

Trie-Based Data Structures for Sequence Assembly

Ting Chen

Steven S. Skiena*

Department of Computer Science

State University of New York

Stony Brook, NY 11794-4400

{tichen—skiena}@cs.sunysb.edu

June 11, 1997

Abstract

We investigate the application of trie-based data structures, *suffix trees* and *suffix arrays* in the problem of overlap detection in fragment assembly. Both data structures are theoretically and experimentally analyzed on speed and space. By using heuristics, we can greatly reduce the calls to the time-consuming dynamic programming, and have improved the speed of overlap detection up to 1,000 times with high accuracy in our collaborative DNA sequencing with Brookhaven National Laboratory. We also studied the problem of approximating maximum space savings in tries structures for *unification factoring* in logic programming, which is proved to be hard.

1 Introduction

Trie-based data structures for strings have proven themselves in a wide variety of text-searching and biological applications. Several distinct string data structures have been developed, including suffix trees and suffix arrays, motivated by tradeoffs between construction time and space, search time, virtual memory performance, and the complexity of implementation.

In this paper, we consider two different applications for string data structures – (1) fragment assembly for DNA sequences and (2) unification factoring for optimizing logic programs. For the biological application, we present experimental results on the performance of three different data structures, dynamic suffix trees, static suffix trees and suffix arrays, for fast fragment assembly. These are a product of our work with Dr. William Studier’s genome sequencing group at Brookhaven National Laboratory to build an assembler for sequencing the one-megabase genome of *Borrelia Burgdorferi*, the bacterium which causes

*Supported by ONR award 400x116yip01 and NSF Grant CCR-9625669.

Lyme disease. For the logic programming application, we demonstrate that, surprisingly, the complexity of optimizing unification factoring for datalog programs is considerably higher than that for more general Prolog programs. This is a product of our work with the XSB logic programming group at Stony Brook.

Although the applications are quite different, a common theme runs through them. Both require space-efficient data structures for strings. Sequence assembly requires fast substring search on large sets of strings. Unification factoring requires minimum size tries capable of representing a fixed set of strings. Indeed, our work with unification factoring suggests a new direction for reducing the size of trie and suffix-tree data structures for other string applications.

Our specific results on fragment assembly include:

- A fast sequence assembler based on exact-match overlap detection, which proves capable of detecting the overlaps of 3,800 fragments of the *Borrelia Burgdorferi* sequence within ten minutes on a Sparc1000. By comparison, the Brookhaven group's previous assembler program takes approximately fifteen hours on the same data. Indeed, our assembler proved fast enough to eliminate the need for incremental fragment assembly, the original problem with which we began this work. Our times are certainly comparable with such state-of-the-art assemblers as the TIGR assembler [15], which assembled the 24,304 fragments of the bacterium *Haemophilus influenzae* in 30 hours.
- A careful experimental study of the impact of exact-match length on the accuracy of overlap detection, which demonstrates that over a wide range of sequencing error rates (including those realized by the Brookhaven group) exact-match suffices for accurate assembly. This study compares the overlap of unprocessed ABI reads (fragments) generated our program with the overlap graph induced in the final assembly on clean data by the biologists. Using a transitive overlap recovery strategy, our exact-matching program misses only *nine* of 4,320 edges on 35kb of *Borrelia* data compared with an exhaustive pairwise Smith-Waterman computation, taking over 1,000 times as long.
- An experimental comparison between three data structures for fast overlap detection – static suffix trees, dynamic suffix trees, and suffix arrays, which shows suffix arrays to be a clear winner over all data structures in both construction and traversal time, and space.

Unification in logic programs is performed by constructing a trie from the arguments of the rule heads and performing a depth-first search of this trie against a specific goal. The time complexity of this operation is a function of the number of edges in the trie. An *open goal* contains all distinct variables in its arguments, like $p(X, Y, Z)$ and hence matches everything. In unifying an open goal against a set of clause heads, each symbol in all the clause heads will be bound to some variable. By doing a depth-first traversal of this minimum edge trie, we minimize the number of operations performed in unifying the goal with all of the clause heads.

For a set of m rule heads, each with n arguments, this trie can range in size from $n + m$ to $n \cdot m$, depending upon the length of common prefixes among the rule heads. Since we have

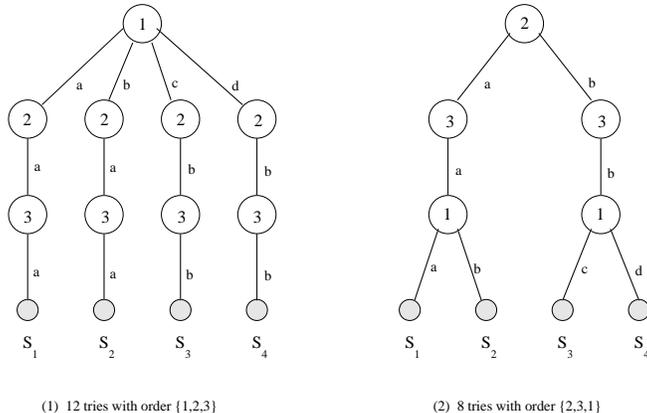


Figure 1: Two different tries for the same set of strings.

all the rule heads at compilation time, we can exploit our freedom to reorder the character positions in the trie in order to reduce its size. Instead of the root node always representing the first argument in the trie, we can choose to have it represent the third argument.

Consider the following example consisting of four strings: $s_1 = aaa$, $s_2 = baa$, $s_3 = cbb$, and $s_4 = dbb$. A trie constructed according to the original string position order (1, 2, 3) uses a total of 12 edges, as shown in Figure 1. However, by permuting the character order to (2, 3, 1), we obtain a trie with only 8 edges. Note that different permutations may be used along various paths of the same trie, since we assume that each internal node (and not level) contains a specification of the next probe position.

The problem of *unification factoring* is to use this freedom to build a minimize size trie for the set of rule heads. Beyond the context of unification, this suggests a new idea for reducing the space needed for trie-like data structures for many off-line string search applications, since there is no reason to compare characters in left-to-right order when the entire string sits in random-access memory. Any root-to-leaf path down the trie defines the full string with some permutation of its characters. The encoding becomes unambiguous when each internal node contains the index of the next character position to be probed.

Note that there is a potential for enormous space reduction even for tries of highly structured strings. For example, consider the trie of the following n binary strings, each m characters long. The first $\lg n$ characters of each of the i th string will represent i written in binary, while the last $n - \lg n$ characters will be all the same character. Building a conventional search trie on these strings will use $(m - \lg n)n + 2n - 1$ edges and quadratic space. However, by permuting the character order to as to probe first from position $\lg n + 1$ yields a linear size trie with only $m - \lg n + 2n - 1$ edges.

We investigate approximation algorithms for minimizing the size of such permuted order tries. Unfortunately, our main result is a negative one, that it is hard to approximate the savings of such a construction to within a polynomial factor. Still, we believe that heuristics for constructing such data structures can have interesting behavior in practice.

Our paper is organized as follow. Section 2 provides an introduction to the problem of fragment assembly for DNA sequencing. Section 3 presents the ideas behind our assembler, and experimental results of its performance. Section 4 provides an introduction to unification

factoring for logic programming, while Section 5 presents our applicable inapproximability results.

2 Overlap Detection for Fragment Assembly

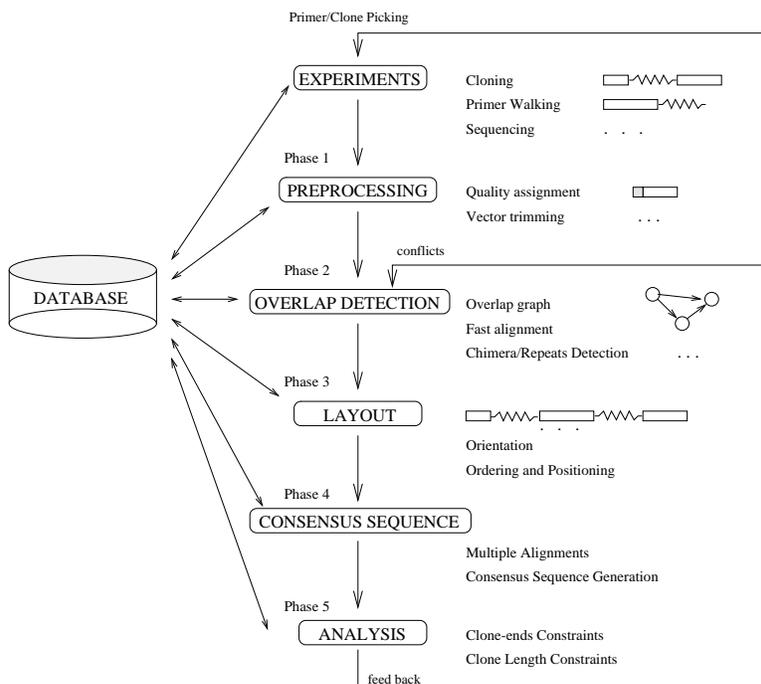


Figure 2: The fragment assembly process for directed primer walking.

The fragment assembly problem has been characterized as finding a shortest common superstring of a set of fragments within a certain assumed error rate ϵ . The common approach to this problem is to divide assembly into three stages: *overlap detection*, *layout generation*, and *consensus sequence construction* [12, 8]. In overlap detection, every fragment has to be compared against all other fragments for similar substrings which define possible overlaps between them. There may exist conflicts among these overlap relations, to be resolved in the layout stage, which determines the *orientation* of each fragment (i.e. whether it belongs to the upper or lower strand of the molecule) selects the subset of overlaps which most reasonably reflect the relations of their physical maps in the genome to determine their *ordering* and also approximate locations. Finally, the consensus stage builds the multiple-alignment of all regions where two or more fragments overlap with each other, and generates a single consensus DNA sequence.

Figure 2 presents the flow of control of Brookhaven's directed primer walking approach to DNA sequencing, being used in the *Borrelia Burgdorferi* project. Although their technique differs somewhat from conventional shotgun sequencing, the overlap detection problem is identical, and forms the focus of our work.

The result of overlap detection is to generate an *overlap graph*, where vertices correspond to fragments, with edges between pairs of fragments which overlap. Edge weights can be

used to measure the confidence of detected overlaps. Overlap detection is the most time-consuming part of typical assembly programs. Given n fragments, there are $n(n + 1)/2$ overlap candidates to consider. The Smith-Waterman dynamic programming [16] algorithm takes $O(l^2)$ time to detect whether two l -length fragments truly overlap.

For small-size sequencing projects (say cosmid level), where $n \approx 500$ and $L \approx 400$, a program with $O(n^2 l^2)$ complexity is feasible, taking several hours. However, for megabase genome-level sequencing projects, such as *Borrelia*, where $n \approx 20,000$, faster algorithms are needed.

The problem of fragment assembly has received considerable attention – see [12] for a detailed survey. Kececioglu and Myers [8] have shown that given an assumption of maximum error rate ϵ , the edit distance of two fragments can be solved in $O(\epsilon l^2)$ time by aligning suffixes incrementally. For fragments of total length nl , the overlap detection can be done in time $O(\epsilon n^2 l^2)$. If the error rate ϵ is low, i.e. 2%, this algorithm will be 50 times faster than the complete dynamic programming. In a large shotgun sequencing project of a complete genome (1,830,137 base pairs) from the bacterium *Haemophilus influenzae*, the TIGR assembler [15], assembles 24,304 fragments in 30 hours. It builds a table of all 10 base pairs oligonucleotide subsequences to generate a list of potential fragment overlaps. Then it uses a fast initial comparison of fragments (similar to BLAST [1]) to eliminate false overlaps in the list before applying the Smith-Waterman dynamic programming algorithm. Phrap [7] compares pairwise fragments by an efficient implementation of the Smith-Waterman algorithm called SWAT, which is claimed to be ten times faster than BLAST. SWAT uses recursion and word-packing to search similarities between two fragments and stores the alignment information for the significant matches. Then based on one or more matching words found, it scores two fragments within a constrained bands of the Smith-Waterman matrix.

2.1 Exact Matches and Overlap Reconstruction

In fact, for real sequencing projects, there are only $O(n)$ true overlaps, because the typical shotgun sequencing strategy uses roughly six-times genome coverage. Thus on average, each fragment will physically overlap only a constant number of other fragments. One approach to avoiding quadratic behavior is to filter the comparisons by search for all exact matches of a certain length k (say $k = 14$) as a threshold to quickly reject many non-overlapping pairs of fragments. For two random l -length fragments, the probability that they do not share any k exact matches is $(1 - l/4^k)^l \approx e^{-l^2/4^k}$, which is 99.905% for $k = 14$ and $l = 500$, so false matches are rare. This idea implies a speedup more than 1,000 times using 14 exact matches.

The potential danger is that sequencing errors may render exact matching too unreliable for overlap detection. We suggest a new strategy to reconstruct some missed overlaps by the transitive relations between three fragments f_i, f_j, f_t : if both f_i and f_j are overlapped with f_t and their alignments on f_t are overlapped, then f_i and f_j are likely overlapped with each other. There are two cases shown in Figure 3. The first case is obvious. The second one can greatly reduce the conflicts happened in the *Layout* phase; layout algorithms such as finding maximum weight path may have trouble in linking A, B and C into a single path.

Another main contribution of this paper is a careful experimental analysis of the accuracy

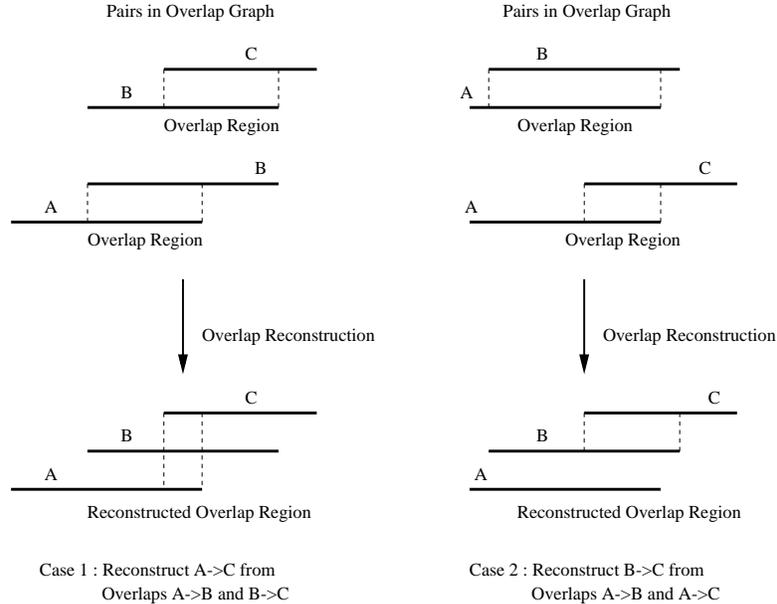


Figure 3: Reconstruct missed overlaps from transitive relation.

of exact-match filtering for both real and simulated data. We investigate the tradeoff between the exact-match length k and the accuracy to be achieved. The larger the value of k , the more likely we will miss some real overlaps, although it is faster and more likely the candidates we find will be true overlaps.

The sequencing error rate varies from laboratory to laboratory, with about 2% per base error rates being typical. The end of fragments (beyond 350 base pairs) have higher error rate than in the middle (between 50 and 350 bps). For two fragments whose physical locations on the genome overlap in a region of 60 nucleotides, a 2% error rate yields an average of less than 3 errors of this region. Thus there is a very high probability that their must exist an exact match of length ≥ 14 . We find that exact matching works well even at higher error rates: most of the missed overlaps can be reconstructed using a third fragment with a transitive relation, because of the redundancy in sequencing coverage. Moreover, the error rates have been steadily reduced with the development of sequencing techniques, and the exact matches strategy will be more and more powerful in the sequencing.

2.2 Suffix Trees and Suffix Arrays

In this paper, we evaluated both suffix trees [3] and suffix arrays [11] data structures for overlap detection. Suffix arrays show a significant advantage over suffix trees, not only on space but also on speed. We review these data structures. For both, we are given a text string $X = x_1x_2x_3\dots x_n$, where each x_i is a member of an alphabet Σ , and seek to preprocess X such that given a pattern $P = p_1p_2p_3\dots p_m$, ($p_i \in \Sigma$), the set $\{i : x_i\dots x_{i+m-1} = P\}$ can be found efficiently.

- *Suffix Trees* – the compressed trie for all the suffixes of X_1, \dots, X_n , where $X_i = x_ix_{i+1}\dots x_n$. A sample suffix tree for the string *aabbaab* is shown in Figure 4. All

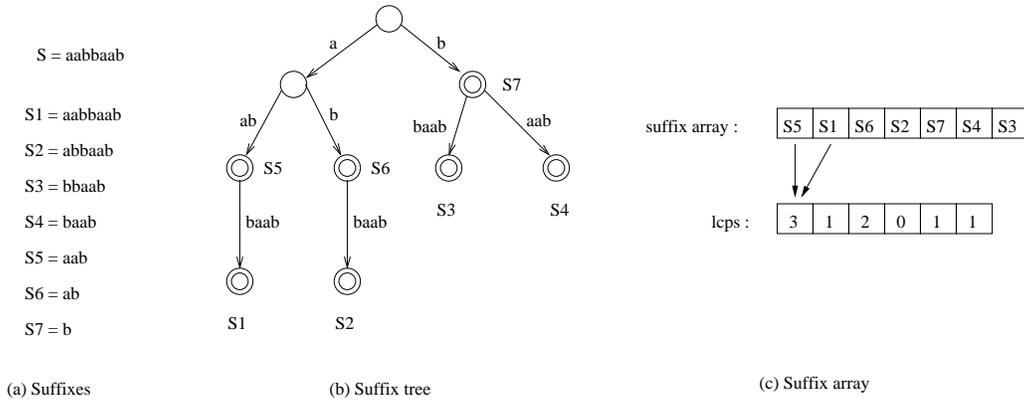


Figure 4: Suffix tree and suffix array for *aabbaab*.

the edges of the tree, representing a substring of X , can be implemented by a pair of pointers, so the total size of the data structure is linear. Suffix trees can be built in $O(m)$ time, where m is the length of the text. They have a memory requirement of approximately $17m$ bytes [11], which is a problem for large texts (such as megabase sequencing projects), but permit searching for a string p in $O(|p|)$ time.

- *Suffix Array* – basically a sorted list of all suffixes of X . If it is coupled with an array of the *longest common prefixes (lcps)* of adjacent elements in the suffix array, string searches can be answered quickly using binary search. Suffix arrays can be built in expected $O(m)$ time, where m is the length of the text. They have a memory requirement of only $6m$ bytes, and support searching for a string p in $O(|p| + \lg m)$ time.

There is no significant complexity difference in building these two data structures. Although the computation of a suffix array consists of sorting M strings, there exist $O(M)$ construction algorithms. One idea is to construct a suffix tree, then build the array with a traversal of the tree. In practice, we can do it in expected linear time via bucket-sort. The best known encoding of technique for the suffix tree requires approximately 17 bytes per index (symbol) while suffix array needs only 4 bytes for identifying each suffix and 2 bytes for each *longest common prefix*. For a very long string or multiple strings, those numbers will be doubled. Both data structures need only constant space for each index, but in the application of long strings, suffix array will be much more efficient than suffix tree: the whole suffix array is more likely to stay in the memory instead of swapping into the secondary storage. Despite the advantage of suffix trees to search a p length pattern in $O(p)$ time, experiments done by Myers [11] show that for 100,000 characters text files, suffix arrays are at least as fast as suffix trees.

A second idea to avoid the $O(l^2)$ expense of a full Smith-Waterman computation is to use a linear or super-linear heuristic to compare two fragments, such as FASTA [13] and BLAST [1]. We use a FASTA-like approach in our implementation.

3 Exact Matching for Fast Sequence Assembly

3.1 Implementation Details for Data Structures

We built two different programs for finding overlaps among DNA sequences, which are strings consisting of symbols a, g, c, t, n , where a, g, c, t are nucleotides and n means unknown.

One suffix tree is constructed by a linear time algorithm and contains each fragment and its reverse complement. To search all the common substrings at least length k , we traverse the tree, and for any node whose distance to the root (as the number of text symbols) is larger than and equal to k , we return the subtrees whose leaf-suffixes share a common prefix at least k .

There are two distinct ways to implement suffix trees, which differ on how the out-edges of a node are encoded. One option is to fully encode the out-edges as a vector which has 5 pointers for the DNA fragments. Let M be the number of suffixes and I as the number of nodes in the suffix tree. This structure requires $4M + 32I$ bytes, consisting of 20 bytes for each out-edges vector, 6 bytes for an arc(or an in-edge), 4 bytes for each suffix, 2 bytes for the distance from a node to the root, and 4 bytes for each suffix link. A second encoding technique dynamically allocates space for nonempty out-edges only. It requires only $4M + 16I$ bytes.

The *suffix array* (sa) and the *longest common prefixes* ($lcps$) array can be constructed in expected linear time. To find all the fragments sharing common substrings at least length k , we can traverse the $lcps$ array and return all the intervals $[l, r]$ satisfying $lcps[l - 1] < k$, $lcps[r + 1] < k$, and for any $l \leq i \leq r$, $lcps[i] \geq k$. It is obvious that for any such an interval $[l, r]$ in $lcps$, the suffixes of $sa[l], sa[l + 1], \dots, sa[r], sa[r + 1]$ in suffix array sa share a common prefix at least length k . In Figure 4, for $k = 1$, the interval $[1, 3]$ in $lcps$ satisfies that suffixes S_5, S_1, S_6 , and S_2 share the common longest prefix a .

The suffix array can be implemented as follows. We first encode the fragments into long integers. For example, every 6 indexes can be encoded into a number between 0 and $5^6 - 1$, and this simple coding technique can speed up the whole construction of suffix array. Then we bucket-sort all the suffixes and quick-sort each bucket. The *longest common prefixes* array can be quickly computed within each bucket by comparing the encoded integers. The structure requires only 6 bytes for each suffix in the sorted array and $lcps$ and 4 bytes for auxiliary encoding, for a total of $10M$ bytes, where M is the number of suffixes.

We evaluated our overlap detection on the following data sets:

1. Edited shotgun sequencing data from a 35kb cosmid of *Borrelia* from Brookhaven, consisting of 448 fragments, with a total length of 187,105 base pairs.
2. Raw shotgun sequencing data from the same cosmid, with total length of 189,286 base pairs and about 5% errors.
3. Simulated shotgun fragments from the cosmid of (1), containing 610 fragments and 246,479 base pairs, with 2% errors.
4. Same fragments as (3) but with 5% errors.

5. Same fragments as (3) but with 7% errors.
6. Same fragments as (3) but with 10% errors.
7. Shotgun sequencing data from the full *Borrelia Burgdorferi* sequencing project at Brookhaven National Laboratory. It consists of 4,612 fragments totaling 2,032,740 base pairs with an unknown error rate estimated at 2-5%.

Test sets (2) (3) (4) and (5) were generated by the shotgun simulation program Genfrag [6] which randomly splices the 35k cosmid DNA sequence into 610 shot gun fragments, totaling 7 times coverage and randomly generate 2%, 5%, 7% and 10% errors (insertion, deletion and mutation) into the fragments.

Data set	Space (Bytes/Base Pair)				
	Fragments	Base pairs	Suffix Trees		Suffix Arrays
			Dynamic	Vector	
(1) Edited 35k	448	187,105	47.5	68.5	20
(2) Raw 35k	448	189,296	67.0	113.8	20
(3) Simulated (2% errors)	610	246,318	65.6	112.7	20
(4) Simulated (5% errors)	610	246,476	69.3	117.3	20
(5) Simulated (7% errors)	610	246,388	68.5	117.1	20
(6) Simulated (10% error)	610	246,373	67.2	109.3	20
(7) BNL Project	4,612	2,032,738	62.6	105.4	20

Data set	Construction Time (Seconds)				
	Fragments	Base pairs	Suffix Trees		Suffix Arrays
			Dynamic	Vector	
(1) Edited 35k	448	187,105	40	31	51
(2) Raw 35k	448	189,296	40	36	25
(3) Simulated (2% errors)	610	246,318	38	38	24
(4) Simulated (5% errors)	610	246,476	50	50	34
(5) Simulated (7% errors)	610	246,388	39	41	25
(6) Simulated (10% errors)	610	246,373	38	39	24
(7) BNL Project	4,612	2,032,738	443	481	334

Data set	Traversal Time (Seconds)				
	Fragments	Base pairs	Suffix Trees		Suffix Arrays
			Dynamic	Vector	
(1) Edited 35k	448	187,105	7	2.8	0.22
(2) Raw 35k	448	189,296	12	4.4	0.22
(3) Simulated (2% errors)	610	246,318	14	5.8	0.28
(4) Simulated (5% errors)	610	246,476	15	6.0	0.31
(5) Simulated (7% errors)	610	246,388	16	6.0	0.29
(6) Simulated (10% errors)	610	246,373	15	5.9	0.29
(7) BNL Project	4,612	2,032,738	118	47.1	2.47

Table 1: Comparing Suffix Tree and Suffix Array time and space performance on sequence data sets

Table 1 summarizes timing and space experiments on each of the data sets, running on Sparc100 Workstation with 512 Mb RAM. The length of each DNA fragment ranges from 100 to 1000 base pairs. Each program was run twice and the timings averaged to compensate for system load. The traversal time for each data structure is important because it is proportional to the time for exhaustive search for exact matches. Each fragment actually

represents two DNA sequences: both itself and its reverse complement. Thus the actual number of text symbols are the double of the *base pairs (bps)*.

Table 1 shows that the dynamic suffix trees save about half the space of the vector suffix trees, and its constructions time is competitive. The reason it is slower to traverse is because the out-edges of a node have to be decoded before its children are visited. The suffix arrays are much more efficient in space, and they will beat the suffix trees in big sequencing projects. The construction time of suffix arrays is better than the suffix trees. The only exception on data (1) is because the edited fragments share very long common substrings and make the computation of *lcps* harder. Another advantage of the suffix arrays is that it is fast to traverse. The traversing time measures the expense to perform the exact matching. Its linear structure makes this work much easier and faster.

3.2 Accuracy

The accuracy of exact matching strategy is measured by *Sensitivity* and *Specificity*, defined as follows.

$$Sensitivity = \frac{\#TrueOverlapsDetected}{\#TrueOverlaps}$$

$$Specificity = \frac{\#TrueOverlapsDetected}{\#DetectedOverlaps}$$

A high *Sensitivity* program will rarely miss any true overlaps, and a high *Specificity* program will rarely claim non-overlaps pairs to be overlaps. Generally, a good strategy must be high in both numbers.

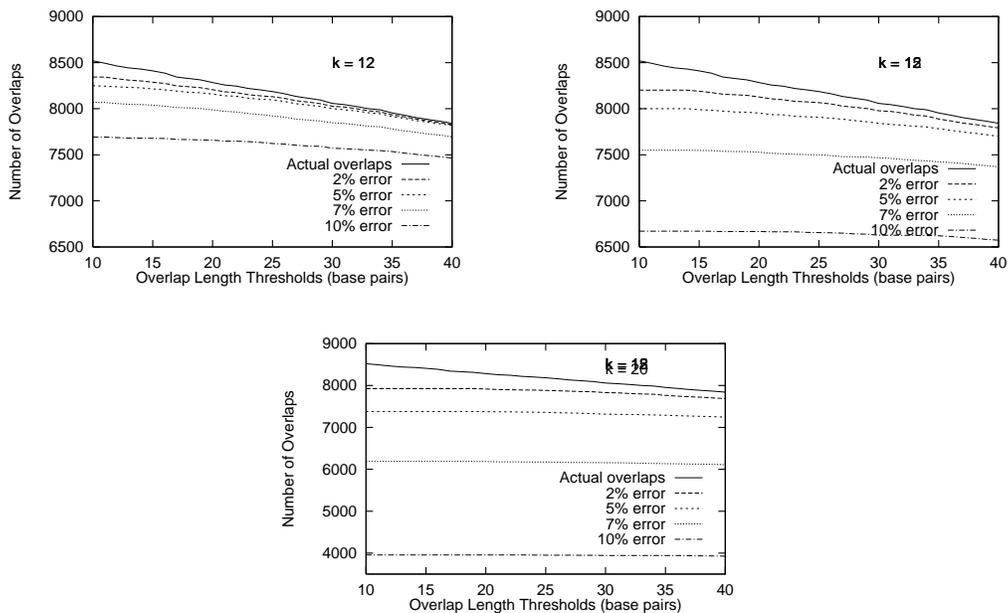


Figure 5: Number of Detected Overlaps when k=12, 15, 20 under errors 2%, 5%, 7%, 10%.

Figure 5 shows the *Sensitivity* under different error rates, which calculates how many actual overlaps are detected by k length exact matches. The first curve in each plot gives the number of actual overlaps with different length thresholds, from 10 to 40. The four curves under it from top to bottom correspond to the number of overlaps found in different data sets with errors 2%, 5%, 7% and 10% respectively. Whenever the sequencing error rate is less than 5%, we rarely miss true overlaps, even for $k = 20$, but with the errors of 10%, $k = 15$ misses 15% of the overlaps with overlap length at least 40. This figure suggests that with high error rate fragments ($> 7\%$), smaller match lengths perform better. With low error rates ($\approx 2\%$), there is no significant difference between a range of k s, so we can select the length which gives us the best running time. Fortunately, most of the sequencing data we have seen falls within the low error rates, so we have the freedom to make the tradeoff.

k	2% Error		5% Error		7% Error		10% Error	
	True ovpls /Candidates	Spec %	True ovpls /Candidates	Spec %	True overlaps /Candidates	Spec %	True ovpls /Candidates	Spec %
12	8,346/50,968	16.4	8,252/41,984	19.7	8,074/38,386	21.0	7,698/32,744	23.5
13	8,296/22,304	37.2	8,172/18,886	43.3	7,902/17,332	45.6	7,386/14,920	49.5
14	8,250/12,830	64.3	8,084/11,686	69.2	7,732/10,672	72.5	7,084/9,538	74.3
15	8,202/9,628	85.2	8,000/9,108	87.8	7,552/8,496	88.9	6,672/7,424	89.9
16	8,150/8,602	94.7	7,876/8,166	96.4	7,344/7,630	96.3	6,198/6,414	96.6
17	8,094/8,352	96.9	7,750/7,860	98.6	7,092/7,204	98.4	5,750/5,812	98.9
18	8,032/8,036	100.0	7,642/7,650	99.9	6,816/6,846	99.6	5,200/5,222	99.6
19	7,994/7,994	100.0	7,502/7,508	100.0	6,536/6,538	100.0	4,572/4,578	99.9
20	7,926/7,926	100.0	7,380/7,380	100.0	6,190/6,190	100.0	3,956/3,956	100.0

Table 2: Specificity of overlap detection on *Borrelia* sequence with simulated errors.

Table 2 shows how many of the candidate overlaps are detected, and how many of them represent true overlaps (Specificity). A small match-length k will preserve most of the true overlaps but contain many false overlaps, while a large k identifies few false overlaps but misses some true overlaps. In general, the number of true overlaps in the candidates reflects the potential accuracy of k -match search, while the number of false overlaps in candidates measures the amount of work to identify them. Ideally, we seek a value of k with the most candidates true and few true overlaps missed.

3.3 Performance

In addition to the simulated fragments, we have run our program on raw ABI machine sequencing data, after only vector trimming. Our program picks the candidates with at least one length- k exact-match, and then evaluates the quality of overlaps by FASTA-like dynamic programming and retain them with at least 25 matches and at most 20% errors. Thus we use a procedure to reconstruct overlaps without a k length common substrings. Later, we apply the strategy of overlap reconstruction described before to detected missed overlaps. We compared the sensitivity, the detected overlaps over all overlaps, the specificity, and the speed of our reconstructed data with that of performing a full Smith-Waterman on all pairs of sequences in Table 3.

Table 3 shows that the overwhelming majority of the overlaps can reconstructed using

k	kmer+DP			Reconstruct			Time
	Ovlps	Sen(%)	Spe(%)	Ovlps	Sen(%)	Spe(%)	Sec.
12	3,963	91.6	100.0	4,031	93.3	100.0	567
13	3,935	91.1	100.0	4,031	93.3	100.0	146
14	3,862	89.9	100.0	4,025	93.2	100.0	98
15	3,834	88.7	100.0	4,008	92.8	100.0	79
16	3,794	87.8	100.0	4,008	92.8	100.0	70
17	3,746	86.7	100.0	4,008	92.8	100.0	67
18	3,683	85.3	100.0	4,003	92.7	100.0	65
19	3,601	83.8	100.0	3,998	92.5	100.0	64
20	3,577	82.8	100.0	3,998	92.5	100.0	63
DP	4,040	93.5	100.0	4,040	93.5	100.0	151,200

Table 3: Sensitivity, Specificity and Speed of k -mer overlap detection on raw 35kb *Borrelia* sequence, comparing to 4,320 true overlaps (length ≥ 25).

large exact-matches. Further, all the overlaps we claim are correct. This is very promising because we can reconstruct almost all the overlaps we could if we had run Smith-Waterman on all pairs of fragments. For example, for $k = 12$, we can reconstruct 68 overlaps and miss only 9 (0.2%) overlaps compared to the overlaps found by fully dynamic programming (DP). Even for $k = 20$, we successfully reconstruct 521 overlaps with only 42 (1.0%) missed. Considering the dynamic programming requiring more than 40 hours to do the job, that we can achieve 99% accuracy in about a minute by a simple transitive relation strategy is significant. We can reconstruct 143 overlaps and miss only 15 (0.4%) overlaps in 2 minutes (in Sparc1000) compared to the overlaps found by fully dynamic programming (DP) run for 40 hours.

The gap of 280 overlaps between dynamic programming and true overlaps, is because the overlap regions between fragments are either having low scores or contain more than 20% errors.

Among all those missed overlaps when $k = 14$, only one overlap proves significant, without which we break the 35k cosmid into two disjoint blocks. This overlap has high errors, even so we are able to reconstruct it by performing block ends comparison.

4 Introduction to Unification Factoring

Unification is the basic computational mechanism in Prolog, and other logic programming languages. A Prolog program consists of an ordered list of rules, where each rule consists of a head with an associated action whenever that rule head matches or unifies with the current computation.

An execution of a Prolog program starts by specifying a goal, say $p(a, X, Y)$, where a is a constant and X and Y are variables. The system then systematically matches the head of the goal with the head of all rules which can be *unified* with the goal. Unification means binding the variables with the constants if it is possible to match them. For example, consider the set of rule heads $p(a, b, c)$, $p(a, b, d)$, $p(a, c, c)$, and $p(b, a, c)$. The goal $p(a, X, Y)$ would match all of the first three rules, since X and Y can be bound to match the extra characters. The goal $p(a, X, X)$ would only match the third rule, since the variable bound to the second and

third position must be the same.

Unification factoring for logic programming was first considered in by Dawson, et.al. [4, 5] who give a dynamic programming algorithm for optimizing the trie size when the strings have an imposed left-right order, as is the case in Prolog programs. Experimental results showed that unification factoring substantially sped up typical Prolog programs. For datalog programs, i.e. Prolog programs without variables, the problem of minimizing trie size was shown to be NP-complete. Lin [9] showed that an augmented version of the trie minimization problem was even harder.

Below, we consider the question of approximation algorithms for unification factoring, i.e. producing a small size trie for a given set of strings. We prove a surprising but negative result, that it is impossible to approximate minimum size trie to within a polynomial factor unless $P = NP$. Along the way, we prove the inapproximability of a new variant of subgraph isomorphism.

5 Inapproximability Results for Unification Factoring

We will relate the problem of unification factoring to the *edge-maximum complete bipartite subgraph* problem. A complete bipartite subgraph defines two disjoint sets of vertices V_1 and V_2 , $V_1, V_2 \subset V$, such that $(v_1, v_2) \in E$ for any $v_1 \in V_1$ and $v_2 \in V_2$. The edge-maximum complete bipartite subgraph of G contains the the maximum number of bipartite edges. i.e. the largest product of $|V_1| \cdot |V_2|$. Edges are permitted to be incident on two vertices either V_1 or V_2 , but they do not contribute to the number of bipartite edges.

The vertex-maximum induced complete bipartite subgraph problem has been shown to be hard to approximate to a polynomial factor by Lund and Yannakakis [10] and Simon [14]. However, these do not resolve the problem of approximating vertex-maximum complete bipartite subgraphs. Note that the vertex-maximum complete bipartite subgraph is easy to approximate to within a factor of two by simply selecting the highest degree vertex of the graph and its neighborhood. In Section 5.1, we prove the inapproximability results for this subgraph problem. In Section 5.2, we use this to demonstrate the hardness of unification factoring.

Proofs will be omitted for space reasons, to appear in the full version of the paper.

5.1 Edge-Maximum Complete Bipartite Subgraph

Consider the following transformation from an arbitrary graph $G = (V, E)$ to a bipartite graph H . H will contain the pair of vertices v_i, v'_i for each vertex v_i of G . For each edge (v_i, v_j) of G , H will contain edges (v_i, v'_j) and (v_j, v'_i) . Finally, H will contain edges (v_i, v'_i) for $1 \leq i \leq n$.

Lemma 1 *If there is a clique C with n_a vertices in G , there must be a complete bipartite subgraph of H with n_a^2 edges.*

Proof: The subgraph formed by inducing $\cup_{i \in C} (v_{i1}, v_{i2})$ for a given clique $C \in G$ is $K_{|C|, |C|}$. ■

Lemma 2 *If there is a complete bipartite subgraph S of n^{1+b} edges in H , then there must exist a clique of n^b vertices in G .*

Proof: The vertices of S can be two-colored, and let b' be the cardinality of the smaller set. Since the cardinality of the larger set is $\leq n$, $b' \geq b$ to realize n^{1+b} edges. By extending S to be a maximal complete bipartite graph, the vertices of b' must define a clique in G . ■

This reduction demonstrates that clique is intimately related to the complete bipartite subgraph problem (CBS), however it does not suffice to show the hardness of approximation of CBS. Note that for small cliques ($\leq n^{1/2}$ vertices) the resulting complete bipartite subgraph may be too small be the largest in H . We must demonstrate that is it hard to find cliques of size $\geq n^{1/2}$.

Lemma 3 *The problem of finding a clique of size $n^{1/2}$ in a graph G containing a clique of size $n^{5/6}$ is NP-hard.*

Proof: The result of [2] demonstrates that it is hard to find a clique within a factor of $n^{1/3}$ times optimal. Thus there exists a d such that finding a clique of size n^d in a graph containing a clique of size $n^{1/3+d}$ is NP-hard. If $d > 1/2$, we are done. Otherwise, consider the graph product $G' = G \times K_n$. This implies that finding a clique in G' of size n^{1+d} in a graph containing a clique of size $n^{4/3+d}$ is NP-hard, where the number of vertices in $G' = n^2$, giving the result. ■

5.2 Minimum-Size Trie

Since the trivial trie for a set of m strings each of length n uses mn edges, we define the *savings* SV of a trie T to be the number of edges saved over the trivial trie, i.e. $SV = mn - |T|$. Thus the optimal trie maximizes the amount of savings.

Theorem 1 *If finding a maximum complete bipartite subgraph from undirected graph G cannot be approximated to within an N^c factor, where N is the number of total vertices in G , then the maximum savings trie cannot be approximated to within a $M^c / \log M$ factor, where M is the number of strings in the trie instance.*

Proof: Consider the following reduction from an input graph $G = (V, E)$ to a set of strings. For each vertex $v_i \in V$, we construct a string s_i of length $n = |V|$ such that for all j , $1 \leq j \leq n$, the j th character of s , $s_i[j] = 1$ if $(v_i, v_j) \in E$; and a unique symbol $\alpha_{i,j}$ otherwise. The set of strings S consist of $\{s_1, s_2, \dots, s_N\}$.

Consider any complete bipartite subgraph of G , defined by disjoint sets of vertices $V_1 = \{v_{j_1}, v_{j_2}, \dots, v_{j_l}\}$ and $V_2 = \{v_{i_1}, v_{i_2}, \dots, v_{i_k}\}$. This subgraph contains bipartite edges $BE = kl$. This subgraph defines a trie with at least $kl/2$ saves, by using the character positions i_1, \dots, i_k as a path from root of the trie. Since all strings $s_{j_1}, s_{j_2}, \dots, s_{j_l}$ share the same value of the

probed characters in common the strings are clustered together through a height of $k + 1$ so the total saves of this suffix tree SV is at least:

$$SV = \sum_{h=1}^k h \cdot (n_{h+1} - 1) \geq k(n_{k+1} - 1) \geq k(l - 1) \geq kl/2$$

where we assume $l > 1$ without loss of generality. Since this holds for any complete bipartite subgraph of G , it holds for the maximum CBS. Let SV_{opt} denote the savings of the optimal trie of strings S , and let BE_{max} denote the number of bipartite edges in the maximum complete bipartite subgraph of G . Then

$$BE_{max} \leq 2SV_{opt}$$

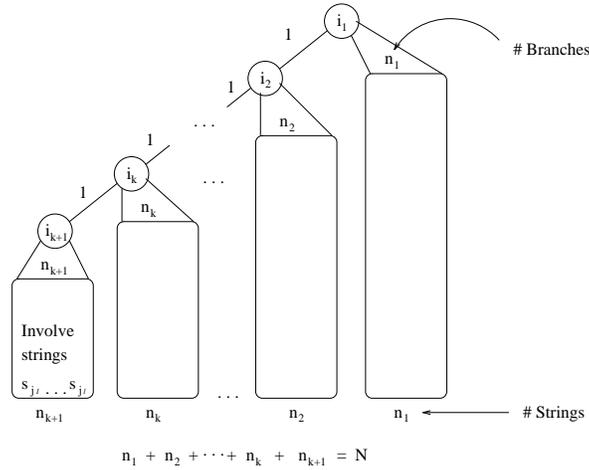


Figure 6: Constructing a suffix tree from complete bipartite subgraph.

Now consider any trie of S with SV saves. We claim that we can construct a complete bipartite subgraph for G containing at least $SV/\log N$ edges.

In any trie for the set S , all savings in the trie must result from a single path from the root, as in Figure 6, because at each probe position the set of strings is broken into singletons except for those containing a 1 at the given position. Once a string belongs to a singleton set, no further saves can be credited to it. Thus the total amount of savings is

$$SV = \sum_{j=1}^k (j - 1)n_j$$

For any j , $1 \leq j \leq k$, the vertex sets $\{v_{i_0}, v_{i_1}, \dots, v_{i_j}\}$ and n_{j+1}, \dots, n_k defines a complete bipartite subgraph G_j , since each string under these branches has symbol 1 at i_0 th, i_1 th, ..., i_j th positions; on the other hand, it means each vertex under these branches is incident to vertices $v_{i_0}, v_{i_1}, \dots, v_{i_j}$. The number of bipartite edges for a given j is

$$BE_j = j \sum_{h=j+1}^k n_h$$

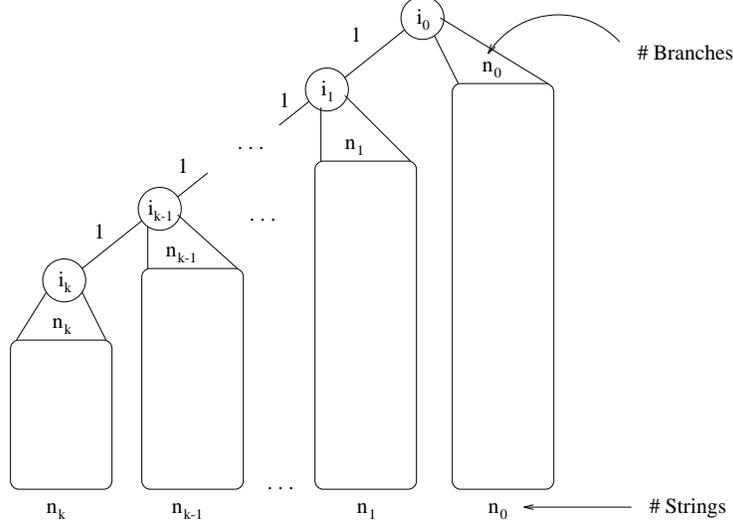


Figure 7: Constructing a complete bipartite subgraph from a suffix tree.

Let r be the value which maximizes this, so $BE_r = \max_j BE_j$. Further

$$j \sum_{h=j+1}^k n_h \leq BE_r$$

and

$$\sum_{h=j+1}^k n_h \leq BE_r/j$$

Thus,

$$SV = \sum_{j=1}^k (j-1)n_j = \sum_{j=2}^k \sum_{i=j}^k n_i \leq \sum_{j=2}^k (BE_r/(j-1)) \leq BE_r \cdot \log N$$

which means there exists a complete bipartite subgraph with at least $SV/\log N$ edges for any trie of S with SV saves.

Since such a subgraph (or r) can be found in linear time, there is an algorithm can approximate the maximum saves trie within $N^c/\log N$ factor, we can approximate the maximum complete bipartite subgraph in N^c factor:

$$BE_{max} \leq 2SV_{opt} \leq SV \cdot (2N^c/\log N) \leq (BE_r \cdot \log N) \cdot (2N^c/\log N) \leq BE_r \cdot 2N^c$$

giving the result. ■

6 Open Problem

Does a constant-factor approximation algorithm exist for the minimum-size trie problem on binary strings?

Acknowledgments

We thank Bill Studier and the rest of the Brookhaven group for interesting discussions on primer walking and sequencing. We thank IV Ramakrishnan for introducing us to unification factoring, and Steve Dawson, Keri Ko, C.R. Ramakrishnan, and Terry Swift for useful discussions.

References

- [1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.
- [2] M Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs, and non-approximability – towards tight results. In *Proc. IEEE 36th Symp. Foundations of Computer Science*, pages 422–431, 1995.
- [3] D.R. Clark and J.I. Munro. Efficient suffix trees on secondary storage. In *Proc. Seventh ACM Symp. on Discrete Algorithms (SODA)*, pages 383–391, 1996.
- [4] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, K. Sagonas, T. Swift, and D.S. Warren. Unification factoring for efficient execution of logic programs. In *2nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 247–258, 1995.
- [5] S. Dawson, C.R. Ramakrishnan, and T. Swift. Principles and practice of unification factoring. In *ACM Trans. on Programming Languages (TOPLAS)*, pages 528–563, 1996.
- [6] M.L. Engle and C. Burks. Artificially generated data sets for testing DNA fragment assembly algorithms. *Genomics*, 16:286–288, 1993.
- [7] P. Green. Documentation for phrap. Genome Center, University of Washington, <http://bozeman.mbt.washington.edu>, 1996.
- [8] J. Kececioglu and E.W. Myers. Exact and approximate algorithms for the sequence reconstruction problem. *Algorithmica*, 13:5–51, 1995.
- [9] C.-L. Lin. Optimizing tries for ordered pattern matching is π_2^P -complete. In *Proc. 10th IEEE Structures in Complexity Theory Conference*, pages 238–244, 1995.
- [10] C. Lund and M. Yannakakis. The approximation of maximum subgraph problems. In *Proc. 20th ICALP*, pages 40–51, 1992.
- [11] U. Manber and E.W. Myers. Suffix arrays : A new method for on-line string searches. *SIAM J. Computing*, 22:935–948, 1993.
- [12] E. W. Myers. Towards simplifying and accurately formulating fragment assembly. *J. Comp. Biol.*, 2(2):275–290, 1995.
- [13] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. In *Proc. Natl. Acad. Sci.*, pages 2444–2448, 1988.
- [14] H. Simon. On approximate solutions for combinatorial optimization problems. *SIAM J. Discrete Math.*, 3:294–310, 1990.
- [15] G.G. Sutton, O. White, M.D. Admas, and A.R. Kerlavage. TIGR assembler: a new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1:9–19, 1995.
- [16] M. S. Waterman. *Introduction to Computational Biology*. Chapman & Hall, London, UK, 1995.