A Learning Algorithm for String Assembly

Mark K. Goldberg, Darren T. Lim, and Malik Magdon-Ismail
Rensselaer Polytechnic Institute

Abstract

We present a supervised learning approach to DNA shotgun sequencing. The oracle (supervisor) is a set of already-sequenced DNA strands; the output of the learning process is a domain-specific algorithm for sequence assembly. Our goal is to learn a fast algorithm for a given problem domain. Our approach is to begin with a parameterized form of a sequencing algorithm and to then *learn* the optimal parameter values, numerical and combinatorial, for the given domain of interest.

We present experimental results using DNA strings from H. pylori and humans, and compare the results from real DNA with results obtained from random strings. Our experiments demonstrate that considerable gains in speed can be achieved without significant loss in accuracy. Further, the resulting algorithms learned on different input domains are distinct, illustrating the value of developing domain specific algorithms. We also present experimental results on the applicability of the algorithms learned on one domain to the other domains. These results have implications for the sequencing of new DNA-strings given already sequenced DNA-strings.

1 Introduction

The *String Assembly Problem*, *SAP*, is to construct a superstring, S, for a given finite collection $\mathcal{C} = \{s_1, s_2, \ldots, s_n\}$ of strings over an alphabet Σ with the requirement that S contain every s_i as a substring. The members of $\mathcal{C}$ are often called *fragments*. A trivial and usually uninteresting solution is to concatenate the fragments to get $S_{conc} = s_1 s_2 \cdots s_n$, thus, one often imposes additional constraints. Using the notation $l(s)$ to denote the length of a string, if the additional constraint is to minimize $l(S)$, then one arrives at the *Shortest Superstring Problem, SSP*, which is proved to be *NP*-hard ([2, 9]). The length-constraint would be useful only if the target-superstring is expected to be the shortest one. In biological applications, this is usually not the case.

Interest in designing efficient algorithms for *SAP* has mainly been prompted by problems in data compression ([2, 11]) and computational biology ([5, 4, 13]). A combinatorial solution to the *DNA sequencing problem* has been a topic of research for over a decade. The early approaches ([3, 10]) that use classical biological techniques are comparatively slow making these methods impractical for large problem sizes. Current approaches to DNA-sequencing often employ *shotgun sequencing* techniques, which are based on the greedy merging strategy. At each step of the greedy algorithm, one merges the two fragments with the largest overlap. This procedure is repeated until the remaining fragments no longer overlap. It is well known that, at some point, when the overlaps become "short" (and the number of remaining fragments "small"), the greedy strategy begins to fail[1] and merges fragments that are actually disjoint substrings in the target superstring.

[1] The greedy algorithm also fails to solve *SSP*; see [4] for references.

Hence, a deviation from the pure greedy strategy is necessary to achieve the correct assembly. An option is to consider multiple merging sequences, possibly including the greedy one. In this case, one needs to solve the problem of determining which of the resulting candidates is "correct." In general, the determination of the "correct" superstring (the target) is a non-trivial problem. For DNA-assembly, one often uses additional information coming from various sources, such as *STS maps* and *clone-libraries.* Other applications may use different means of determining the target superstring. Hence, it would be useful to have a general approach that formalizes such additional information and outputs explicit algorithms for string assembly. The final assembly algorithm only need be successful at assembling fragments originated from a target superstring chosen from a particular domain of interest. To this extent, one would like to be able to encode information about the target domain into the resulting algorithm. Thus, we restate the problem as follows.

Plausible String Assembly Problem (PSAP): Given a collection of strings, $\mathcal{C} = \{s_1, \ldots, s_N\}$, construct a **physically plausible** superstring.

To understand what physically plausible means, it is necessary to consider the physical process that created the fragments. For biological applications such as *DNA* assembly, this physical process could be modeled in the following way.

1. The input to the physical process is a string $\mathcal{S}_T$ from a given *input domain*, $\mathcal{I}$. Often, we have information about the input domain other than what is contained in the set of fragments $\mathcal{C}$. For example, if we have available STS markers, then we know that the only possible target strings are those in which the STS markers occur in the correct order.

2. λ copies of S_T are then fragmented by a *fragmenting process*, $\mathcal{P}$, to yield[2] the set of fragments $\mathcal{C}$; λ is usually called the *cover ratio.*

$$\underbrace{\{S_T, S_T, \ldots, S_T\}}_{\lambda \text{ copies}} \xrightarrow{\mathcal{P}} \mathcal{C} = \{s_1, s_2, \ldots, s_n\}$$

$$\textit{Input Domain, } I \longrightarrow S_T \xrightarrow{\mathcal{P}} \mathcal{C} \xrightarrow{\mathcal{P}^{-1}} S$$

The task of *PSAP* is to construct an algorithm that represents the inverse of the fragmenting process, $\mathcal{P}^{-1}$. Since different strings could yield the same observed collection of fragments, the "correct" superstring (the target) should be picked so that the pair $\{S, \mathcal{C}\}$ is consistent with the additional information and with the physical process that created $\mathcal{C}$ from S. Thus, the following two plausibility criteria need to be enforced.

i. (Prior Information) S should be a *probable* candidate from the input domain.

ii. (Physical Consistency) It should be *probable* that the fragmenting process $\mathcal{P}$ would yield $\mathcal{C}$ when applied to S, in particular, $\lambda l(S) \geq \sum_i l(s_i)$, where the cover ratio[3] λ is assumed to be known.

A natural approach to incorporating information about the problem domain is to *learn* how to solve the problem from solutions to *known*, representative examples drawn from this domain. If the *new* instances that we eventually want to solve are "similar" to the instances we learned on, then we expect to be successful. This approach is adopted in our paper: we construct an algorithm

[2]We assume that the fragmenting is done by making a cut at each position i of S_T with a fixed probability p.

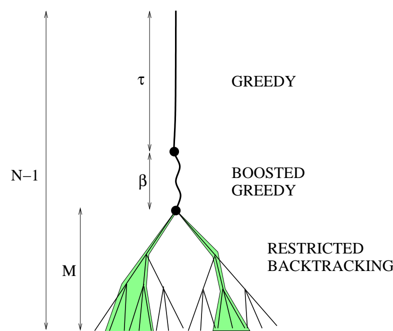
[3]Since it is possible that some fragments are lost in the fragmenting process (due to practical considerations), some parts of S_T may not be represented in $\mathcal{C}$, which leads to the existence of gaps.

for $\mathcal{P}^{-1}$ by learning, in a supervised manner, from a *training set* of solutions to instances drawn from the input domain of interest. Our first step is to develop an algorithm for generating such a training set. We use a technique that we call *inverse algorithm engineering* (to be discussed in detail in the next section) to generate the data for learning $\mathcal{P}^{-1}$. Since we learn from examples drawn from the domain of interest, our final algorithms will be domain specific. Further, any prior knowledge about this domain and about the nature of the fragmenting process can be built into the learning algorithm. Our approach is quite general; however, for the purpose of illustration, we present our algorithms in the context of DNA sequencing. Our computational experiments indicate that accurate sequencing can be achieved in a computationally tractable way using our learning approach.

The summary of the remainder of the paper is as follows. First, we present a description of the learning approach, along with the method for data generation. The following section describes the input domains used in the initial experiments and a modified model which conforms more realistically to the currently-practiced DNA-sequencing process. Finally, we present the experimental methodology used and the results of the simulations. We test our learning approach by evaluating the performance of the learned algorithms both on the domains for which they were designed, as well as on different domains.

2 Parameterized Assembly Algorithms.

In addition to pure greedy merges, we introduce *boosted greedy* merges. Given an ordered pair of fragments (f_1, f_2), we use $c(f_1, f_2))$ to denote the longest overlap[4] of the two strings. We say that a third fragment g *certifies* the merge $\{f_1, f_2\}$, if either $c(c(f_1, f_2), c(f_2, g))$ or $c(c(g, f_1), c(f_1, f_2))$ is non-zero. A *boosted greedy step* merges the certified pair with the largest overlap. Searching for certificates makes boosted greedy merges more costly than greedy merges, however, boosted greedy merges have a higher probability to be true than pure greedy ones. Both, greedy and boosted greedy merges, can be performed efficiently.



It is observed that for many applications, greedy initial steps tend to be correct. Our experiments demonstrate that a number of boosted greedy merges are then usually correct, even after pure greedy merges are no longer so. However, after both types of greedy merges are no longer reliable, a more extensive search is warranted. This prompted us to consider a generic assembly algorithm comprised of the following three stages (also illustrated in the figure above).

Stage 1. Greedy. τ of the initial merges are greedy.

Stage 2. Boosted greedy. β of the following merges are boosted greedy.

Stage 3. Restricted backtracking. After the boosted greedy stage is completed, for the remaining M merges, the algorithm outputs a list of possible superstrings developed using a partial enumeration of the set of all possible merging sequences.

[4]Here, when f_1 and f_2 are merged, a suffix of f_1 is identified with a prefix of f_2.

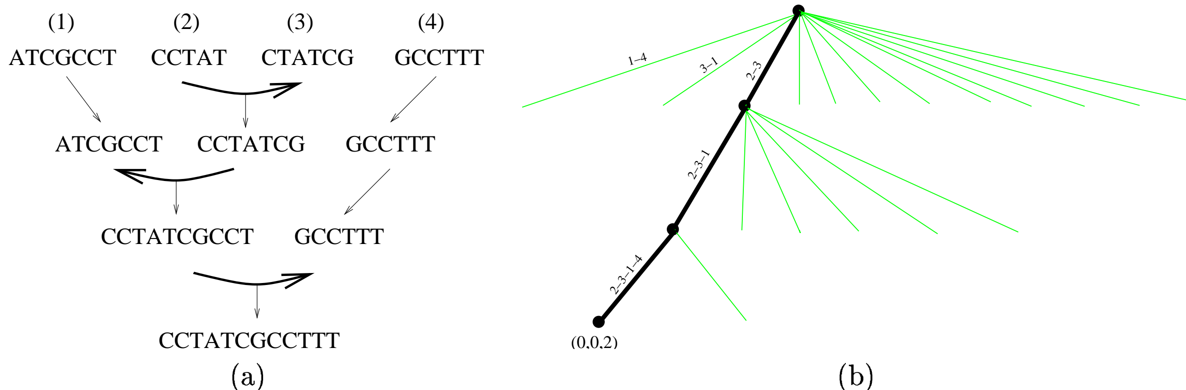


Figure 1: An example assembly problem and its associated assembly tree. The merges are ordered according to the length of the merge and then according to the length of the resulting merged fragment with ties being broken lexicographically.

The third stage performs an "intelligent" partial enumeration of superstrings consistent with the result of the previous stages. The language we use to describe this enumeration is that of backtracking coordinates. An arbitrary assembly proceeds in a sequence of steps, each step consisting of a merge between two fragments (in some cases, it is their concatenation) for a total of $M-1$ merges. (see the example in Figure 1 (a)). The process can be visualized using the *assembly tree* (see Figure 1 (b)). The first step is to pick one of the possible $M(M-1)$ merges (remember that $s_i s_j$ and $s_j s_i$ are two potentially different merges). We rank each merge by the length of the overlap and then sort the the pairs in the non-increasing order of the ranks, breaking ties by the indices of the fragments. The root node of our tree is the collection of M fragments. Each of the possible ordered $M(M-1)$ merges gives rise to $M(M-1)$ branches leading to potentially different collections of $M-1$ fragments, and these form the first level nodes. This process is recursively continued to generate the entire assembly tree, the leaves of which are fully assembled candidate superstrings. A path from the root to a leaf is a sequence of merges, and can be indexed by the branches traversed, with the left-most being branch 0. The vector of these indices forms the backtracking coordinates of the leaf; see Figure 1 (b).

If the parameters τ, β, and a set $\mathbf{b} = \{(b_1, b_2, \ldots)\}$ of backtracking coordinates are fixed, the generic three-stage algorithm becomes a well defined algorithm for the assembly problem. Our *learning model* for sequence assembly is a collection of such algorithms. After learning is done, we will have picked a specific member from our learning model, based upon the training set (to be described shortly). Experiments show that the optimal values of these parameters can be widely different for different input domains. Furthermore, they change for different fragmenting processes, $\mathcal{P}$. Thus, any single specification of the parameters would likely yield an algorithm which, for a given input domain may or may not be accurate in terms of consistently assembling the fragments into the correct target superstring. We solve the parameter specification problem by designing a *learning algorithm* that extracts a set of parameters suitable to the given domain from learning data.

The learning algorithm we develop is an expansion of that in [7], and we follow the methodology of using supervised learning for constructing efficient and accurate optimization algorithms originated in ([1, 8, 6]). We operate within the PAC[5] learning paradigm [12] which assumes the availability

[5]Probabilistically Approximately Correct.

of an *oracle* algorithm $\mathcal{O}$ and an *instance generator* $\mathcal{G}$ that are used to generate training data. For our problem, we need a training data set of the form

$$\mathcal{D} = \{\mathcal{C}_i, \tau_i, \beta_i, \mathbf{b}_i\}_{i=1}^{T} \quad (1)$$

where each $\mathcal{C}_i$ is a collection of fragments generated from the domain of interest, the correct values for τ, β, M and $\mathbf{b}$ are provided by the oracle and T is the number of training samples. Using $\mathcal{D}$, one then learns a value (or set of values) for the parameters that yield a good performance on this input domain.

3 Inverse Algorithm Engineering

```
Procedure Generate_Data
{ for (i = 0; i < T; i++)
  {
   Generate target superstring S_{T_i} ∈ I;
   Generate the fragments C_i using P;
   Obtain # of true greedy merges (τ_i);
   Obtain # of true boosted greedy merges (β_i);
   Obtain # of remaining merges (M_i);
   Reorder remaining merges by overlap length;
   Obtain the backtracking coordinates(b_i)
       of S_{T_i} according to this reordering.
   Store (C_i, τ_i, β_i, M_i, b_i)
  }
}
```

The success of our learning approach relies on the ability to efficiently generate a data set for learning the process $\mathcal{P}^{-1}$. Unfortunately, an oracle that could be used to construct a solution to an assembly problem (such as *SSP* or *PSAP*) for particular instances in the training set would usually entail solving an NP-hard problem, hence learning would be infeasible. However, we can use a trick: we can use the inverse of the process we are trying to learn, namely $\mathcal{P}$, to generate the data. Instead of generating a collection of strings and asking the oracle to construct the shortest or most plausible superstring, we start by generating the superstring S_{T_i} from the given input domain (say *DNA*), and then generate the collection $\mathcal{C}_i$ of substrings of S_{T_i}, according to the process $\mathcal{P}$. S_{T_i} is then used as the target[6] superstring for solving *PSAP* on $\mathcal{C}_i$. One can use the known target superstring, S_{T_i}, to extract $\tau_i, \beta_i, M_i, \mathbf{b}_i$ as follows. One performs greedy merges until a greedy merge leads to a fragment that is not a substring of the target (obtaining τ_i). One then repeats this procedure with boosted greedy merges (obtaining β_i). Finally $M_i + 1$ fragments remain. We use the following ordering procedure at each level to construct the assembly tree for the remaining M_i merges. First, order the fragments according to their lengths, breaking ties lexicographically. The longest fragment is chosen and the $2M_i$ possible merges with it are ordered according to overlap length. Thus, we have a systematic ordering for the branches in the assembly tree, and, so, the M_i dimensional backtracking coordinates of the target can be obtained ($\mathbf{b}_i$). Note that every merge in the restricted backtracking stage is obtained by merging the longest remaining fragment with one of the other fragments. Consequently, it is computationally easy to develop the learning data set $\mathcal{D}$, hence the learning approach is at least feasible.

For *DNA* sequencing, the work done by *TIGR*, *Celera*, the *Human Genome Project* have made available to us a broad range of input DNA strands. We used various samples from these strings

[6] In general, u is not necessarily the shortest superstring for the collection S_{T_i}; however, it appears, that for all applications we tried, it is close to the shortest.

to model sequences coming from *DNA*, and for comparison, we also use random strings to create instances from a random domain. In the next section we describe in greater detail the learning algorithm and the specific approach to restricting the components of the backtracking vector, $\mathbf{b}$, so that the learned algorithm will be efficient, without much loss in accuracy.

4 Algorithms

We now present the details of the learning algorithm to determine an appropriate set of parameters for a generic *PSAP* problem. Once these parameters have been fixed, they can be used in a final assembly algorithm as shown on the right. Procedure `Initialize` determines the $N \times (N-1)$ table of overlaps for the initial fragments. Fragments that are substrings of other fragments are removed in this process. Procedure `Greedy` performs τ greedy merges. Typically, increasing τ boosts the efficiency of the assembly algorithm, at the expense of accuracy. Procedure `Boosted_Greedy` performs β boosted greedy merges. The remaining merges are performed by procedure `Restr_Backtracking`. At the end of the boosted greedy stage, the remaining overlaps are small, making it difficult to distinguish true overlaps from false ones. At this stage, finding the target superstring requires that we observe the results of different sequences of merges. These merge sequences are represented in the set of backtracking vectors $\{\mathbf{b}\}$, that yield the candidate superstrings.

```
procedure Assembly
{  Initialize();
   Greedy(τ);
   Boosted_Greedy(β);
   Restr_Backtrk({b});
}
```

Figure 2: Final assembly algorithm.

We use a simple yet safe learning algorithm to obtain β and M. For a given assembly problem, we define the *Non-Greedy Height, (NGH)*, as the number of fragments remaining once stage one ends; we also define the *Boosted-Greedy Height, (BGH)*, as the number of fragments remaining once stage two ends. For M, we pick the largest number of remaining merges that were needed in the restricted backtracking stage, and for β we pick the largest number (over all the training instances) of remaining fragments once stage one ends minus M.

$$M = \max_i M_i \qquad \beta = \max_i \beta_i \tag{2}$$

τ is then defined implicitly by knowing β and M.

Experimentally, we find that $M \sim 50$, hence an exhaustive search is not practical for the restricted backtracking stage. However, our experiments show that most of the available merge sequences have very low probability of being successful. Thus, we should be able to narrow down considerably the possible choices for $\mathbf{b}$. Once again, we use a simple yet safe learning algorithm as follows (we follow a strategy that was introduced in [6]).

Having determined M, some of the samples will have $M_i < M$, in which case their backtracking coordinates will have fewer than M components. These are augmented by prefixing the necessary number of 0's to the vector to get an M component vector for every sample. This database of backtracking coordinates for every sample is used to eliminate from consideration the branches of the backtracking tree that are unlikely to yield an optimal superstring. We define a partial ordering ($<$) in the space of backtracking coordinates by $\mathbf{x} < \mathbf{y}$ iff $\mathbf{x}_i \leq \mathbf{y}_i$ for all $i = 1, \ldots, M$, and we say that $\mathbf{x}$ is *dominated* by $\mathbf{y}$. To complete the parameter specification, all that remains is to specify the set of backtracking coordinate vectors, $\{\mathbf{b}\}$, used as input to procedure `Restr_Backtracking`.

For this set, we will use the set of all backtracking vectors that are dominated by at least one backtracking vector[7] in our database, thus

$$\mathbf{x} \in \{\mathbf{b}\} \iff \mathbf{x} < \mathbf{b}_i \text{ for some } \mathbf{b}_i \text{ in the learning database} \tag{3}$$

The final assembly algorithm is then given by Figure 2 with the learned parameters β, M, $\{\mathbf{b}\}$ (τ is determined implicitly). The input domain and the fragmenting process play a role in developing the domain specific assembly algorithm (through their determination of the training data). The final algorithm may be applied to any assembly problem, however, the best performance should be expected when applied in the domain for which it was developed. The number of training samples is a heuristic that we determined empirically, largely based upon practical considerations. One further issue needs to be discussed: once we have an assembly algorithm that outputs a set of candidate superstrings, how do we pick one as the target? Measures for doing this will be developed and compared in the next section.

5 Experiments

We now present the experimental parameters and the results of testing. We used a random input domain and the domains defined by two DNA strings collected from genomic databases. Strings created from the random domain **R**, called *Random*, are generated by a procedure which selects characters of the string at random according to the uniform distribution. The other domain, termed *DNA*, uses sequenced DNA strands as the source for target superstrings. The strands of DNA that are used for the experiments presented here are H. pylori, and Human Chromosome 22. The length of the former is close to 0.5 million base pairs and that of the latter is close to 32×10^6 characters in length. Substrings of H. pylori form the domain **P**; the substrings of the Chromosome 22 form the domain **H**.

We have two different criterion for picking from among our candidates. The first is *length*. The second is a purely statistical measure, using the following procedure:

- Given a set of fragments, calculate the frequency distribution of each character in regards to the entire system of fragments. Call these values $\Delta_A, \ldots, \Delta_T$.
- For every candidate in $\mathcal{C}$, calculate the frequency distribution of each base. Call these values $\delta_A, \ldots, \delta_T$.
- Choose the candidate which minimizes the differences between the frequency of the fragments and its own frequency. That is, find the candidate that minimizes the criterion function

$$\kappa = \sum_{a \in A} (\delta_a - \Delta_a)^2$$

In our experiments, the substrings were generated with lengths that range from 100,000 to 500,000 at intervals of 50,000. All experiments were performed on Ultra-SPARC 10 workstations running

[7] This condition eliminates the need for explicitly enumerating all vectors of the database.

Solaris 5.8. For all domains, random and DNA-based, 100 randomly selected substrings were used as superstrings for learning. An additional 100 strings were used as targets for testing purposes. Cover ratios used for are fragment systems were 5.0 and 6.0; they were chosen because of the efficient and accurate databases produced by our algorithms.

Every version of `Assembly` was tested on the domain that was used for its training, but also on "wrong" domains. During testing, the algorithms were given the collections of fragments, but not direct knowledge of the superstrings, as was the case for learning. The knowledge of the superstrings allowed us to evaluate the approximation ratios of the algorithms by comparing the results found by the algorithm with the lengths of the corresponding superstrings. Since the experiments showed no significant difference in the performance of `Assembly` learned on the three random domains, we show here only the results of experiments related to domain *Random*.

The two main objectives of the experiments were the accuracy of the algorithms on the domains and the running time measured in terms of the volume of the database of coordinates. In terms of the growth of the running time, the statistics were very similar for both the length criterion and the statistical measure. At this time, we present only the results for the statistical measure, and leave the length criterion data for the full paper.

The accuracy of learning is measured by an **approximation ratio**. For the shortest superstring criterion, we compute from the experiments the approximation ratio according the following formula:

$$\frac{1}{N}\sum_{i=1}^{N}\frac{length(T_i)}{length(u_i)},$$

where N is the number of test-inputs (in our case $N = 50$); u_i is the ith superstring used for testing; T_i is the superstring constructed by `Assembly` for the ith test. When applying the approximation ratio to gapped fragment systems, $length(u_i)$ is the length of the portion of ith superstring covered by the fragment system $\mathcal{F}_i$.

The accuracy results for both criteria are shown in Table 1. The results for the accuracy of assembly for gapped fragment systems includes the information of the approximation ratio and the number of times out of 100 trials that the best fit set of meta-fragments according to our feasibility test were in fact direct substrings of the original target superstring. Since we are dealing with sets of fragments the approximation ratio compares the average length of the concatenated set of fragments to the total length of the covered regions of the original target superstring. Similar to the previous tests, we are interested in testing a learned database with a different input domain. In this case, we have tested the Human database with the Pylori input domain, shown in table 1.

Table 1: Accuracy of Assembly (length=500,000)

	Statistical Criterion				Length Criterion			
	H/H	**P/P**	**H/P**	**P/H**	**H/H**	**H/P**	**R/H**	**R/R**
5.0	1.007|92	1.003|95	1.009|84	1.008|88	1.003|45	1.005|38	1.0191|28	1.001|48
6.0	1.006|92	1.003|97	1.007|89	1.008|92	1.002|45	1.006|36	1.0165|27	1.001|49

Every column of the table is labeled with two symbols that show the learning domain (left) and the testing domain (right). The two numbers of each entry in the table show the accuracy (left) and the number of tests for which `Assembly` constructed the initial superstring (right) for input lengths of 500,000. The data show that the H-trained `Assembly` performs significantly better on domains H and P than the Random-trained `Assembly`.

For our statistical criterion, we use the Human Chromosome 22 and H. Pylori domains to create databases from gapped fragment systems. Figure 3 shows the growth of the non-greedy heights, $O(n^{2.1})$ and $O(n^{1.81})$ respectively for Ratios 5 and 6 of the Human Chromosome 22 domain. Figure 4 shows the growth of the boosted greedy bounds, $O(n^{2.07})$ and $O(n^{1.95})$ respectively for Ratios 5 and 6. Figure 5 shows the growth of the database. The growth of the database volumes is at rates $O(n^{3.09})$ and $O(n^{2.96})$. For H. Pylori, we see that its non-greedy height bounds grow at rates $O(n^{1.65})$ and $O(n^{1.19})$ respectively for Ratios 5 and 6. Its boosted greedy bounds grow at rates $O(n^{2.97})$ and $O(n^{2.84})$

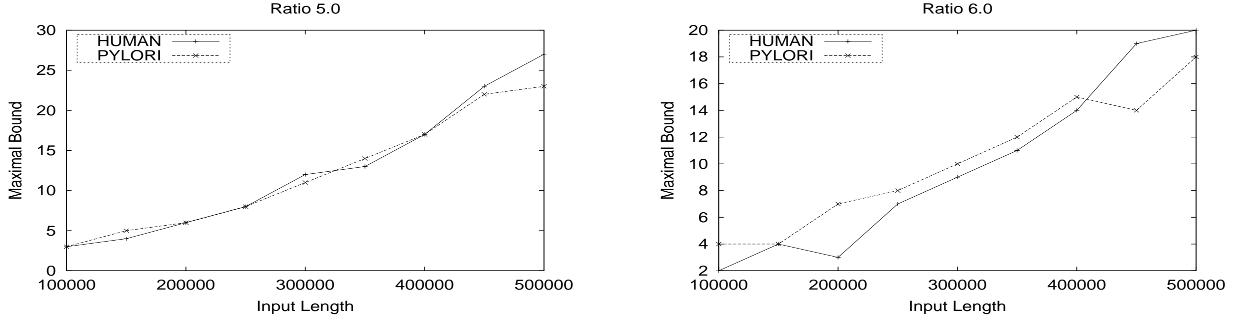


Figure 3: Maximal Non Greedy Height Bound for Ratios 5.0 and 6.0

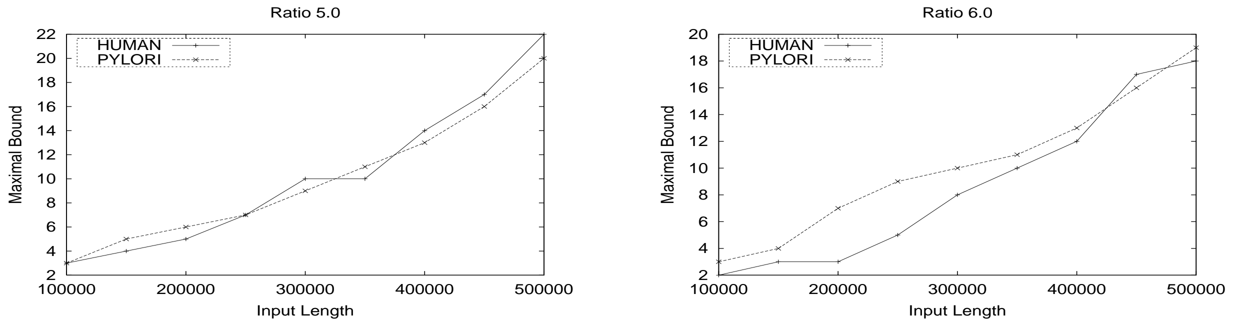


Figure 4: Boosted Greedy Bounds for Ratios 5.0 and 6.0

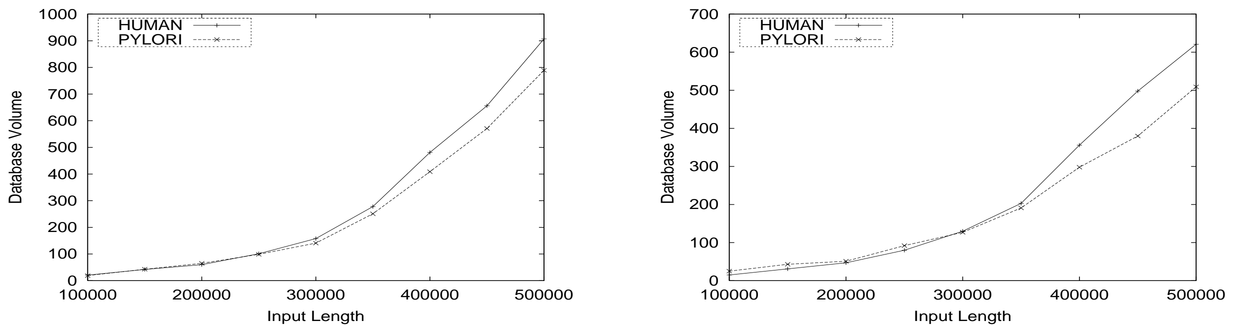


Figure 5: Database Volume for Ratio 5.0 and 6.0

References

[1] E. A. Breimer, M. K. Goldberg, and D.T. Lim, "A Learning Algorithm for the Longest Common Subsequence Problem" *Proceedings of Algorithms and Experiments (ALENEX00)*, pp.96-105, 2000.

[2] J. Gallant, D. Maier, J. Storer, "On finding minial length superstrings," *J. of Computer and System Sciences*,20(1) pp. 50-58, 1980.

[3] W. Gilbert, A. Maxam, "A new method for sequencing DNA," *Proc. Natl. Acad. Sci.*, 74, pp. 560-564, 1977.

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

[5] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, "Biological sequence analysis", Cambridge University Press, 1998.

[6] M. Goldberg and D. Hollinger, "Designing Algorithms by Sampling", Proceedings of *Algorithms and Experiments,* Trento, Italy, Feb. 9 -11 1998 (to appear in *Discrete Applied Mathematics*).

[7] M. Goldberg and D. Lim, "*A Learning Algorithm for the Shortest Superstring Problem*," in Proceedings of Atlantic Symposium on Computational Biology, Genome Information Systems & Technology, pp. 171 - 175. (2001).

[8] M. Goldberg and R. Rivenburgh, "Constructing Cliques Using Restricted Backtracking," in D. Johnson and M. Trick, (*eds.*) *DIMACS Series in Discrete Mathematics and Theoretical Computer Science: Cliques, Coloring, and Satisfiability, Second DIMACS Computational Challenge* Volume 26, American Mathematical Society, pp. 75-88 (1996).

[9] D. Maier, "The complexity of some problems on subsequences and supersequences" *J. of ACM*, 25, pp. 322-336, 1978.

[10] F. Sanger, S. Nicklen, A. Coulson, "DNA sequencing with chainterminating inhibitors," *Proc. Natl. Acad. Sci.* 91, pp. 3072-3076, 1974.

[11] J. Storer, T. Szymanski "Data Compression via textual substitution," *J. ACM*, 29, pp. 928-951, 1982.

[12] L. G. Valiant, "A theory of the learnable," *Communications of the ACM,* 27(11), pp. 1134-1142, 1984.

[13] M. S. Waterman, *Introduction to Computational Biology*, Chapman and Hall, 1998.