# Fast Sequence Clustering Using a Suffix Array Algorithm

Ketil Malde*, Eivind Coward, and Inge Jonassen
Department of Informatics, University of Bergen, HIB, N5020 Norway

January 13, 2003

*To whom correspondence should be addressed

# Abstract

**Motivation:** Efficient clustering is important for handling the large amount of available EST sequences. Most contemporary methods are based on some kind of all-against-all comparison, resulting in a quadratic time complexity. A different approach is needed to keep up with the rapid growth of EST data.
**Results:** A new, fast EST clustering algorithm is presented. Sub-quadratic time complexity is achieved by using an algorithm based on suffix arrays. A prototype implementation has been developed and run on a benchmark data set. The produced clusterings are validated by comparing them to clusterings produced by other methods, and the results are quite promising.
**Availability:** The source code for the prototype implementation is available under a GPL license from
http://www.ii.uib.no/~ketil/bio/
**Contact:** `ketil@ii.uib.no`


# Introduction

Expressed sequence tags (ESTs) constitute a valuable and rapidly growing resource for different kinds of gene analysis; for instance identification of genes, analysis of gene structure (including alternative splicing), and gene expression analysis. For human alone, there are now more than four million EST sequences available.

Because of high redundancy, low quality, and short sequences, clustering of related EST sequences is important in order to extract useful information efficiently. One natural goal for clustering is to group all ESTs originating from the same gene (including possible splice variants). This is the goal of e.g. the UniGene clustering (Boguski and Schuler, 1995). Each cluster can then be represented by one or more consensus sequences, assembled from the ESTs (Haas et al., 2000).

Commonly employed methods for sequence clustering (e.g. Holm and Sander, 1998; Burke et al. 1999; Liang et al., 2000) is based on pairwise comparison of sequences, resulting in a similarity score. This score can either be calculated using standard software like BLAST (Altschul et al., 1990) or FASTA (Lipman and Pearson, 1988), or use other more specialized algorithms like d2_cluster's word-based approach.

In general, a transitive closure clustering based on a similarity score can avoid doing an explicit all-against-all comparison, but will in the worst case still need to calculate the similarity between all sequence pairs. Thus, clustering based on pairwise similarity imposes $O(m^2)$ time complexity (where $m$ is the number of sequences to be clustered).

The *UIcluster* program (Pedretti, 2001) uses such a technique, maintaining a global lookup table for known substrings to quickly eliminate many of the comparisons. Additionally it avoids many of the comparisons by comparing sequences only to a representative sequence from each of the existing clusters.

Quadratic scalability can be acceptable for small numbers of sequences, but as databases available contain millions of ESTs, implying on the order of $10^{12}$ sequence comparisons, faster algorithms are desirable.

The algorithm described in this paper uses a different approach, inspired by suffix trees. Suffix trees, first described by Weiner (1973), allow all occurrences of identical substrings to be identified in $O(n)$ time. This gives us a very quick way to find all *exact* matches between sequences, and we use this as a starting point for EST clustering with sub-quadratic behavior.

# Terminology

A *suffix array* for a set of strings is a lexicographically ordered array of all suffixes of the strings (Manber and Myers, 1993).

Two sequences have a *matching block* if they have contiguous segments that match exactly. A matching block is *maximal* for a pair of sequences if the sequences differ immediately beyond each end point of the block.

We define a parameter, $k$, that specifies the length of the shortest matching blocks that the algorithm will detect. A *k-clique* is the set of all sequences that have a specific matching block of length $k$.

A *score* of a pair of sequences is a numeric measure of their similarity.

Unless otherwise specified, we will in the following define the score of a set of matching blocks to be the sum of the block lengths, and the score of a pair of sequences to be the score of the highest scoring consistent set of matching blocks, where a consistent set is a subset of blocks that are non-overlapping and co-linear (i.e. appearing in the same order) in the sequences.

In order to perform the actual clustering, it is necessary to select the minimal score that will cause two sequences to be clustered together. This value will be referred to as the *clustering threshold.*

We let $m$ denote the number of sequences, and $n$ the total number of nucleotides (i.e. average sequence length is $n/m$).

# Algorithm

The whole algorithm is divided into three parts. The first part of the algorithm identifies the pairs that have matching blocks. The second part uses the information

generated from this process to calculate a score for pairs of sequences, and finally these scores form the basis for a hierarchical clustering.

1. Identify all matching blocks of length $k$:
   (a) Construct all suffixes from the data
   (b) Sort the suffixes into a suffix array
   (c) Group the suffixes that share a prefix of length at least $k$ into cliques
   (d) For each clique, generate the maximal matching blocks between each pair of suffixes in the clique
2. Score the resulting sequence pairs:
   (a) For each pair of sequences sharing at least one matching block, collect all matching blocks between the two sequences
   (b) Calculate the largest consistent set of matching blocks, and the corresponding score for each pair
3. Generate the clustering:
   (a) Starting with the highest scoring sequence pair and working downward, build clusters hierarchically by connecting sequences
   (b) Split the clusters according to the clustering threshold

## Identifying high-scoring pairs

While there exist general data clustering methods that take into account the similarities between multiple objects in order to determine whether to join clusters, single linkage clustering is the most useful approach for clustering of sequence fragments.

The underlying idea for the algorithm is that, while a full similarity matrix contains $O(m^2)$ scores, less than $m$ scores are actually used in single linkage clustering. Thus, one way to achieve significant improvement over algorithms based on a similarity matrix, is to avoid performing the low scoring comparisons.

We observe that since a suffix array is ordered, every clique constitutes a contiguous section of it. Since all sequences that share a matching block of length $k$ are grouped in a $k$-clique, we can, from each clique in turn, obtain all sequence pairs that share each $k$-block.

Suffix array generation uses a straightforward, left-wise radix sort, the $i$th pass sorting the suffixes on the $i$th nucleotide. While the array could conceivably have been built using a suffix tree algorithm (Weiner, 1973; McCreight, 1976; Ukkonen, 1995), the sorting algorithm is conceptually very simple, and performs well in the expected case. Also, while algorithms for suffix tree construction have good asymptotic behavior, they tend to consume a lot of memory, with poor locality characteristics (Giegerich and Kurtz, 1995).

By first glance, our algorithm hardly does any better, since it produces large intermediate data structures at each level in the recursive sorting algorithm. However, we can get the same result by performing the algorithm on subsets of the data: Selecting a prefix length $l \leq k$, we can for all nucleotide words of length $l$ extract from the data all suffixes that have this word as a prefix. These suffixes can then be sorted into a suffix array, cliques identified, and pairs generated. We retain the pairs, but have no further use for the suffixes in the suffix array, and its space can be reclaimed before generating the suffix array for the next prefix.

While the memory footprint of suffix array generation thus is reduced by a factor of $4^l$, up to a maximum of $4^k$, the trade-off is increased time to perform the multiple passes over data. The choice for prefix length that is optimal in terms of running time, is thus the smallest value that allows the entire program to execute in available RAM.

The sorting algorithm is also modified to keep track of the match length between subsequent suffixes in the array; this is simply the depth where the sorting terminates (i.e. when there is only a single suffix to sort, or where suffixes have been compared to their full length).

To reduce the number of pairs that are generated, and to simplify book-keeping, we ignore matches that are not maximal by checking that the sequences differ in the preceding nucleotide — if not, the pair is redundant anyway, since the maximal match must be present in a clique elsewhere. Collecting and collapsing multiple matches is achieved by sorting the list of pairs with respect to the ESTs referenced.

## Scoring and cluster construction

Sequence scores can be calculated with tools like BLAST, FASTA, or with the d2 algorithm. The present algorithm instead constructs the largest consistent set from the matching blocks already identified, and uses the sum of the block lengths as the score. This calculation is performed efficiently using a dynamic programming algorithm.

Clustering of the scored pairs starts with an empty set of clusters. It takes at each step the highest scoring sequence pair, and checks it against the current clusters to see whether the ESTs are already in the same cluster. If not, we have three possibilities: If the two existing clusters have one each of the sequences, the clusters are merged. If only one of the sequences are clustered, the

other sequence is added as a leaf to that cluster. Finally, if none of the sequences are in any clusters, a new cluster is generated with the sequences as leaves. The pair is then removed, and the process is repeated for the next-highest scoring pair, and so on.

The result is a set of clusters, where each cluster is a binary tree, with the sequences as leaves. Each branching node represents a sub-cluster less strongly connected (since it is generated by a lower scoring sequence pair), than sub-clusters below.

# Analysis

Generating the suffix array can be implemented in linear time and space using a suffix tree. However, our sorting approach turns out to be no different from constructing a suffix tree using the *lazytree* algorithm by Giegerich and Kurtz (1995) while collapsing the resulting tree. Thus, the efficiency of such a direct sort should be quite good in practice, and run in expected $O(n \log n)$ time, although the worst-case behavior is quadratic.

Generating the suffix array in parts by prefix changes the complexity slightly. With a prefix length of $l$, the data set is traversed $4^l$ times, and in each iteration the suffix array for $1/4^l$th of the data set is constructed. However, the depth of the tree does not change, so the resulting time complexity becomes $O(4^l n + n \log n)$, and the space complexity $O(n/4^l)$.

The algorithm for generating pairs from a clique is quadratic in the clique size. This is potentially a costly step, there is a trade off between sensitivity and performance; with $k$ too small, the cliques will grow large and running time will be impacted, with $k$ too large, we may miss some matches, which again may be necessary to produce the correct clustering.

Score calculation will be performed on all generated pairs, and thus for $p$ pairs, the complexity will have a factor $p$. Different algorithms defining the score metric can have different time complexities; the one in the current implementation uses a dynamic programming algorithm that runs in time linear in the number of blocks as long as the blocks are non-overlapping. In order to handle overlapping blocks, the worst case bounds becomes $O(b^2)$ for $b$ matching blocks, but as the worst case is rare, and the number of blocks matching in two sequences is rarely large for reasonable values of $k$, scoring is essentially linear in $p$.

When a pair is to be added to the clustering, the ESTs must be looked up in the existing clusters. Consequently, each cluster stores the ESTs referenced in a searchable structure, which typically has a lookup cost logarithmic in the number of values stored. For $p$ pairs and $c$ clusters, the cost thus becomes $O(pc \log p)$, and, as the clustering stores each sequence exactly once, the space complexity is linear in the number of sequences.

# Results

The algorithm was implemented in Haskell[1], a high level functional language. The program reads ESTs in FASTA format and produces the clustering as a list of identifiers (usually UniGene accession numbers).

The program was run on a PC equipped with a 1266 MHz Intel Xeon processor and 1 GB RAM, using a benchmark dataset available from SANBI[2]. The data set contained 10000 sequences from human eye tissue, and was masked for repeats and vector sequences using cross_match.

## Efficiency

Time consumption is almost solely dependent on the choice of matching block lengths, $k$. As shown in Figure 1, the speed is approximately constant at around three minutes for block lengths over 20, indicating that pair generation no longer is a significant factor. For block lengths of less than 14, the running time becomes impractical.

Fig 1

For comparison, an all-against-all BLAST search takes 19 minutes, while d2_cluster uses about 20 minutes on the same dataset on an 866MHz CPU (Groenewald, pers. comm.).

In order to measure scalability, the program was run on data set consisting of 1250, 2500, and 5000 of the sequences, in addition to all 10000. Timing for the dominating parts, the suffix array construction and suffix extraction (including data reading) is shown in Figure 2. While suffix extraction is almost perfectly linear, suffix array construction is slightly worse than linear.

Fig 2

Running UIcluster on the benchmark set completed in less than 30 seconds, which is quite impressive. When measuring scalability as described in above, it follows a quadratic curve closely; it may scale better for data sets that result in fewer clusters, however.

## Sensitivity and specificity in pair detection

We note that for the algorithm to be able to produce a particular (e.g. the "correct") clustering, we need to generate enough pairs that the correct clusters can be constructed — in other words, there must exist a chain of sequences that match between any pair of sequences within a cluster. More formally stated, for each pair

[1] http://www.haskell.org
[2] http://www.sanbi.ac.za/benchmarks/

$(x, y)$ of sequences that should be clustered, there must exist a set of pairs $(x_i, y_i)$ for $i = 1, 2..s$, so that $x_1 = x$, $y_s = y$ and $x_{i+1} = y_i$ in the pair generation stage.

Note that this is a necessary, but not sufficient condition; there may be additional pairs generated that span clusters, which again may result in "over-clustering".

Similarly, whether the detected pairs are used in the final clustering will depend on their final score. Thus, even if the set of detected pairs is sufficient, the resulting clustering may differ from the correct one.

As a way of measuring sensitivity, we can see whether enough pairs are produced to connect the sequences within each cluster, and as a specificity measure, we can measure the fraction of pairs produced that do not cross cluster boundaries.

Fig 3

In Figure 3, we see that while the number of pairs generated grows faster than exponentially as block size decreases, the number of "true positives", i.e. pairs that are internal to a BLAST or d2-based cluster, grows much more slowly.

Another interesting property is the relative redundancy of matches, represented by the distance between the two upper graphs. At a block size of 40, we have about twice as many matching blocks as sequence pairs (i.e. each sequence pair has an average of two distinct blocks that match). As the block size decrease, the average number of matching blocks per sequence pair increases as well, until the number of distinct sequence pairs start to rise sharply (at a block length of about 16). This could be interpreted as a measure of quality; a high number of blocks per sequence pair means that the pairs represent real similarities, rather than being due to random matches.

This view is reinforced when taking into account the fraction of detected sequence pairs being inside reference clusters. The "true" pairs make up a large portion of the detected matches until block sizes decrease below 20 or so, at which point the fraction quickly diminishes.

While we have a measure of "true positives" detected by our algorithm, a corresponding measure of "false negatives", i.e. pairs that are similar but fail to be detected, is harder to calculate. The clusterings contain information about which sequences are clustered together, but not information about which pairs of sequences that match. Thus, we cannot compare the detected pairs directly, and comparing the resulting clusters (see the next section) is at any rate a more relevant measure for determining clustering quality.

We can, however, get an indication on the sensitivity of the pair detection by examining at which block lengths we no longer have enough matches to connect all reference clusters. Experiments show that for the most aggressive choice of parameters tested (block size 12, cluster threshold 32), there are no clusters from neither the d2 nor the BLAST-based clusterings that are split up. Indeed, this holds true for choices of block size up to 16.

## Clustering quality

The Jaccard index (Jain and Dubes, 1988) is a measure of cluster similarity. It assigns to $a$ the number of sequence pairs where the sequences are in the same cluster in both clusterings, and to $b$ the number of pairs where sequences are in the same cluster in the first but not the second clustering, and to $c$ the number of pairs where sequences are in the same cluster in the second, but not the first. The Jaccard index is then defined as the number $\frac{a}{a+b+c}$.

Fig 4

In Figure 4, we plot the Jaccard indices for various clustering thresholds with a block size of 20, measured against the output from d2_cluster, BLAST clustering, and UIcluster.

We see that all clusterings reach a plateau for clustering thresholds of 45–50 and up. The closest clustering is d2_cluster, reaching a peak of 0.947 for a threshold of 72. The BLAST based clustering is also close with a maximum of 0.864 for a threshold of 94, with UIcluster less similar, with it's best value, 0.564, at 78.

Another measure, used by Burke et.al. (1999), is the number of exactly matching clusters, as well as clusters in one clustering consisting entirely of a set of clusters in the other. The number of exact clusters compared to the other clusterings is shown in Figure 5. Again, we see that we are closer to d2_cluster than to the BLAST based clustering. The number of exact clusters grows steadily, peaking at 7999 exactly matching clusters for a threshold of 88 for d2, and 7876 clusters for BLAST at 98 and 100. UIcluster's closest value is 7731 for a threshold of 100.

Fig 5

Tab 1

Table 1 shows a comparison between the different clusterings using a block size of 20 and a clustering threshold of 72. UniGene is also included, but as it does not contain all of the benchmark sequences, and as it contains additional sequences that can impact clustering, these values are, at best, only indicative. However, the values give an indication of the level of agreement between the different clusterings.

Again, the proximity of d2 and the present algorithm is underscored, while d2 is marginally closer to BLAST-based clustering.

# Discussion and Conclusion

The current prototype implementation runs with performance comparable to or better than d2_cluster and BLAST based clustering, as long as block sizes are larger than 12.

Short block sizes produce a large number of false positives, and the clustering quality as well as performance appears to be much better for larger block sizes.

While UIcluster is remarkably fast, the comparison to the more well-evaluated methods indicates that the quality is inferior. Pedretti points out some weaknesses of the implementation in his honors project[3] that support this observation.

Scalability for our algorithm appears to be close to linear in practice, and implementing the performance-critical parts of the algorithm in a lower level language with more focus on optimization should result in a noticeable constant factor speedup. The critical parts of the algorithm also lend themselves to parallel implementation. The combination of these properties suggests that a tool capable of clustering of the complete set of human ESTs available from public databases using reasonable computing resources may be within reach.

When comparing with the established algorithms, the choice for block size appears to be optimal in terms of clustering quality between 18 and 30. This is probably best interpreted as a consequence of real similarities producing longish matching regions containing matches separated by short gaps, while in some cases, the algorithm identifies short matching blocks separated by large gaps.

The current algorithm does not use gap penalties or other refinements, these options should be explored in the future in order to provide better specificity while maintaining high sensitivity.

## Acknowledgments

We wish to thank the people at SANBI and Electric Genetics who have been extremely helpful. In particular Liza Groenewald has helped to clarify the details concerning d2_cluster, and provided a version of the benchmark data masked for repeats and vector sequences, as well as the complete clustering output.

Also, we thank the anonymous referees, whose constructive criticism has improved the article

Finally, we are grateful to Stephan Haas and department Vingron at the Max Planck Institute for Molecular Genetics for their hospitality, as well as for advice, suggestions, and for providing the BLAST based clustering results.

This work was funded by The Salmon Genome Project, a Norwegian Research Council program.

[3] http://genome.uiowa.edu/pubsoft/clustering/uicluster.doc/

## References

Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. (1990). A basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

Boguski, M. and Schuler, G. (1995). ESTablishing a human transcript map. *Nature Genetics*, 10:369–371.

Burke, J., Davidson, D., and Hide, W. (1999). d2_cluster: A validated method for clustering EST and full-length cDNA sequences. *Genome Research*, 9:1135–1142.

Giegerich, R. and Kurtz, S. (1995). A comparison of imperative and purely functional suffix tree constructions. *Science of Computer Programming*, 25(2-3):187–218.

Haas, S. A., Beissbarth, T., Ribals, E., Krause, A., and Vingron, M. (2000). GeneNest: automated generation and visualization of gene indices. *Trends in Genetics*, 16(11):521–523.

Holm, L. and Sander, C. (1998). Removing near-neighbour redundancy from large protein sequence collections. *Bioinformatics*, 14(5):423–429.

Jain, A. K. and Dubes, R. C. (1988). *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, NJ.

Liang, F., Holt, I., Pertea, G., Karamycheva, S., Salzberg, S. L., and Quackenbush, J. (2000). An optimized protocol for analysis of EST sequences. *Nucleic Acids Research Vol 28 No 18*, pages 3657–3665.

Lipman, D. J. and Pearson, W. R. (1988). Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Science of the USA*, volume 85, pages 2444–2448.

Manber, U. and Myers, G. (1993). Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22:935–948.

McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of Advanced Computing Machinery (J.ACM)*, 23:262–272.

Pedretti, K. (2001). Accurate, parallel clustering of EST (gene) sequences. Master's thesis, University of Iowa, Dept. of Electrical and Computer Engineering.

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

Weiner, P. (1973). Linear pattern matching algorithms. In *Proceedings of 14th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1–11.

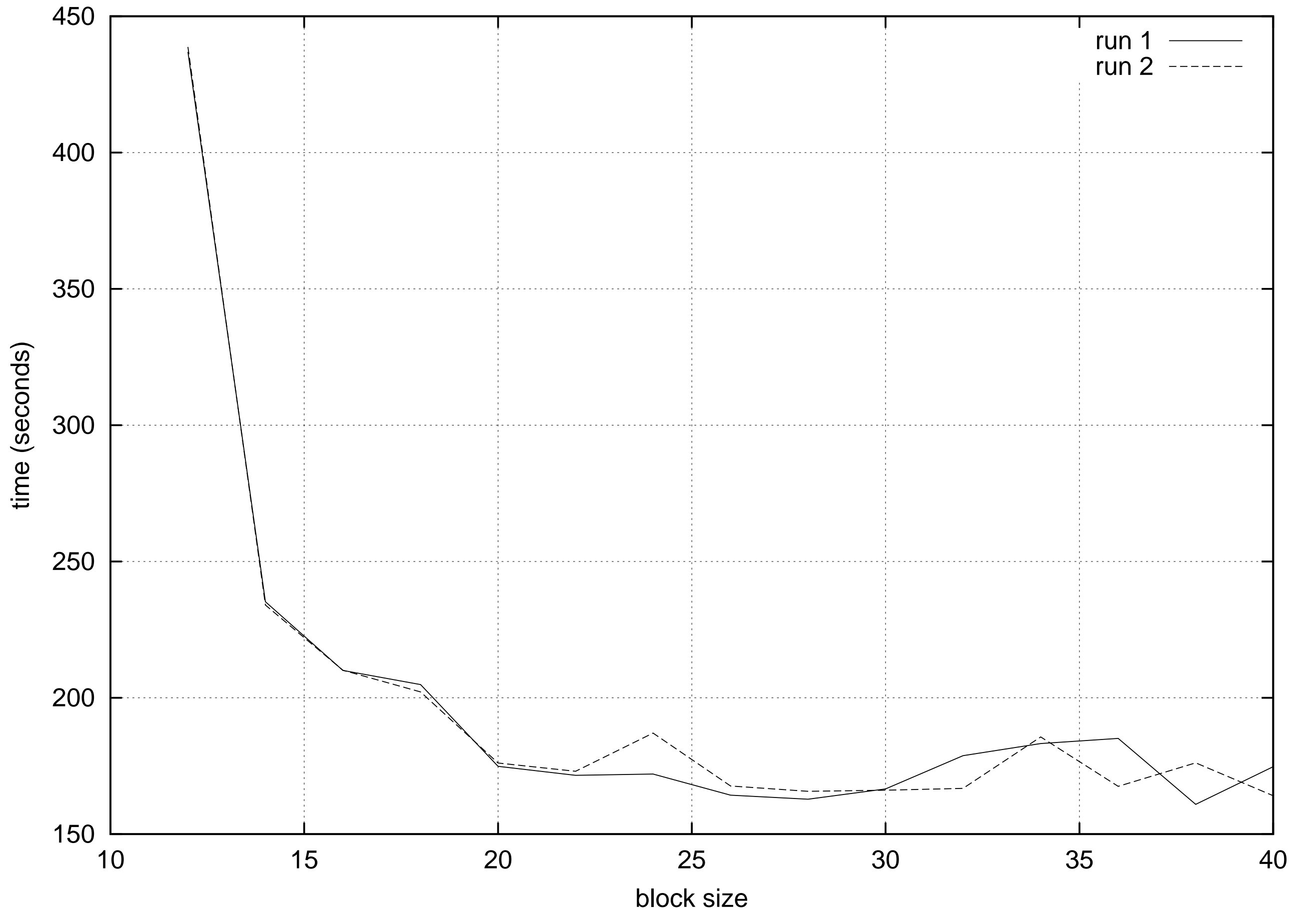


Figure 1: Times for two complete clustering runs with the block size varying from 12 to 40.

| | 20/72 | D2 | BLAST | UI | UG |
|---|---|---|---|---|---|
| 20/72 | **8103** | *0.947* | *0.854* | *0.560* | *0.503* |
| D2 | 7975 | **8118** | *0.865* | *0.559* | *0.494* |
| BLAST | 7792 | 7812 | **8298** | *0.537* | *0.473* |
| UI | 7638 | 7631 | 7672 | **8419** | *0.307* |
| UG | 1629 | 1619 | 1578 | 1458 | **2910** |

Table 1: The similarity between clusterings, with the number of clusters on the diagonal, the Jaccard index in the upper half, and the number of exactly matching clusters in the lower half.

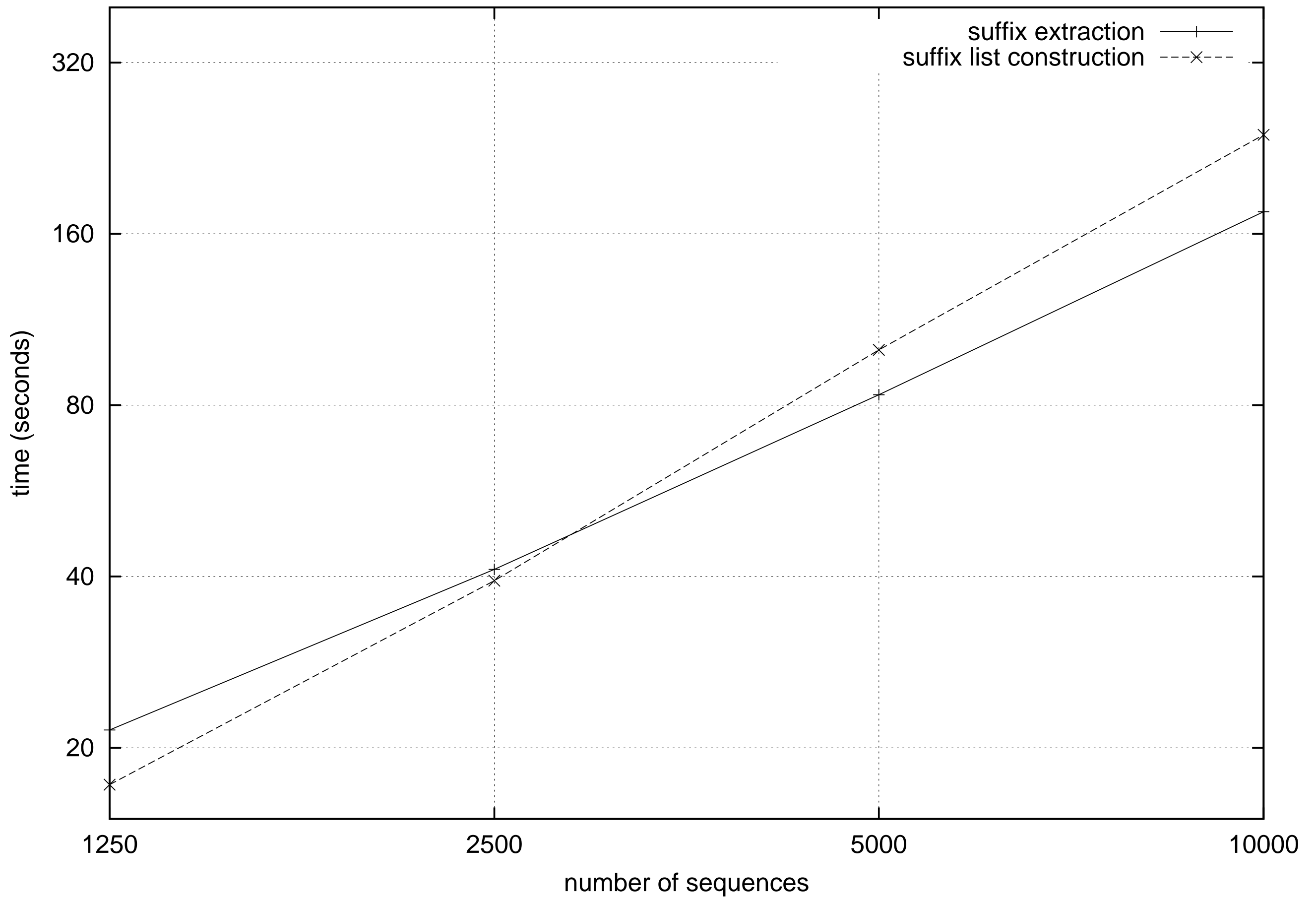


Figure 2: Time consumed when clustering data sets with different sizes.

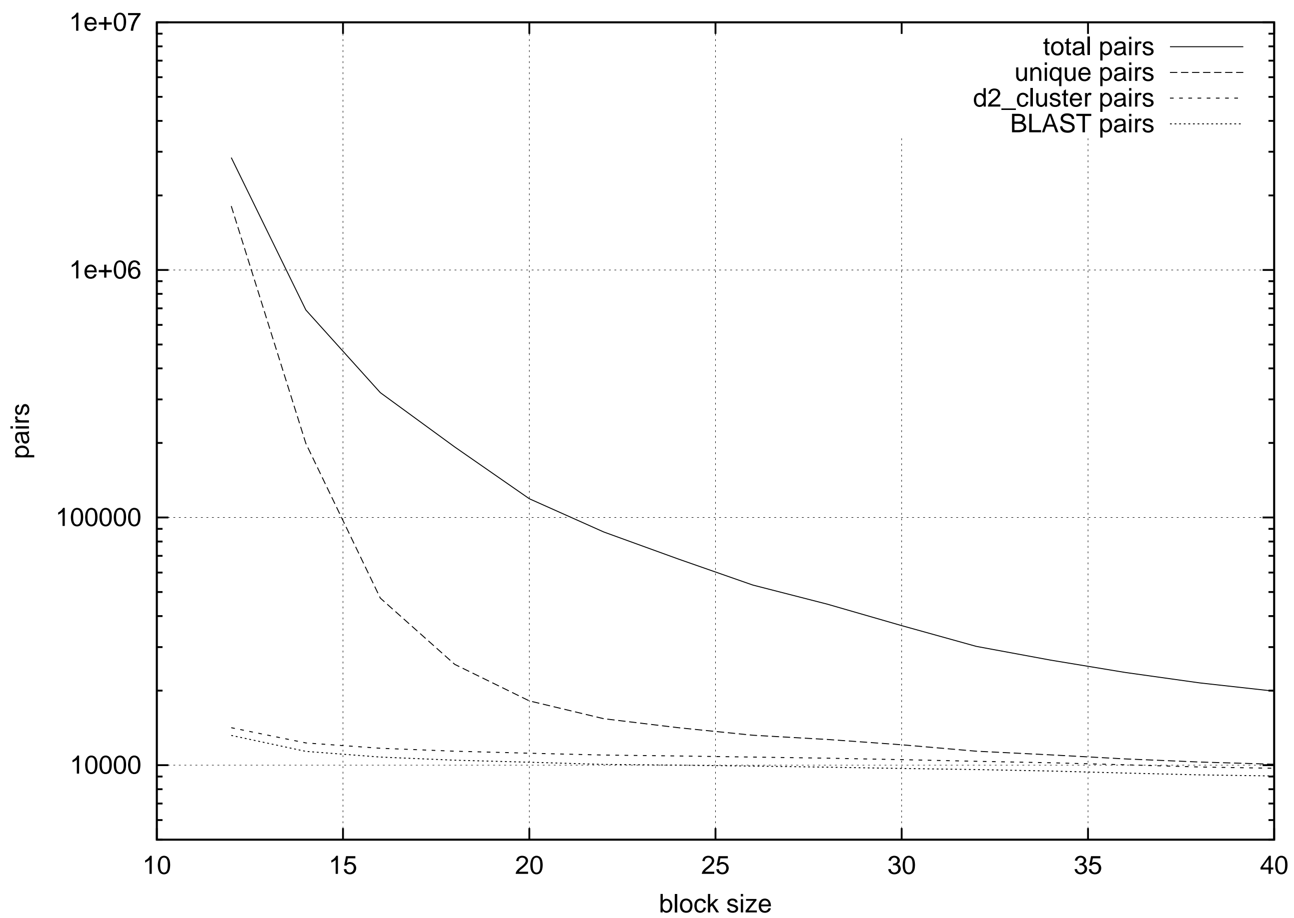


Figure 3: The total number of matches found, the number of resulting unique sequence pairs, and the number of "true" pairs produced (i.e. where both sequences are in the same d2 or BLAST cluster).

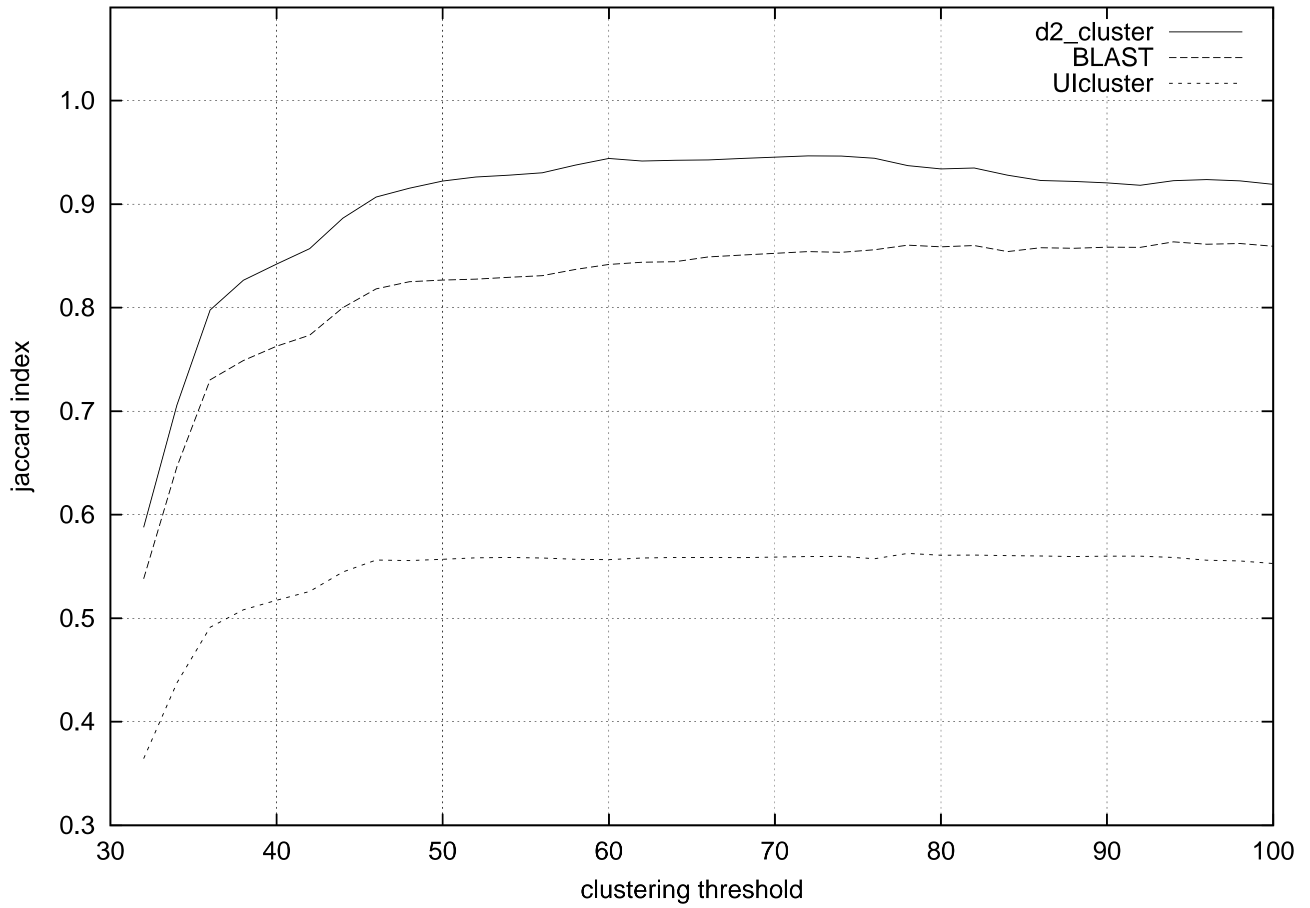


Figure 4: The Jaccard index with block size 20.

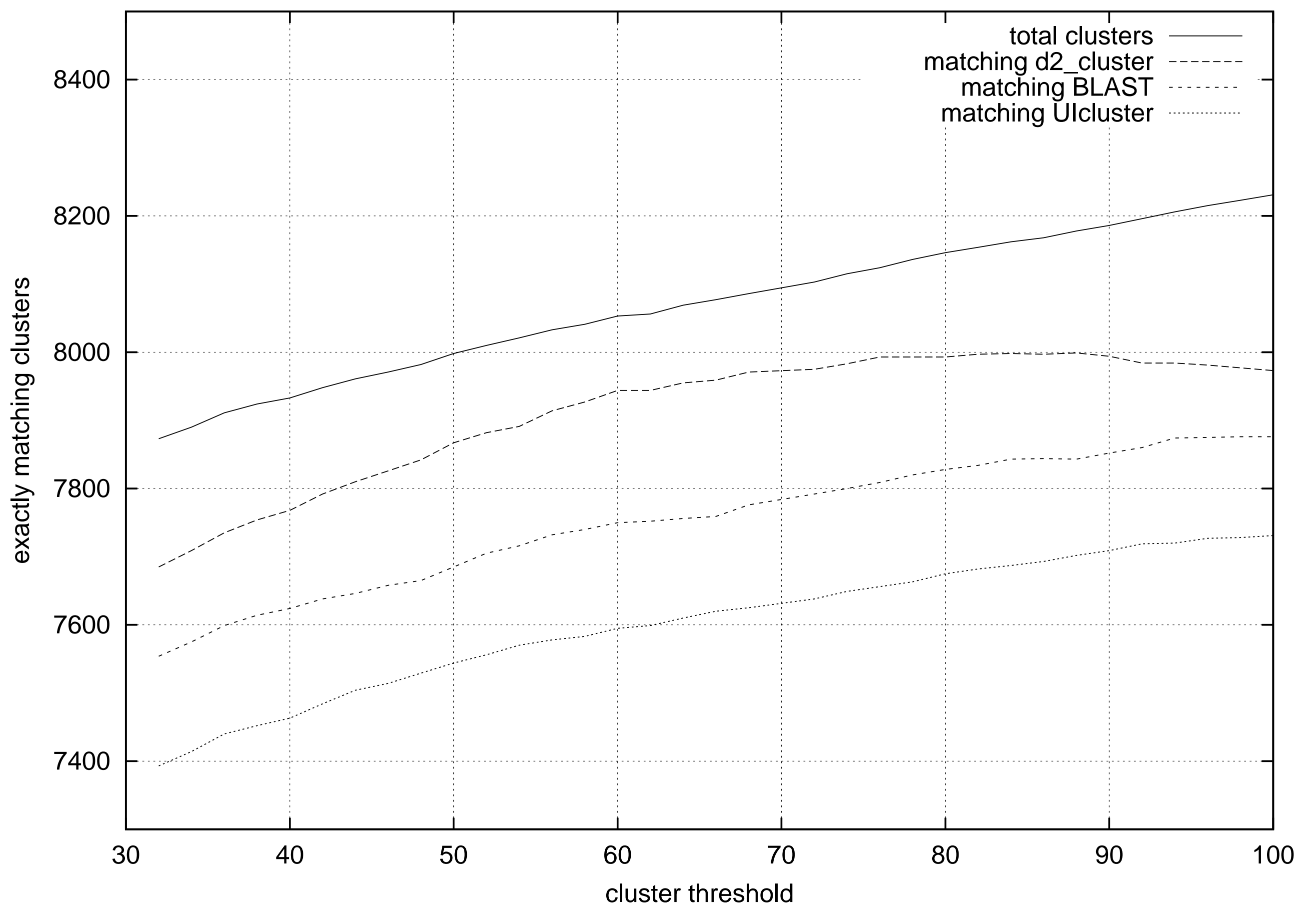


Figure 5: Counting the number of exactly matching clusters with block size 20.