did not record the exact running times past 10 days since this is several orders of magnitude higher than the running time of starcode.

The memory footprint of the different tools on the same datasets is presented in Table 5. The values represent the peak memory usage throughout the run on the datasets described above. On short reads (dataset 1), starcode outperforms the other tools taking advantage of the trie compaction. On dataset 3, starcode had a significantly larger memory usage than rainbow. Starcode and cd-hit used similar amount of memory on dataset 2. Both needed twice as much memory as slidesort, which has the advantage of not storing the complete graph during the all-pair comparison.

3.3 Identifying enriched sequence motifs

As a sequence clustering algorithm, starcode can also be used for other applications, such as the identification of enriched motifs.

Sequence motifs are thought to play an important role in DNA metabolism. Key regulators, such as transcription factors, nucleosomes and non coding RNAs have sequence preferences targeting them to the sites where they act. Identifying those sequences is a way to pinpoint the regulators and the mechanisms they are involved in. However, the sequence motifs are not strictly identical at different sites, hence they are better identified by inexact matching. This problem becomes computationally difficult for long motifs (above 12–13 nucleotides) because of the combinatorial scaling.

We set up a test based on the meningitis-causing agent *Neisseria meningitidis*. The genome of this bacterium is interspersed with a frequent 12 bp sequence known as DNA uptake sequence (Smith *et al.*, 1999). We extracted the 12-mers from both orientations of the 2.19 Mb genome, yielding 4.39 million 12-mers, consisting of 2.77 million unique sequences. Clustering the 12-mers with starcode within a Levenshtein distance of 2, we identified the known DNA uptake sequence of *N.meningitidis* (ATGCCGTCTGAA) as the most abundant 12-mer, with 1466 exact and 2096 inexact hits. This result testifies to the fact that starcode can be used to identify biologically relevant motifs in bacterial genomes.

To test starcode on another application, we used the RNA–protein interaction data produced by RNAcompete (Ray *et al.*, 2009). The mammalian splicing factor SRSF1 is known to bind RNA GA-rich motifs, but there is some disagreement about the motif that it recognizes (Pandit *et al.*, 2013). For each replicate of the human SRSF1 in the RNAcompete dataset, we replaced the microarray signals by their rank and extracted the 10-mers from the microarray probes. The 10-mers were given a score equal to the rank of the probe they belong, and enriched motifs were found using the sphere clustering of starcode with maximum Levenshtein distance 2. The score of the most enriched 10-mer is thus the sum of the ranks of all 10-mers within this distance. Among the six replicates, the most enriched 10-mers were AGGACACGGA, AGGACACGGA, AGGACGGAGG, AGGACGGAGG, AGGAC ACGGA and AGGATACAGG. Except for the last replicate, the motifs consist of AGGAC and GGA, with a spacer of variable length. This suggests that the binding of SRSF1 to RNA may involve a spacer sequence, which would explain the disagreement between the motifs derived from 6-mers or 7-mers.

4 Discussion and conclusion

Starcode is a solid algorithm for sequence clustering based on all-pairs matching. It achieves high precision, and on experimental datasets it can be faster than popular heuristics. By design, starcode is tailored to process high throughput sequencing data on multi-core platforms with sufficient amount of memory. Because of its superior precision and faster running time, it fills a gap among available software, by allowing to take full advantage of middle to high end hardware.

It is somewhat surprising that starcode is significantly faster than competing tools on experimental datasets, whereas seed and cd-hit are faster on artificial datasets. Starcode was developed ground up from TRIP experimental datasets and the poucet search was selected for giving the best empirical results. We speculate that the trie structure benefits from the entropy deficit that is typically observed in experimental data versus pseudorandom reads.

The speed and precision of starcode also makes it useful for other clustering tasks, such as identifying enriched motifs in microbial genomes and in experimental data. Here, we have given two examples of such applications. In the first, we recover a known enriched 12-mer in the genome of *N.meningitidis*. In the second, we recover the motif of the human RNA binding protein SRSF1 and notice that it seems to consist of two halves separated by a linker. This hypothesis is consistent with the fact that SRSF1 binds RNA through two consecutive RNA-recognition motifs that are known to bind 3–4 nucleotides in a row (Daubner *et al.*, 2013). The Levenshtein distance, which incorporates insertions and deletions, is more likely to capture bi-partite binding motifs than position weight matrix representations. The use of a clustering method to tackle this problem is unusual, but it illustrates the potential advantages of distance-based approaches.

One of the reasons why starcode appears to be faster than alternative tools is that it is designed to cluster relatively similar sequences. When clustering related sequences, the Levenshtein distance will have to be increased, leading to exponentially longer running times (Fig. 5b). However, for the important practical case of correcting errors introduced by sequencing, starcode illustrates that there is still room for developing algorithms that are both faster and more accurate than the current state of the art.

Acknowledgements

We would like to thank Maria Chatzou for her precious feedback on the preliminary version of this manuscript and Heng-Chang Chen for performing the *Drosophila* TRIP experiments.

References

Akhtar,W. *et al.* (2013). Chromatin position effects assayed by thousands of reporters integrated in parallel. *Cell,* **154**, 914–927.

Bao,E. *et al.* (2011). SEED: efficient clustering of next-generation sequences. *Bioinformatics,* **27**, 2502–2509.

Chong,Z. *et al.* (2012). Rainbow: an integrated tool for efficient clustering and assembling RAD-seq reads. *Bioinformatics,* **28**, 2732–2737.

Daubner,G.M. *et al.* (2013). RRM-RNA recognition: NMR or crystallography and new findings. *Curr. Opin. Struct. Biol.,* **23**, 100–108.

Dohm,J.C. *et al.* (2008). Substantial biases in ultra-short read data sets from high-throughput DNA sequencing. *Nucleic Acids Res.,* **36**, e105.

Eid,J. *et al.* (2009). Real-time DNA sequencing from single polymerase molecules. *Science,* **323**, 133–138.

Fu,L. *et al.* (2012). CD-HIT: accelerated for clustering the next-generation sequencing data. *Bioinformatics,* **28**, 3150–3152.

MacKay,D.J.C. (2002). *Information Theory, Inference & Learning Algorithms.* Cambridge University Press, New York.

Margulies,M. *et al.* (2005). Genome sequencing in microfabricated high-density picolitre reactors. *Nature,* **437**, 376–380.

Nakamura,K. *et al.* (2011). Sequence-specific error profile of Illumina sequencers. *Nucleic Acids Res.,* **39**, e90.

Needleman,S.B. and Wunsch,C.D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.,* **48**, 443–453.

Pandit,S. *et al.* (2013). Genome-wide analysis reveals SR protein cooperation and competition in regulated splicing. *Mol. Cell,* **50**, 223–235.

Ray,D. *et al.* (2009). Rapid and systematic analysis of the RNA recognition specificities of RNA-binding proteins. *Nat. Biotechnol.,* **27**, 667–670.

Schepers,K. *et al.* (2008). Dissecting T cell lineage relationships by cellular barcoding. *J. Exp. Med.,* **205**, 2309–2318.

Shimizu,K. and Tsuda,K. (2011). SlideSort: all pairs similarity search for short reads. *Bioinformatics,* **27**, 464–470.

Smith,H.O. *et al.* (1999). DNA uptake signal sequences in naturally transformable bacteria. *Res. Microbiol.,* **150**, 603–616.

Ukkonen,E. (1995). On-line construction of suffix trees. *Algorithmica,* **14**, 249–260.

Wu,S. and Manber,U. (1992). Fast text searching: allowing errors. *Commun. ACM,* **35**, 83–91.