Trie-Based Data Structures for Sequence Assembly (Extended Abstract)

Ting Chen
Steven S. Skiena*
Department of Computer Science
State University of New York
Stony Brook, NY 11794-4400
{tichen—skiena}@cs.sunysb.edu

January 27, 1997

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 for fast fragment assembly. These are a product of our work with Dr. William Studier's group at Brookhaven National Laboratory to build an assembler for sequencing the one-megabase genome of *Borrelia Burgdorferi*, the bacterium which causes 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 product of our work with the XSB logic programming group at Stony Brook.

Although the applications are quite different, a common theme which 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 assembling 3,800 clones of the *Borrelia Burgdorferi* project 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 proves fast enough to eliminate the need for incremental fragment assembly, the original problem with which we began this

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

work. Our times are certainly comparable with such state-of-the-art assemblers as the TIGR assembler [14], 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 fragments generated our program with the overlap graph induced in the final assembly on clean data by biologists. 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 – suffix trees, suffix arrays, and augmented suffix arrays, which shows augmented suffix arrays to be a clear winner over both data structures in both construction and traversal time, and space.

Unification in logic program 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 how the length of common prefixes among the rule heads. Since we have 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 the ith 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

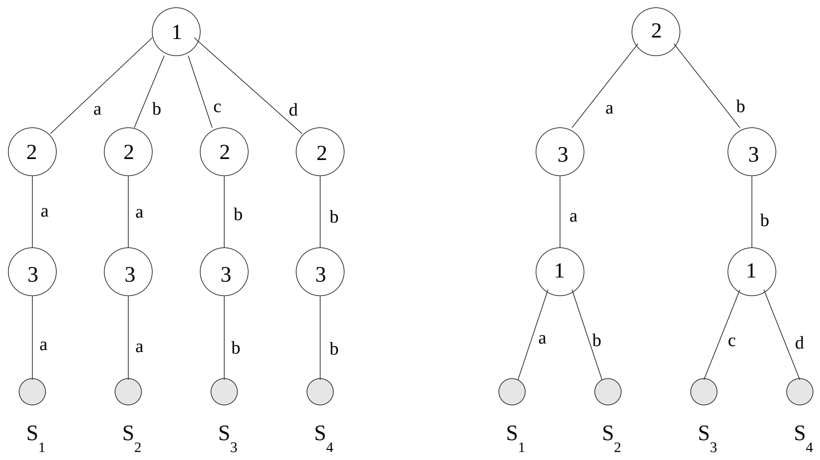

Figure 1: Two different tries for the same set of strings.

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 inapproximatibility results.

2 Overlap Detection for Fragment Assembly

The fragment assembly problem has been characterized as finding a shortest common super-string 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* [11, 7]. 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 that are most reasonablely reflect the relations of their physical maps in the genome to determine their *ordering*. Finally, the consensus stage builds the multiple-alignment of all regions where two or more fragments overlap each other and generates a single consensus DNA sequence.

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 measuring 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 [15] 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 $O(n^2l^2)$ complexity is durable, 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 [11] for a detailed survey. Kececiolglu and Myers [7] 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 then 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 [14], 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 [6] 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 seach 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.

In fact, for real sequencing projects, there are only $O(n)$ true overlaps, for 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 $k = 12$) as a threshold to quickly reject many non-overlapping pairs of fragments.

The potential danger is that sequencing errors may render exact matching too unreliable for overlap detection. One of the main contributions of this paper is a careful experimental analysis of the accuracy 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 various from laboratory to laboratory, with about 2-5% 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 5% error rate yields an average of six errors of this region. Thus there is a very high probability that their must exist an exact match of length ≥ 12. We find that exact matching works well even at higher error rates.

In this paper, we evaluated both suffix tree [2] and suffix array [10] data structures for overlap detection. Suffix array shows a significant advantage over suffix tree, not only on the space but also the speed. We review these data structures. For both, we are given a text string $X = x_1x_2x_3...x_n$, where each x_i is a member of an alphabet Σ, and seek to preprocess X such that given a pattern $P = p_1p_2p_3...p_m$, $(p_i \in \Sigma)$, the set $\{i : x_i...x_{i+m-1} = P\}$ can be found efficiently.

- *Suffix Trees* – the compressed trie for all the suffixes of $X_1, \ldots, X_n$, where $X_i = x_ix_{i+1}...x_n$. A sample suffix tree for the string *aabbaab* is shown in Figure 2. All the edges of the tree, representing a substring of X, can be implemented

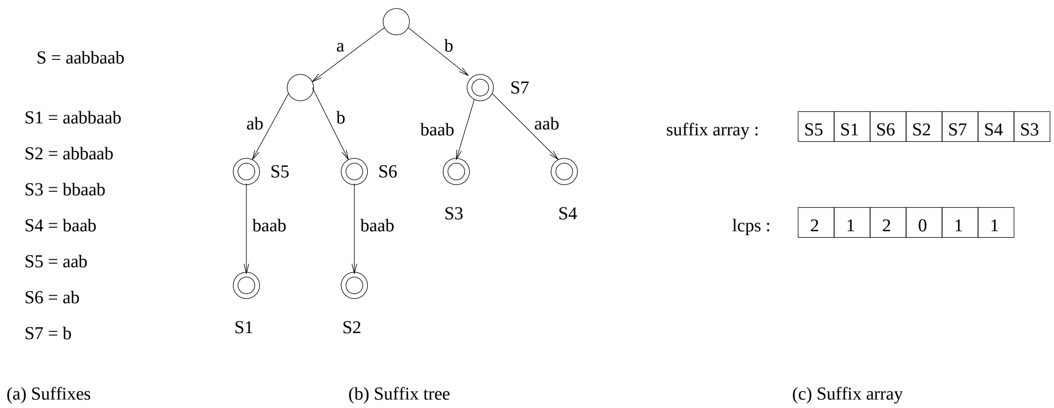

Figure 2: Suffix tree and suffix array for *aabbaab*.

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 [10], 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 size 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.

A second consideration 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 [12] and BLAST [1]. We use such a FASTA-similar approach in our implementation.

3 Exact Matching for Fast Sequence Assembly

Our algorithms for fast overlap detection rely on proper use of the suffix array data structures. Due to space concerns, we will omit discussions of implementation details from this extended abstract, focusing only on experimental results for our program.

We evaluated our overlap detection on the following datasets:

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.

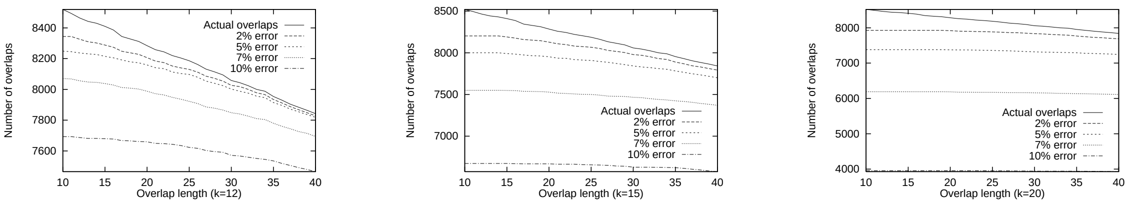

Figure 3: Accuracy of k=12, 15, 20 under errors 2%, 5%, 7%, 10%.

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 [5] 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 into the fragments.

Figure 3 shows how many actual overlaps are detected by exact-matching of length k under different error rates. The first curve in each plot is the number of actual overlaps with different length thresholds from 10 to 40 and 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 length at least 40, as opposed to 5% and 50% for $k = 12$ and $k = 20$. This figure suggests that with high error rate fragments ($> 7\%$), smaller match lengths perform better. With low error rates ($< 5\%$), there is no significant difference between a range of ks, 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.

Table 1 shows how many of the candidate overlaps are detected and how many of them represent true overlaps. A small match-length k will reserve 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 most candidates true and few true overlaps missed. Comparing Table 1 with Figure 3, $k = 14$ and $k = 15$ achieve the best performance.

In addition to the simulated fragments, we 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 bounded dynamic programming and retain them with at least 25 matches and at most 20% errors. Most of the overlaps we miss can be constructed through a simple verification strategy: for any sequence s_t, if there exist two other sequences s_i and s_j both overlapping s_t and the overlapped regions on s_t are overlapped. Thus we use a procedure to reconstruct overlaps without a k length common substrings. We compared the accuracy of our reconstructed data with that of performing a full Smith-Waterman on all pairs of sequences. In Table 2, FN(false negative) gives the number of overlaps we missed comparing to 4320 true overlaps (length ≥ 25) and TP(false positive) represents the amount of overlaps we claim are not overlaps.

Table 2 shows that the overwhelming majority of the overlaps can reconstructed using large exact-matches. Further, all the overlaps we claim after the filtering and reconstruction are correct (ie. there are no false positive errors). This is very

	2% Error		*5% Error*		*7% Error*		*10% Error*	
k	True ovlps /Cands	Pcnt %	True ovlps /Cands	Pcnt %	True overlaps /Cands	Pcnt %	True ovlps /Cands	Pcnt %
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
21	7,870/7,870	100.0	7,216/7,216	100.0	5,860/5,860	100.0	3,366/3,366	100.0
22	7,812/7,812	100.0	7,048/7,048	100.0	5,490/5,490	100.0	2,878/2,878	100.0
23	7,770/7,770	100.0	6,862/6,862	100.0	5,096/5,096	100.0	2,436/2,436	100.0
24	7,694/7,694	100.0	6,630/6,630	100.0	4,724/4,724	100.0	2,040/2,040	100.0
25	7,634/7,634	100.0	6,394/6,394	100.0	4,326/4,326	100.0	1,700/1,700	100.0

Table 1: Accuracy of overlap detection on Borrelia sequence with simulated errors.

	FP Candidates		**FN Candidates**		**FN Filtering**		**FN Reconstruct**		**Time**
k	Ovlps	Pcnt(%)	Ovlps	Pcnt(%)	Ovlps	Pcnt(%)	Ovlps	Pcnt(%)	
12	16,338	378.2	194	4.5	357	8.3	289	6.68	567
13	5,126	118.7	262	6.1	385	8.9	289	6.68	146
14	1,584	36.7	340	7.9	438	10.1	295	6.83	98
15	558	12.9	410	9.5	486	11.3	312	7.22	79
16	100	2.3	460	10.6	526	12.2	312	7.22	70
17	62	1.4	514	11.9	574	13.3	312	7.22	67
18	20	0.46	588	13.6	637	14.7	317	7.34	65
19	12	0.28	654	15.1	699	16.2	322	7.45	64
20	10	0.23	700	16.2	743	17.2	322	7.45	63
21	10	0.23	788	18.2	819	19.0	342	7.92	62
22	2	0.046	854	19.8	873	20.2	344	7.96	60
23	2	0.046	910	21.1	925	21.4	377	8.73	59
24	0	0	940	21.8	955	22.1	379	8.77	59
25	0	0	1,004	23.2	1,017	23.5	435	10.1	59
DP	**0**	**0**	**280**	**6.48**	**280**	**6.48**	**280**	**6.48**	**151,200**

Table 2: Quality of k-mer overlap detection on raw 35kb Borrelia sequence, comparing to 4,320 true overlaps (length ≥ 25).

Data set	Fragments	Base pairs	Space(Bytes/Base Pairs) Suffix Trees Dynamic	Vector	Suffix Arrays
(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	Fragments	Base pairs	Construction Time Suffix Trees Dynamic	Vector	Suffix Arrays
(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	Fragments	Base pairs	Traversing Time Suffix Trees Dynamic	Vector	Suffix Arrays
(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 3: Comparing Suffix Tree and Suffix Array time and space performance on sequence data sets

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 comparing 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.

Table 3 summarizes timing and space experiments on each of the datasets, 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 traversing time for each data structure to be 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)*. The total time involves building a suffix array, detecting overlaps, bounded dynamic programming, and filtering. The final result of the program is a weighted adjacency alignment graph of all the DNA sequence. The accuracy of our program is shown in the next section.

Table 3 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 in traversing is because the out-edges of a node have to be decoded before its children are visited. The suffix array are much more efficient in space, and it will beat the suffix trees in big sequencing project on DNA like human genome. 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 array is it is fast in traversing. The traversing time measures the expense to perform the exact matching. Its linear structure makes this work much easier and faster.

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. [3, 4] 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, ie. Prolog programs without variables, the problem of minimizing trie size was shown to be NP-complete. Lin [8] showed that an augmented version of the trie minimization problem was even harder.

Below, we consider the question of approximation algorithms for unification factoring, ie. 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 inapproximatibility of a new variant of subgraph isomorphism.

5 Inapproximatibility 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. ie. 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 has been shown to be hard to approximate to a polynomial factor by Lund and Yannakakis [9] and Simon [13]. 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 inapproximatibility 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.*

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.*

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.*

5.2 Minimum-Size Trie

Since the trivial trie for a set of m strings each of length n uses mn edges, with define the *savings* SV of a trie T to be the number of edges saved over the trivial trie, ie. $SV = mn - |T|$. 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 jth character of s, $s_i[j] = 1$ if $(v_i, v_j) \in E$; and a unique symbol $\alpha_{i,j}$ otherwise. The set strings S will consist of $\{s_1, s_2, \ldots, s_N\}$.

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

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

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, 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

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

which means there exists a complete bipartite subgraph with at least $SV/\log N$ edges for any trie of S with SV saves. Details appear in the full version of the paper.

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. ∎

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] 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.

[3] 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.

[4] S. Dawson, C.R. Ramakrishnan, and T. Swift. Principles and practice of unification factoring. In *To appear in ACM Trans. on Programming Languages (TOPLAS)*, 1996.

[5] M.L. Engle and C. Burks. Artificially generated data sets for testing dna fragment assembly algorithms. *Genomics*, 16:286–288, 1993.

[6] P. Green. Documentation for phrap. Genome Center, University of Washington, 1996.

[7] J. Kececioglu and E.W. Myers. Exact and approximate algorithms for the sequence reconstruction problem. *Algorithmica*, 13:5–51, 1995.

[8] 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.

[9] C. Lund and M. Yannakakis. The approximation of maximum subgraph problems. In *Proc. 20th ICALP*, pages 40–51, 1992.

[10] U. Manber and E.W. Myers. Suffix arrays : A new method for on-line string searches. *SIAM J. Computing*, 22:935–948, 1993.

[11] E. W. Myers. Towards simplifying and accurately formulating fragment assembly. *J. Comp. Biol.*, 2(2):275–290, 1995.

[12] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. In *Proc. Natl. Acad. Sci.*, pages 2444–2448, 1988.

[13] H. Simon. On approximate solutions for combinatorial optimization problems. *SIAM J. Discrete Math.*, 3:294–310, 1990.

[14] 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.

[15] M. S. Waterman. *Introduction to Computational Biology.* Chapman & Hall, London, UK, 1995.