

A Learning Algorithm for the Shortest Superstring Problem

Mark K. Goldberg and Darren T. Lim
Rensselaer Polytechnic Institute

Abstract

The Shortest Superstring Problem, *SSP*, asks for a shortest string which contains each string from a given set of strings. The problem arises in DNA-sequencing and data compression. The unique importance of *SSP* for DNA-sequencing justifies an attempt to develop efficient domain-specific algorithms for the problem. We present a learning algorithm which outputs an *SSP*-procedure that is efficient and accurate on the domain for which it is learned. Learning is accomplished through “inverse algorithm engineering,” which makes possible to avoid (for learning) the NP-hardness of the underlying problem. We experimentally test the learning algorithm and the evolved algorithms on the input domains originating from various DNA sequences and random domains. The experiments show that learning yields efficient algorithms for *SSP* that are more than 0.999-accurate. The data suggest that a successfully sequenced DNA strand can be used to significantly speed up the sequencing of other DNA strands.

1 Introduction

The instance of the *Shortest Superstring Problem*, *SSP* (sometimes termed *SCS* ([14]), for *Shortest Common Superstring*), is a finite collection $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ of strings written in a finite alphabet. The problem is to construct a shortest string u such that every s_j appears in u as its substring. *SSP* is known to be NP-hard ([14]). A number of approximation algorithms with a fixed approximation ratio have been developed that use different variations of the greedy strategy ([1],[6],[7],[13]).

SSP is closely related to DNA-sequencing, which aims to sequence a strand of DNA from a set of fragments whose locations in the strand are not known. Practical methods for DNA-sequencing often employ *shotgun sequencing*, which is essentially the greedy merging strategy. The pairwise overlaps of the fragments are identified, and the pair with the largest overlap is merged accordingly, since long overlaps are usually accurate, or *true*.

The procedure is then repeated a number of times. After a number of merges, the length of the largest remaining overlap becomes small and the greedy strategy fails, *i.e.* it occasionally merges two fragments that do not overlap as intervals of the DNA strand in question, even though their overlap as strings is currently the largest. To avoid *false* merges, the greedy strategy is terminated when false overlaps are expected to occur; this decision is often based on the sizes of the overlaps and/or the number of remaining fragments. The process is then completed using biology-based methods.

In this paper, we describe a new, purely computational strategy for replacing the ending of the shotgun assembly. Our programs were tested on various DNA-strings and strings generated by two different random string generators. The experiments show that the new procedures are accurate and computationally efficient. The new strategy consists of two parts:

- *Boosted greedy.* The standard greedy algorithm is supplemented with a certification procedure. The pairs of fragments are checked by the certification procedure and the pair with the longest overlap which passes the certification is merged.
- *Restricted backtracking.* After the boosted greedy stage is completed, the end of the sequencing can be done using an exhaustive search (almost always infeasible). Our experiments demonstrate that optimal or close to optimal solutions can often be found using restricted backtracking, which is practically feasible.

The strategy outlined above raises three crucial questions: When do you halt the pure greedy stage and start the boosted greedy stage? When do you halt the boosted greedy stage and start the restricted backtracking? What kind of restriction should be employed to make the third stage tractable, yet accurate? The answers to these questions are widely different for different input domains. Furthermore, the answers change for different *cover ratios* of fragment collections. The cover ratio of a collection of fragments is defined as the sum of the lengths of the fragments divided by the length of the shortest superstring.

Thus, we can expect that any “singular” specifications of the parameters that answer the three questions above would likely yield an algorithm which, for some input domains, is either inefficient or inaccurate. We solve the problem by designing a *learning algorithm* which consults an oracle for the “best” answers for a given input domain. We follow the methodology of using supervised algorithm learning for constructing efficient and accurate optimization algorithms developed in [3, 11, 5]. Our learning model is essentially PAC¹ learning [17]. It assumes the availability of an *oracle* algorithm \mathcal{O} , whose solutions are used to create a new algorithm for a given input domain. The domain is given by an *instance generator* \mathcal{G} ; it is assumed that \mathcal{G} generates instances according to a predefined probability distribution Γ . For each instance generated by \mathcal{G} , the answers to the questions above provided by \mathcal{O} are added to a database, which is processed to output an algorithm for *SSP*. The performance of the evolved algorithm is tested on the instances that are generated according to Γ , although the evolved algorithm can be applied to an arbitrary input to *SSP*. The “quality” of learning is then measured by the number of applications of \mathcal{O} needed to achieve predefined performance parameters of the evolved algorithm.

The oracle for learning an *SSP*-algorithm is designed through *inverse algorithm engineering*, which avoids the NP-hardness of *SSP*. Instead of generating a collection of strings and asking the oracle to construct the shortest superstring, we start by generating a superstring u from a given input domain, then select a cover ratio and generate a random collection \mathcal{S} of substrings of u , according to the preselected cover ratio. Since u is a superstring for \mathcal{S} , the oracle can quickly check for correctness every merge done by the evolved *SSP*-algorithm. In particular, it can quickly determine the “degree of non-greediness” in the third stage of the *SSP*-algorithm. Consequently, it is computationally easy for the oracle to develop the database of backtracking coordinates that describes the search area for the third stage of the evolved *SSP*-algorithm.

The learning strategy outlined above can be applied to the domains generated by the superstrings that are already sequenced. Thus, the practical question is whether a strategy learned on one domain is successful for a collection of fragments generated from another DNA-string; in other words, whether the difference between the domains can cause a substantial difference between the “search areas” of the respective domains. Our experiments showed that such a difference exists between random strings and DNA-strings. On the other hand,

the difference in the search spaces generated by different DNA-strings is not significant to cause a failure of the evolved *SSP*-algorithm when it is applied to a “wrong” input. Thus, it appears that a successfully sequenced DNA strand can be used to develop an efficient search for sequencing other DNA’s. In particular, the search developed based on direct learning from an already sequenced piece of DNA (e.g. of length 500,000) can be used to efficiently sequence the remaining parts of the string.

The next section describes the input domains used in our computational experiments and presents the results of the simulations. We are testing the learning algorithm LA by testing the performance of the evolved *SSP*-algorithms. The algorithms are tested on the “proper” domains for which they were designed by LA, as well as on the “wrong” domains.

2 Experiments

All inputs in our experiments are 4-symbol strings. For our testing, we used three random input domains and the domains defined by several DNA strings collected from various web-sites. The random domains include *Random*, *Weighted_Random*, and *Markov*. The strings from *Random* are generated by a procedure which selects characters of the string at random according to the uniform distribution. The *Weighted_Random* domain generates strings at random according to a weighted distribution ($A = 0.261$, $C = 0.239$, $G = 0.239$, $T = 0.261$) based on the base frequencies of Human Chromosome 22 (see <http://www.sanger.ac.uk/HGP/Chr22/>). The generator for *Markov* creates strings by a Markov process using the table computed from Human Chromosome 22. The fourth domain, termed *DNA*, uses DNA-strings as the source of the superstrings. The DNA-strings that are used for the experiments presented here are helicobacter pylori (*H. pylori*), (see <http://www.tigr.org/>) and Human Chromosome 22. The length of the former is close to 0.5 million base pairs and that of the latter is close 32×10^6 . Substrings of *H. pylori* form the domain P; the substrings of the Chromosome 22 form the domain H.

In our experiments, the substrings were selected at random with lengths that range from 200,000 to 500,000 at the interval of 50,000. Most of the experiments were performed on Ultra-SPARC 10 workstations running Solaris 8; early experiments were run using Solaris 7. For all domains, random and DNA-based, 100 randomly selected substrings were used as superstrings for learning

¹Probabilistically Approximately Correct

and different 50 strings were used as superstrings for testing experiments. In DNA sequencing, it is not always the case that the shortest superstring is chosen as the correct answer, but for the purposes of testing our databases, we look for the shortest superstring found within the search area. The fragments were obtained by randomly splitting each of the superstrings. The length of the fragments was fixed at 500; the cover ratios that we used for the results presented here were 5.0 and 6.0. 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 the superstrings, as was done 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 details of other experiments, as well as experiments with other DNA strands and test-results related to the rate of learning by LA, will be presented in the full paper. 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 number of operations and actual execution time on the the computers that were used for the experiments.

Table 1: Accuracy of Assembly, Length 500,000

	H/H	H/P	R/H	R/R
5.0	1.003 45	1.005 38	1.0191 28	1.001 48
6.0	1.002 45	1.006 36	1.0165 27	1.001 49

The accuracy of learning is measured by the **approximation ratio**, which we compute from the experiments according the following formula:

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

where N is the number of test-inputs (in our case $N = 50$); u_i is the i^{th} superstring used for testing; T_i is the superstring constructed by *Assembly* for the i^{th} test. The accuracy results are shown in Table 1 for cover ratios 5.0 and 6.0. Every column of the tables is labeled with two symbols that show the learning domain (left) and the testing domain (right). The rows correspond to the cover ratios. The two numbers of each entry in the tables 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*.

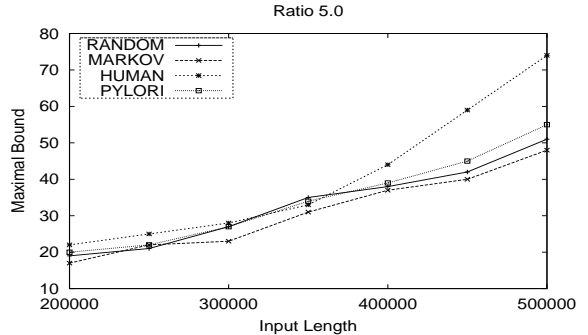


Figure 1: Maximal Non Greedy Height Bound for Ratio 5.0

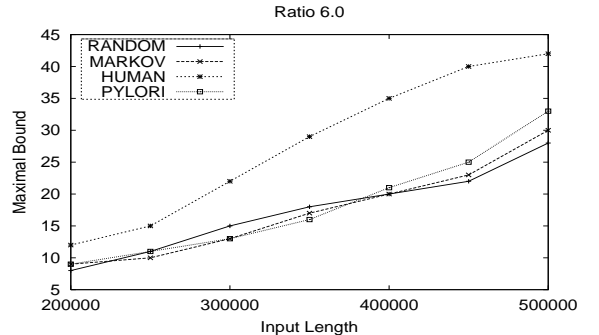


Figure 2: Maximal Non Greedy Height Bound for Ratio 6.0

The running time of the evolved algorithm *Assembly* is determined by the running times of its procedures *Greedy*, *B_Greedy*, (both polynomial) and *Restricted_backtracking* (probably exponential). Figures 1 and 2 present the growth of the parameter *Bound* of *Greedy* for four domains. Similarly, Figures 3 and 4 present the growth of the parameter *B_Bound* for *B_Greedy* for four domains. Note that even for maximal length of 500000, $\text{Bound} \leq 70$ and $\text{B_Bound} \leq 50$ (different for different domains). In all cases, both bounds are significantly smaller for cover ratio 6.0 than that for 5.0. The data also show a significant difference between the values coming from the random domains and the DNA-domains. This and other plots suggest the reason for the poor performance of the random-trained *Assembly* on the DNA-strings.

Figures 5 and 6 present the growth, for all four domains and for the two cover ratio values, of the search space size for *Restricted_Backtracking* bounded by the database

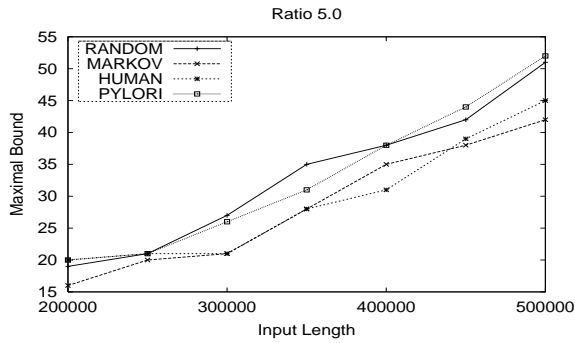


Figure 3: Boosted Greedy Bounds for Ratio 5.0

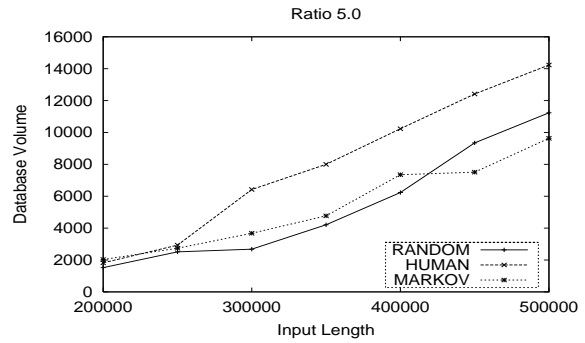


Figure 5: Database Volume for Ratio 5.0

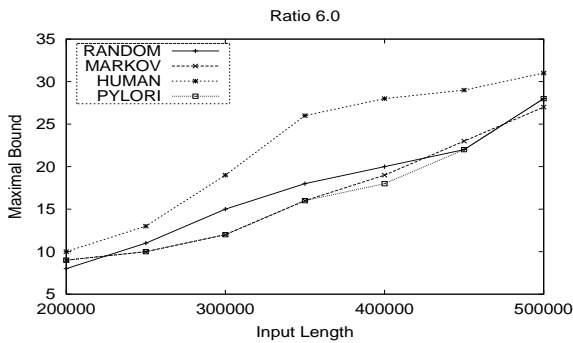


Figure 4: Boosted Greedy Bounds for Ratio 6.0

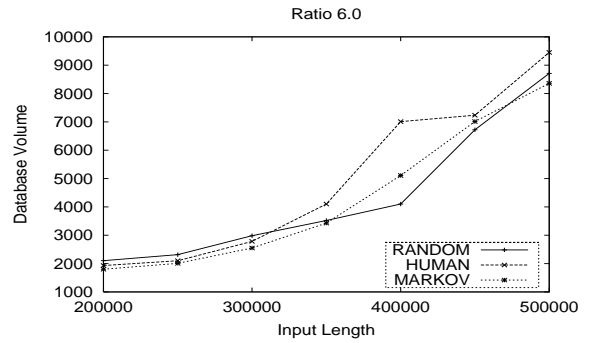


Figure 6: Database Volume for Ratio 6.0

which is accumulated during learning. The running time of the procedure is proportional to the size of the search space. Figures 7 and 8 present the growth of the size of the database (the number of lines) depending on the sizes of the strings and the cover ratios. The plots suggest a sub-exponential growth of the size of the space, but one would need more experiments with higher values of the length of the superstrings to conjecture a polynomial growth.

The timing data for *Random* using random strings is shown on Figure 9. Recent results, such as those shown on Figure 10, were obtained on Solaris 8.

References

- [1] C. Armen, C. Stein, "A 2.75 approximation algorithm for the shortest superstring problem," *DIMACS Workshop on Sequencing and Mapping*, 1994.
- [2] A. Blum, T. Jiang, M. Li, J. Tromp, M. Yannakakis, "Linear approximation of shortest superstrings," *Journal of the ACM*, 41-4, pp. 630-647, 1994.
- [3] J. Berry and M. Goldberg, "Path Optimization and Near-Greedy Analysis for Graph Partitioning: An Empirical Study," *Proceedings of Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1995.
- [4] J. Berry and M. Goldberg, "Path Optimization for Graph Partitioning Problems," *Discrete Applied Mathematics* (special issue on approximation algorithms), (90), pp. 27–50, 1999.
- [5] 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.
- [6] A. Czumaj, L. Gasieniec, M. Piotrow, W. Rytter, "Sequential and parallel approximation of shortest superstrings," 4th SWAT pp. 95-106, 1994.
- [7] M. Elloumi, "Algorithms for the Shortest Exact Common Superstring Problem," *Special Issue of the South African Computer Journal*, University of the Witwatersrand, South Africa Publish, 2000.

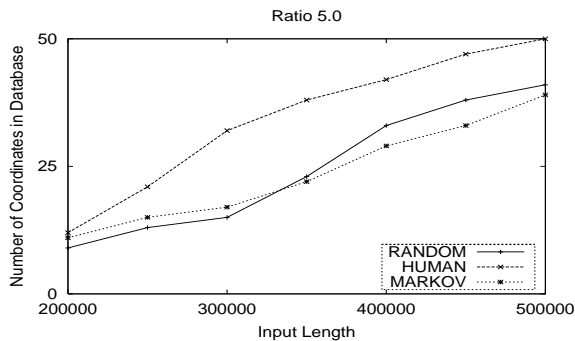


Figure 7: Database Size Ratios 5.0

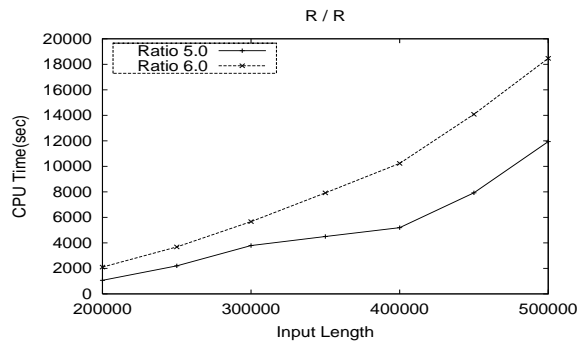


Figure 9: CPU Times for Database Testing

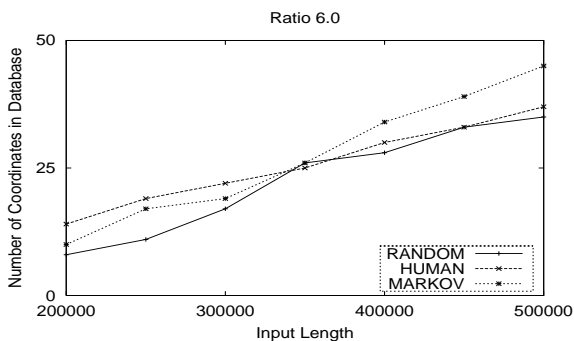


Figure 8: Database Size Ratios 6.0

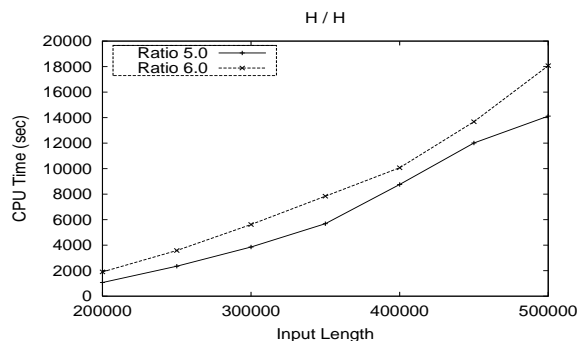


Figure 10: CPU Times for Database Testing

- [8] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
- [9] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, "Biological sequence analysis", Cambridge University Press, 1998.
- [10] M.R. Garey, and D.S. Johnson, *Computer and Intractability— A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., 1979.
- [11] 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*).
- [12] D. S. Hochbaum, (ed.) *Approximation Algorithms for NP-hard problems*. PWS Publishing Company, 1997.
- [13] R. Kosaraju, J. Park, C. Stein, "Long tours and short superstrings", *35th IEEE Symposium on Foundation of Computer Science*, 1994.
- [14] D. Maier, "The complexity of some problems on subsequences and supersequences" *J. of ACM*, 25, pp. 322-336, 1978.
- [15] T. M. Mitchell, *Machine Learning*, McGraw-Hill, 1997.
- [16] C. Papadimitriou and M. Yannakakis, "Optimization, Approximation, and Complexity Classes," *Journal of Computer and System Sciences*, Vol 43, No 3, pp. 425-440, 1991.
- [17] L. G. Valiant, "A theory of the learnable," *Communications of the ACM*, 27(11), pp. 1134-1142, 1984.
- [18] M. S. Waterman, *Introduction to Computational Biology*, Chapman and Hall, 1998.