Efficient Sequence Alignment of Network Traffic

Christian Kreibich, Jon Crowcroft
University of Cambridge Computer Laboratory
{firstname.lastname}@cl.cam.ac.uk

ABSTRACT

String comparison algorithms, inspired by methods used in bioinformatics, have recently gained popularity in network applications. In this paper we demonstrate the need for careful selection of alignment models if such algorithms are to yield the desired results when applied to network traffic. We introduce a novel variant of the Jacobson-Vo algorithm employing a flexible gap-minimising alignment model suitable for network traffic, and find that our software implementation outperforms the commonly used Smith-Waterman approach by a factor of 33 on average and up to 58.5 in the best case on a wide range of network protocols.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Algorithms, Measurement, Experimentation

Keywords

Traffic Monitoring, Sequence Alignment, Sequence Analysis

1. INTRODUCTION

Traditionally, content-based traffic analysis has typically involved searching the payloads of packets, either in isolation or reassembled into flows when necessary, for *known* patterns. Lately, traffic flows have also been compared to *each other*, in an attempt to identify commonalities by aligning the flow contents in a suitable fashion. Discovered commonalities can be used for follow-up analyses of many kinds, for example to automatically fingerprint malicious traffic [1, 2], mimic the modus operandi of network protocols [3], or for building traffic models suitable for anomaly detection and traffic classification [4]. Many of the relevant algorithms in this *sequence alignment* problem setting are inspired by the field of bioinformatics, where such methods are regularly used for problems of *motif finding* in large DNA sequence databases.

The contributions of this paper are twofold. First, we point out the need for careful choice of the right alignment model and implementation in Section 2. We demonstrate that subtleties in either can lead to inconsistent and, in the worst case, incorrect results. Second, we show that while the commonly used and closely related Smith-Waterman and Needleman-Wunsch algorithms provide high flexibility, they may be a suboptimal choice if the top concern is runtime performance of longest common subsequence computations on network traffic. We next introduce a novel variant of the Jacobson-Vo algorithm in Section 3. To the best of our knowledge, this algorithm has not previously been used in the networking domain. Our variant is consistent with the goals we present in Section 2 and borrows dynamic programming elements from Smith-Waterman to yield the desired results, while retaining the runtime complexity of the original approach. Jacobson-Vo's runtime performance depends on the distribution of characters in the input strings, making it hard to make *a priori* statements about its performance on network traffic. As we show in Section 4, a careful implementation in software outperforms Smith-Waterman across a wide range of network protocols and flow sizes by a factor of up to 33 in the average case, reaching up to 58.5 in the best case. We briefly discuss our findings in Section 5 before concluding the paper in Section 6.

2. CAVEATS IN SEQUENCE ALIGNMENT OF NETWORK TRAFFIC

There are many ways of finding common substrings among multiple input strings. In this paper we focus on procedures that yield *multiple* such strings per pairwise alignment and include *precise* locational information per substring. In our experience, such precision is highly desirable for many applications of alignment algorithms. More approximate methods for extracting frequent content have been proposed in the literature [5, 6].

Two input strings S_1 and S_2 of respective lengths s_1 and s_2 can be aligned either globally or locally. Global alignment assumes that two strings are largely similar and that only minor misalignments have to be identified. By contrast, local alignment assumes no inherent similarity between strings and focuses on finding regions of similarity. In practice, both problems can be solved by a tabular dynamic programming procedure introduced by Smith and Waterman.[1] We sum-

[1]The global alignment procedure is commonly referred to as "Needleman-Wunsch", though Needleman and Wunsch only discussed the global alignment problem but proposed a different (and slower) algorithm [7, 8].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IMC'06, October 25–27, 2006, Rio de Janeiro, Brazil.
Copyright 2006 ACM 1-59593-561-4/06/0010 ...$5.00.

marise the algorithm here and refer to the literature for details [9, 8, 10]. The algorithm runs in $O(s_1\ s_2)$ by filling a table of size $s_1 \times s_2$ row-by-row, in each cell recording the best alignment of the prefixes of S_1 and S_2 up to the cell's row/column indices by selecting an *edit operation* on the pair of characters at the current row/column. These operations can (*i*) skip characters of either string, (*ii*) align the characters directly, or (*iii*) accept mismatching characters via substitution. Each operation is assigned a cost/score, and the best resulting alignment is the one with the highest score (for local alignment) or lowest cost (for global alignment). The resulting alignment is extracted by walking backward through the table, starting in the bottom-right corner, following the alignment decisions taken at each cell. The algorithm computes a *longest common subsequence* $\mathrm{LCS}(S_1, S_2)$ of input strings S_1 and S_2. A common subsequence is a sequence of common substrings; a longest common subsequence maximises cumulative length.

The cost model chosen directly affects the resulting LCS and thus requires careful consideration. Showing the soundness of cost models with per-character-pair substitution costs takes substantial effort [8] and their investigation is a topic of future research for the networking domain. We feel that the soundness of this approach is undermined by limitations of the similarity to the bioinformatics domain: while nature introduces at most limited random mutation, protocols do not undergo such a process; rather, they evolve over a sequence of more or less well-specified implementations. Worse, the networking domain faces a *malicious* adversary who is aware of the model decisions we are making. We argue that when using Smith-Waterman approaches, rather than choosing an ad-hoc cost model, the alignment decision should be coded *explicitly* into the algorithm, for example by excluding the possibility of substitutions altogether and interweaving exactly matching subsequences with gaps only. We will now show how even when doing so, subtleties in the implementation can critically affect the outcome. Consider the following pair of strings:

```
'GET / HTTP'
'GET /a/a.HTM HTTP'
```

The LCSs for these strings have a maximum length of 10. Note the plural; the following are both possible LCSs of this length:

```
'GET /', ' HTTP'
'GET ', '/', ' HTTP'
```

Examples with more ambiguity are easily constructed. The key difference among the alternatives is the *number of gaps*—the most compact version has only one gap while the other has two. Without forethought it depends on the subtleties of the implementation which result is obtained, even in such basic cases as the given example. Worse, once we restrict our interest to substrings of a minimum length larger than a single character, the second LCS will yield the wrong result if we filter the detected substrings, as it will fail to report the 'GET /' substring. We argue that an LCS computation should report the LCS with the *minimum number of gaps*, since it yields more consistent results. To get Smith-Waterman to return such results, one needs to employ alignment scoring schemes that actively encourage longer contiguous substrings.

3. FAST LCS FOR NETWORK TRAFFIC

We now introduce a method for computing LCSs that adheres to the requirements presented in the previous section and that is typically substantially faster than Smith-Waterman. The method is an adaptation of an algorithm that was presented independently by Jacobson and Vo [11] and Pevzner and Waterman [12]. We will refer to the original algorithm as Jacobson-Vo and summarise its operation before enhancing it.

3.1 Jacobson-Vo: Combinatorial Reduction

Jacobson-Vo reduces a related combinatorial problem for which there is a potentially more efficient solution than $O(s_1\ s_2)$ to the LCS problem. This combinatorial problem is the identification of a longest increasing subsequence (LIS) in a sequence of numbers. Again referring the reader to the literature for details [11, 8], we continue the example of the previous section to demonstrate the algorithm. The idea is that an LIS has a one-to-one correspondence with an LCS if the sequence of numbers is produced from the two input strings in the following fashion: iterating over the characters in S_1, we list once per occurring character all indices in S_2 at which that character occurs, in descending order. This yields:

```
G  →  0              /  →  6 4
E  →  1              H  →  13 9
T  →  15 14 10 2     P  →  16
␣  →  12 3
```

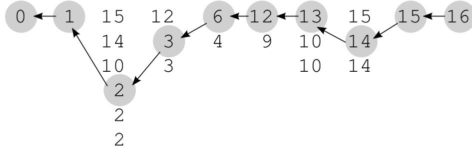
These character occurrence lists are then concatenated into a numerical sequence Π of length π. For S_1 and S_2 the beginning of Π looks as follows (dots indicate occurrence list merge points):

$$0 \cdot 1 \cdot 15\ 14\ 10\ 2 \cdot 12\ 3 \cdot 6\ 4 \cdot 12\ 3 \cdot 13\ 9 \cdot \ldots$$

The next step is to greedily extract a *cover* of Π. A cover is a set of non-increasing subsequences of Π that together use up all of its members. We can perform this extraction in a tabular fashion by building up each subsequence in one column of a *subsequence table*. Let $\mathcal{S}_n$ be the nth such subsequence. An arbitrary element in $\mathcal{S}_i$ is denoted e_i, and $\mathcal{I}^{S_1}_{e_i}$ and $\mathcal{I}^{S_2}_{e_i}$ are e_i's indices in S_1 and S_2, respectively. Iterating over the elements of Π, one selects for each element the leftmost subsequence (i.e., column in the table) that the element can extend. Extension is possible whenever the last number in a sequence is larger than or equal to new element. If no subsequence fulfills this requirement, a new one is added to the table. We obtain:

```
0   1  15  12   6  12  13  15  15  16
       14   3   4   9  10  14
       10   3          10  14
        2
        2
        2
```

To extract an LCS, first an arbitrary element in the last subsequence is selected. Afterward, the remaining subsequences are scanned downward in right-to-left order, selecting the first element e_i in each $\mathcal{S}_i$ for which $\mathcal{I}^{S_2}_{e_i} < \mathcal{I}^{S_2}_{e_{i+1}}$, where e_{i+1} is the element chosen in $\mathcal{S}_{i+1}$:



The resulting sequence of S_2 indices is an LCS of S_1 and S_2: `'GET '`, `'/'`, `' HTTP'`. To estimate the runtime complexity of this procedure, observe that the last numbers of the subsequences are sorted in increasing order at all times when scanning the table left-to-right. We can thus find the correct column for insertion via binary search. Let S_1 be the shorter of the two strings, without loss of generality. Since there can never be more than s_1 sequences in the table and we insert π elements in total, this algorithm runs in $O(\pi \log s_1)$.

3.2 Improving Jacobson-Vo: Targeted LCS Selection

Note that for our running example, standard Jacobson-Vo yields an LCS that violates the goals of gap minimisation and substring maximisation. We now extend the algorithm to overcome this limitation, borrowing several concepts from Smith-Waterman: we introduce dynamic programming to Jacobson-Vo to track incrementally the LCS that yields the smallest number of gaps and longest-possible substrings, and collect the optimal LCS via back-pointer traversal. As we will show, these extensions render Jacobson-Vo gap-minimising and substring-maximising, while retaining the same algorithmic complexity as the original algorithm.

3.2.1 *Path Selection Through Dynamic Programming*

The unmodified Jacobson-Vo algorithm does not consider possible alternatives in the selection of each subsequence's LCS member. The first step therefore is to consider the choices we have whenever an LCS member in $\mathcal{S}_{i-1}$ is selected after having selected one in $\mathcal{S}_i$. Adding the S_1 indices of each element in Π to the subsequence table (in small type), we obtain the following:

0_0	1_1	15_2	12_3	6_4	12_5	13_6	15_7	15_8	16_9
		14_2	3_3	4_4	9_6	10_7	14_7		
		10_2	3_5			10_8	14_8		
		2_2							
		2_7							
		2_8							

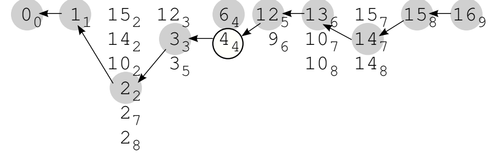
Observe that while the S_2 indices are non-increasing in each subsequence when reading top-down, the S_1 indices are non-*decreasing*. This follows from the mechanics of the algorithm: later insertions appear further down in the subsequences and are made using elements further to the right in Π. Those elements have equal or larger S_1 indices.

Assume now that we have just chosen an element e_{i+1} in $\mathcal{S}_{i+1}$. Since every element in $\mathcal{S}_i$ has least one element in $\mathcal{S}_{i-1}$ that can be chosen as its predecessor, we can pick any element e_i in $\mathcal{S}_i$ as an LCS member subject to the condition that $\mathcal{I}^{S_1}_{e_i} < \mathcal{I}^{S_1}_{e_{i+1}}$ and $\mathcal{I}^{S_2}_{e_i} < \mathcal{I}^{S_2}_{e_{i+1}}$ since only then does e_i appear before e_{i+1} in both S_1 and S_2. Given the opposite growth directions of the indices in each subsequence in the table, this means that for each e_{i+1} there exists a *window* of possible predecessors in subsequence i, and, by symmetry, for each e_i there exists a window of possible successors in subsequence $i{+}1$. More formally, the sets of elements $W_p(e_i)$ in the predecessor window of e_i and $W_s(e_i)$ in its successor window are defined as follows:

$$W_p(e_i) = \Big\{ e_{i-1} \in \mathcal{S}_{i-1} : \;\; \mathcal{I}^{S_1}_{e_{i-1}} < \mathcal{I}^{S_1}_{e_i} \;\wedge\; \mathcal{I}^{S_2}_{e_{i-1}} < \mathcal{I}^{S_2}_{e_i} \Big\}$$

$$W_s(e_i) = \Big\{ e_{i+1} \in \mathcal{S}_{i+1} : \;\; \mathcal{I}^{S_1}_{e_{i+1}} > \mathcal{I}^{S_1}_{e_i} \;\wedge\; \mathcal{I}^{S_2}_{e_{i+1}} > \mathcal{I}^{S_2}_{e_i} \Big\}$$

Returning to our running example, we see that $W_p(\mathtt{12_5}) = \{\mathtt{6_4}, \mathtt{4_4}\}$, i.e., when choosing the predecessor of element $\mathtt{12_5}$ (up to which there are no alternatives since window size is always one) we can choose between $\mathtt{6_4}$ and $\mathtt{4_4}$. If we choose the latter, we end up with the desired LCS `'GET /'` - `' HTTP'`:



Thus the goal is to use the limited freedom in selecting LCS members to minimise gap counts and maximise substring lengths in the resulting LCS. By tracking those properties incrementally on all possible paths through the table and identifying the path with least gaps and longest substrings, the algorithm will compute the desired LCS. This LCS is collected by traversing the table left-to-right from the element in the first subsequence with the highest score, using back-pointers. Note that the search still starts in the last subsequence, since scanning right-to-left has the benefit of eliminating more elements from consideration. As in the original approach, all elements of the last subsequence are potential starting points. The core strategy is to perform a *parallel downward scan* of pairs of subsequences adjacent in the table. If the table contains n subsequences, the first scan uses $\mathcal{S}_{n-1}$ and $\mathcal{S}_n$, the second $\mathcal{S}_{n-2}$ and $\mathcal{S}_{n-1}$, etc., until eventually $\mathcal{S}_1$ and $\mathcal{S}_2$ are reached.[2]

Assume the scan currently examines $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$. The scan considers all elements in $\mathcal{S}_i$ in top-down order that have a non-empty window in $\mathcal{S}_{i+1}$, ignoring the ones at the beginning of $\mathcal{S}_i$ with too high an S_2 index as well as those at the end of the subsequence with too high an S_1 index. The elements linked by back-pointers thus form a *corridor* through the subsequence table, and the upper boundary of the corridor is the LCS selected by the original Jacobson-Vo algorithm. This is illustrated in Figure 1.

As the scan proceeds from one $\mathcal{S}_i$ element to the next, the successor window $W_s(e_i)$ moves down $\mathcal{S}_{i+1}$. Let the currently considered element in subsequence i be e_i. The idea is to compute alignment scores, akin to Smith-Waterman, incrementally for all LCSs as they are considered, tracking the best-scoring one. As with Smith-Waterman, it depends on the alignment model whether "best" means maximisation (of alignment similarity) or minimisation (of edit distances). Alignment scores can penalise gaps and encourage long common substrings, but also realise other alignment policies. By prefixing e_i to the partial LCSs starting with the elements in $W_s(e_i)$ and ending in $\mathcal{S}_n$, e_i's score can be computed for each LCS depending on whether e_i introduces a gap, starts a common substring, or extends one. The best-scoring element $e^*_{i+1} \in W_s(e_i)$ is remembered by setting e_i's back-pointer to e^*_{i+1} and storing the corresponding score in e_i.

[2] Single-column tables do not permit this approach, however their occurrence means that the LCSs consist only of a single character and any member of the sole column will do.

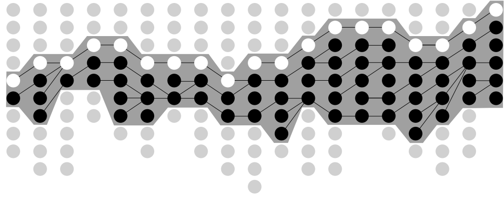


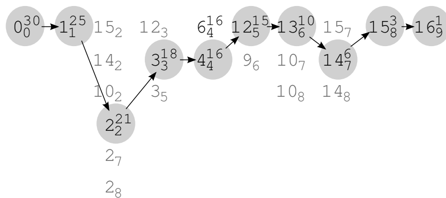
Figure 1: Corridor of linked LCS elements (shaded in grey background) through an idealised subsequence table. The elements along the upper boundary of the corridor (in white) form the LCS selected by the original Jacobson-Vo algorithm.

In order to be able to score common substrings differently from gaps, the algorithm must be able to track common substrings as they occur. Common substrings consisting of at least two characters exist whenever two LCS elements e_i and e_{i+1} have the property that $\mathcal{I}^{S_1}_{e_i}+1 = \mathcal{I}^{S_1}_{e_{i+1}}$ and $\mathcal{I}^{S_2}_{e_i}+1 = \mathcal{I}^{S_2}_{e_{i+1}}$. To notice when this is the case, the algorithm tracks the the element inside $W_s(e_i)$ whose S_1 and S_2 indices are as close as possible to, but strictly larger than, e_1's. Let this element be called e_i's *neighbour*, denoted e^n_i. A *direct neighbour* is a neighbour e^n_i for which $\mathcal{I}^{S_1}_{e_i}+1 = \mathcal{I}^{S_1}_{e^n_i}$ and $\mathcal{I}^{S_2}_{e_i}+1 = \mathcal{I}^{S_2}_{e^n_i}$, i.e., one that e_i can extend as a common substring. To formalise the neighbour definition, let the distance $\mathcal{D}$ of subsequence members e_i and e_{i+1} be defined as $\mathcal{D}(e_i, e_{i+1}) = (\mathcal{I}^{S_1}_{e_{i+1}} - \mathcal{I}^{S_1}_{e_i}) + (\mathcal{I}^{S_2}_{e_{i+1}} - \mathcal{I}^{S_2}_{e_i})$. Then e^n_i is defined as follows:

$$e^n_i \;=\; e \in W_s(e_i): \qquad \mathcal{I}^{S_1}_{e} > \mathcal{I}^{S_1}_{e_i} \;\wedge\; \mathcal{I}^{S_2}_{e} > \mathcal{I}^{S_2}_{e_i} \;\wedge\; \mathcal{D}(e_i, e) = \min_{e' \in W_s(e_i)} \left[\mathcal{D}(e_i, e')\right]$$

The neighbour always resides within W_s, since it is a legitimate successor of e_i, all of which are by definition contained in W_s. As the elements inside e_i's window are considered, a direct neighbour can be scored in a way ensures extension of an existing common substring as opposed to introducing a gap. Figure 2 illustrates sliding windows with neighbour tracking.

The introduction of alignment scoring adds significant flexibility to the algorithm, since many different scoring models become feasible. Below we show the subsequence table for the running example, with each visited element's alignment score in the top right corner, showing previous pointers where set, and using a scoring scheme that quadratically favours longer common substrings (by adding the length of the common substring to the score, for each substring character) while linearly increasing the score for gaps. Elements in grey font are outside of the corridor and not considered:



3.2.2 *Overcoming Greedy Substring Extension*

The algorithm is now gap-minimising if a scoring scheme favouring common substrings over gaps is used, because such a scoring scheme will never introduce a gap if it can extend a common substring. Whichever path has the least amount of gaps globally will be the one with the largest overall score. One problem remains: the greediness of common substring extension means that a common sequence will *always* be extended when possible due to its locally higher score, even when it would be beneficial to stop a substring and begin a new one. This situation occurs when one common substring's suffix is a later common substring's prefix.

Thankfully the problem is easy to fix: in addition to tracking with every element e_i the *globally* best score it obtains by linking with the best element in $\mathcal{S}_{i+1}$, we now also track the *local* score the element has when following the common substring it is part of through to the end. If this common substring turns out to be longer than the one it overlaps with, the local score will eventually exceed the global one and take its stead. What is left to do is to adjust the backpointer that cuts off the tail of the longer substring back into the substring.

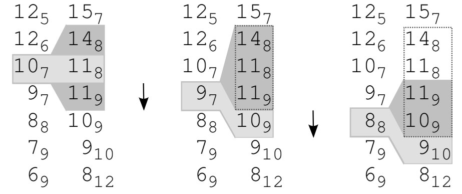


Figure 2: Parallel subsequence scanning with sliding windows. As the iteration proceeds over the left sequence's elements 10_7, 9_7, and 8_8, the window of possible successor elements slides downward. The dotted border indicates the previous window. Along with the window boundaries, the current element's neighbour (shown with lighter background) moves down as well: while 10_7 can extend the substring ending at 11_8, for 9_7 and 8_8 the introduction of a gap is unavoidable. (The string indices shown are hypothetical and not related to the running example.)

3.3 Complexity Analysis

The extended Jacobson-Vo is identical to the original one as far as construction of the subsequence table is concerned. Clearly the extended variant's runtime complexity cannot beat the original algorithm's $O(\pi \log s_1)$, since the latter does less work. The question is how costly the extension of the algorithm is. The parallel scanning phase considers every element in the left subsequence at most once, implying $O(\pi)$ additional cost. Naïvely, for each element e_i in $\mathcal{S}_i$, every element in $W_s(e_i)$ must be considered. This implies a non-constant amount of additional work per Π element which would certainly affect the overall runtime complexity negatively. The following observation comes to the rescue: unless e_i's neighbour in $W_s(e_i)$ is direct, *all* elements in the window are going to introduce gaps. In this case, and unless our alignment model scores different gaps differently, there is no reason to consider each window member. We only need to know which window member's score is *best*, and update that score according to our scoring schema. This trick renders the amount of work needed per e_i element constant, since we only need to track the best-scoring node in the window, as well as e_i's neighbour. Three pointers suffice, and since the parallel scanning phase only slides the window over each subsequence once, each of those pointers will similarly

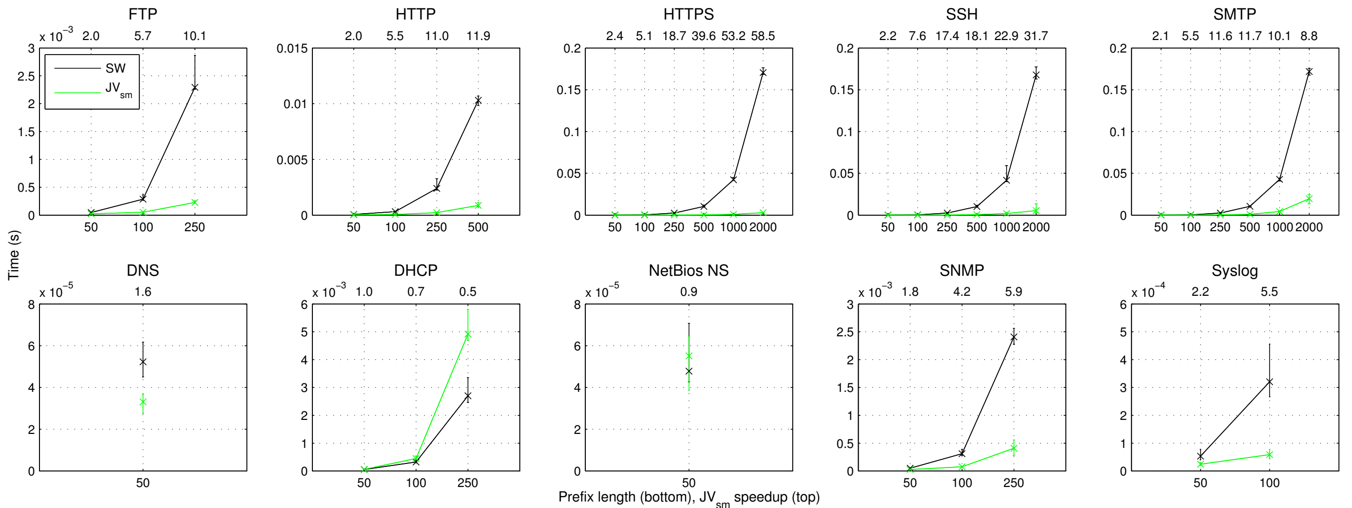


Figure 3: Performance comparison of Smith-Waterman and Jacobson-Vo on intra-protocol alignments of various TCP and UDP protocol flows. Error bars indicate the minimum and maximum runtimes.

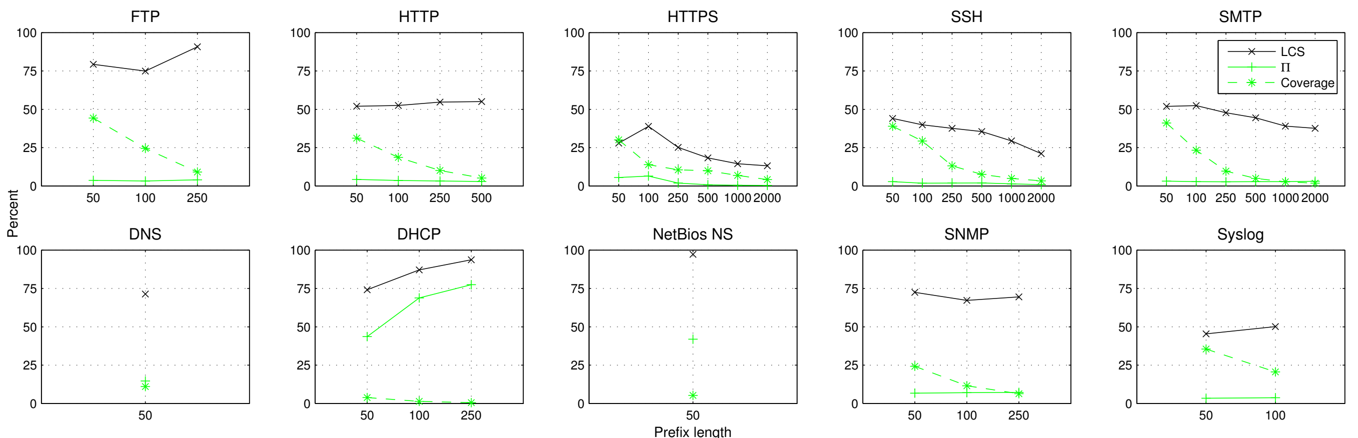


Figure 4: Behaviour of various Jacobson-Vo aspects with TCP and UDP protocol flows: length of LCS relative to $\min(s_1, s_2)$, length of Π relative to $s_1 s_2$, coverage of corridors relative to subsequence table.

visit each member of Π at most once.

At this point, the runtime complexity depends on the logistics of tracking the best-scoring node in the window. By storing the window elements in a priority queue, we can access the best-scoring element in constant time. Assume the priority queue contains n elements. As the window moves downward over a subsequence, new elements are inserted into the priority queue as the low window boundary advances. Using a heap, this can be done in $O(\log n)$. At the same time, existing elements need to be removed from the priority queue whenever the top boundary advances. Removal can likewise be done in $O(\log n)$. To estimate the maximum size n of the queue, we need to bound the size of Π's subsequences. Note that a single occurrence list can exist in a subsequence at most once in its entirety, and an occurrence list can be at most of size s_2. Beyond that, a subsequence can only grow by adding the bottom-most index repeatedly, which can occur at most s_1 times. Thus, subsequence size is bounded from above by $s_1 + s_2$.

We can now summarise the runtime complexity of our extended Jacobson-Vo algorithm. As in the original approach, we insert each member of Π into the subsequence table using binary search, requiring $O(\pi \log s_1)$. The parallel scanning phase visits each element in Π at most once in the left subsequence, while each element in the right subsequence is at most once inserted into the priority queue and removed from it, which takes at most $O(\log(s_1 + s_2))$. Combining subsequence table construction and parallel scanning phase, we obtain $O(\pi(\log s_1 + \log(s_1 + s_2)))$. Since normally we can assume $s_1 \approx s_2$ and thus $O(s_1 + s_2) = O(s_1)$, we obtain $O(2\pi \log s_1) = O(\pi \log s_1)$.

Remarkably, extending Jacobson-Vo to target gap-minimising and substring-maximising LCSs does not hurt the runtime complexity bound, making only modest assumptions about the scoring schema, namely uniform gap penalties.

4. EVALUATION

We implemented Smith-Waterman and our variant of Jacobson-Vo in about 500 and 600 lines of C++, respectively. To compare performance, we selected a number of popular servers from a one-day full-content trace of our laboratory's uplink. We selected TCP services running FTP, HTTP, HTTPS, SSH, and SMTP as well as UDP services for DNS, DHCP, NetBios NS, SNMP, and Syslog, picking $n = 142$ flows each so that we could perform $\binom{n}{2} > 10,000$

Protocol	Prefix Length 50	100	250	500	1000	2000
FTP	9,870	9,730	5,460	×	×	×
HTTP	10,000	10,000	8,778	561	×	×
HTTPS	10,000	10,000	9,870	9,316	2,346	630
SSH	10,000	10,000	10,000	9,730	8,385	5,253
SMTP	10,000	7,381	1,431	1,271	703	136
DNS	496	×	×	×	×	×
DHCP	10,000	10,000	10,000	×	×	×
NetBios NS	10,000	×	×	×	×	×
SNMP	5,778	3,828	1,596	×	×	×
Syslog	3,655	435	×	×	×	×

Figure 5: Number of LCS computations per service and prefix length.

Prefix Length	50	100	250	500	1000	2000
Avg. Speed-up	1.82	4.98	10.74	20.1	28.73	33

Figure 6: Average speed-up of extended Jacobson-Vo compared to Smith-Waterman.

LCS computations among flow pairs of the same service, an operation more meaningful than cross-service alignments and commonly performed by systems employing sequence alignment. We reassembled the originator→responder flows, where feasible, using the Bro IDS [13] and stored them in reassembled form. Next we measured the runtime for pairwise LCS computations with minimum substring length 1 of flows belonging to the same service, averaged the runtime of 100 iterations, and varied the string length in separate runs over 50, 100, 250, 500, 1000, and 2000 bytes. The experiments were run on an otherwise idle Pentium 4 running at 2.53GHz with 512MB of memory. Since flows of at least 2000 bytes are less frequent than those of at least 50 bytes, the actual number of string pairs varied per protocol. We chose 100 comparisons as the lower bound for investigation, and show the actual number of comparisons in Table 5. Figure 3 shows the performance comparison for all protocols, including the speed-up factors of Jacobson-Vo over Smith-Waterman. Our extended Jacobson-Vo algorithm is up to 33 times faster on average (see Table 6) with the best speed-up factor being 58.5 for HTTPS flows of 2000 bytes. The runtime overhead of the extended Jacobson-Vo's additional operations is marginal; we do not show it.

There are two cases where Jacobson-Vo is not the clear winner: NetBios NS and, more strongly, DHCP. To understand the reason, recall that the runtime performance of Jacobson-Vo is largely determined by the length of Π, and consider Figure 4. With NetBios NS, and DHCP in particular, Π is substantially larger than $s_1 s_2$ (the amount of work Smith-Waterman has to do) than with the other protocols. At the same time, their ratio of LCS length to input string length is not substantially higher than that of other protocols such as FTP, where speed-up is substantial. The corridor sizes relative to the full subsequence table sizes also cannot explain DHCP's behaviour, since it is among the lowest in the dataset. In summary, these observations confirm that our modifications have kept the length of Π the defining factor of Jacobson-Vo's performance.

5. DISCUSSION

Generally, Jacobson-Vo tends to perform better on content with a high number of characters in random distribution [8]. Our results confirm this: first, both ICMP and NetBios NS contain a large number of zero-bytes and are of highly fixed structure: the LCSs reach up to 93% of the input string length for DHCP and 97% for NetBios NS, indicating that the input strings are nearly identical. Second, the encrypted HTTPS has high randomisation in large parts of the content, and brings out overall best performance. Knowledge of a protocol's statistical content distribution is thus a guideline for the choice of alignment algorithm.

6. CONCLUSION

Sequence alignment algorithms have many potential applications in the network setting. As we have shown, the employed alignment models and algorithms require careful consideration. We have introduced an extension of the Jacobson-Vo algorithm that allows flexible alignment scoring, borrowing concepts from Smith-Waterman. Our software implementation outperforms Smith-Waterman by a factor of 33 on average and 58.5 in the best case. Both our Smith-Waterman and Jacobson-Vo implementations will be available with the next release of the Bro IDS.

7. REFERENCES

[1] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Christian Kreibich and Jon Crowcroft. Honeycomb — creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (Hotnets II)*, Boston, November 2003.

[3] W. Cui, V. Paxson, N. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *13th Annual Network and Distributed System Security Symposium (NDSS), San Diego, USA*, February 2006.

[4] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. Voelker. Unexpected means of identifying protocols. In *Proceedings of the Internet Measurement Conference*. SIGCOMM/USENIX, October 2006.

[5] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Usenix Security Symposium, San Diego, CA*, 2004.

[6] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation*, Dec 2004.

[7] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[8] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[9] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147, 1981.

[10] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[11] G. Jacobson and K. P. Vo. Heaviest increasing/common subsequence problems. In *Proc. of the 3rd Symposium on Combinatorial Pattern Matching*, volume 644, pages 52–65. Springer LNCS, 1992.

[12] P. Pevzner and M. Waterman. Matrix longest common subsequence problem, duality and Hilbert bases. In *Proc. of the 3rd Symposium on Combinatorial Pattern Matching*, volume 644, pages 79–89. Springer LNCS, 1992.

[13] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23-24):2435–2463, 1998.