

A Learning Algorithm for the Longest Common Subsequence Problem

Eric A. Breimer, Mark K. Goldberg and Darren T. Lim
Rensselaer Polytechnic Institute

We present an experimental study of a learning algorithm for the longest common subsequence problem, *LCS*. Given an arbitrary input domain, the algorithm *learns* an *LCS*-procedure tailored to that domain. The learning is done with the help of an *oracle*, which can be any *LCS*-algorithm. After solving a limited number of training inputs using an oracle, the learning algorithm outputs a new *LCS*-procedure.

Our experiments demonstrate that, by allowing a slight loss of optimality, learning yields a procedure which is significantly faster than the oracle. The oracle used for the experiments is the *np*-procedure by Wu *et al.*, a modification of Myers' classical *LCS*-algorithm. We show how to scale up the results of learning on small inputs to inputs of arbitrary lengths. For the domain of two random 2-symbol inputs of length n , learning yields a program with 0.999 expected accuracy, which runs in $O(n^{1.41})$ -time, in contrast with $O(n^2/\log n)$ running time of the fastest theoretical algorithm that produces optimal solutions. For the domain of random 2-symbol inputs of length 100,000, the program runs 10.5 times faster than the *np*-procedure, producing 0.999-accurate outputs. The scaled version of the evolved algorithm applied to random inputs of length 1 million runs approximately 30 times faster than the *np*-procedure while constructing 0.999-accurate solutions. We apply the evolved algorithm to DNA sequences of various lengths by training on random 4-symbol sequences of up to length 10,000. The evolved algorithm, scaled up to the lengths of up to 1.8 million, produces solutions with the 0.998-accuracy in a fraction of the time used by the *np*.

1. INTRODUCTION

This paper presents an experimental study of a learning algorithm for the longest common subsequence problem, *LCS*. Given $k \geq 2$ sequences $\{A^j = a_1^j \cdots a_{n_j}^j\}$ over a finite alphabet Σ , the problem is to find a longest sequence $Z = z_1 \cdots z_r$, which is a subsequence to each A^j ($j \in [1, k]$). *LCS* has important applications in many fields, including file comparison, molecular biology, and genetic evolution. The problem has been studied extensively (see [Bonizzoni et al. 1998; Bonizzoni and Ehrenfeuch 1996; Breimer and Goldberg 1998; Dančik and Paterson 1994;

This work was supported in part by NSF Grants #EIA-9634485 and #CDA-97-24495
Address: Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street,
Troy, NY 12180-3590; e-mail: {breime, goldberg, limd}@cs.rpi.edu

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.

Hirschberg 1975; Hirschberg 1977; Hunt and Szymanski 1977; Jiang and Li 1995; Masek and Paterson 1980; Myers 1986; Nakatsu et al. 1982; Sankoff and Kruskal 1983; Smith and Waterman 1981; Ukkonen 1985; Wagner and Fisher 1974; Wu et al. 1990]). The folk dynamic programming algorithm for *LCS* [Wagner and Fisher 1974], in the case of the inputs with two sequences of length n and m , runs in $O(nm)$ time. The question of whether subquadratic algorithms for *LCS* exist was posed in [Chvatal et al. 1972]. An $O(nm/\log n)$ -time algorithm ($n \leq m$) was developed in [Masek and Paterson 1980]. Unfortunately, even for moderately long input sequences, the speedup provided by $\log n$ is essentially eliminated by the drastic increase of the multiplicative constant. Researchers had more success with the development of algorithms that run well on special classes of inputs. An $O((r+n)\log n)$ -time algorithm, where r is the total number of ordered pairs of positions at which the two sequences match, was constructed in [Hunt and Szymanski 1977]. Ukkonen [Ukkonen 1985] and Myers [Myers 1986] developed efficient algorithms that run in $O(nd)$ -time, where d is the edit distance. Wu *et al.* [Wu et al. 1990] improved upon the running time with their $O(np)$ -time algorithm, where $p = d/2$. Although these algorithms are very fast for special cases of practical importance, they are still quadratic for sequences where $l = n - d/2$ is proportional to the input size. In particular, l is close to $0.8n$ for two binary sequences of length n generated uniformly at random [Jiang and Li 1995].

Our approach to *LCS* is to use machine learning methods. A learning algorithm for the problem of constructing a maximum clique in a graph was presented in [Goldberg and Hollinger 1998] and a learning algorithm for graph partitioning was described in [Berry and Goldberg 1995]. We adopt here the same learning model, which is essentially that of PAC learning [Valiant 1984]:

- training is done on an input domain determined by a probability distribution (or equivalently, by an input generator);
- the evolved *LCS*-procedure is tested (and expected to perform “well”) on the same domain on which it was trained;
- the performance of the evolved procedure is characterized by its performance profile, comprised of the accuracy of its output, *i.e.*, the lengths of the common subsequences, and the probability of achieving that accuracy; and
- the performance of the learning algorithm is characterized by the number of training samples needed to construct an *LCS*-procedure with a given performance profile.

When designing a learning algorithm, the main question is: “What does one *ask* the oracle so that the answers can be used for designing a new algorithm?”. Our learning algorithm uses the traces of the solutions to develop a *search area* for the future algorithm. Although the search area and the new algorithm are tailored for a specific input domain (given by the distribution of training samples), they can be applied to any input and may work well for a variety of different domains.

The next two sections describe the *learning algorithm* and the experimental data; the last section outlines some further avenues of research. Throughout the paper, for a given sequence A , $A[i]$ denotes the i th entry of A ; $A[i..j]$ denotes the subsequence of A composed of all entries from $A[i]$ till $A[j]$ ($i \leq j$); $l(A, B)$ denotes the maximal length of a common subsequence for A and B .

2. LEARNING ALGORITHM

The idea of our learning algorithm, *LA*, is to view the traces of the solutions as points in the search area. Thus, learning is the construction of the subset of the two-dimensional dynamic programming matrix¹ which comprises the entries “essential” for the given domain. It turns out that just collecting the traces and clustering the resulting set do not yield a substantial gain in efficiency. To speed up the programs, we refine the search area, a process which reduces the size of the search area while preserving the accuracy within the prescribed bound. *LA* has four stages.

Stage 1: Training inputs to *LCS* are generated.

Stage 2: The inputs are solved by an oracle-algorithm.

Stage 3: The database of solution traces is processed, yielding a *search area*.

Stage 4: A procedure for a restricted search through the search area is developed.

Explanations.

Stage 1: The training inputs can be obtained by an input generator which supplies inputs according to a fixed probability distribution. The same generator is used to test the evolved algorithm. If the domain is simply a database of inputs, then learning is done according to the standard machine learning practice ([Mitchell 1997]) by selecting one part of the database for training and the rest for testing. It should be expected that the number L of training inputs needed to learn up to a certain degree of accuracy grows together with the lengths of the input sequences. In our experiments, the number of training examples is fixed to 300.

Stage 2: Any algorithm for *LCS* can be used as an oracle. In our experiments, the oracle was an implementation of the $O(np)$ -time algorithm described in [Wu et al. 1990]. Our implementation employs the technique from ([Hirschberg 1975; Hirschberg 1977]) to create a linear memory algorithm. Given input sequences A and B of length n and m respectively, ($n \leq m$), the output of the oracle used for the database comprises a non-decreasing sequence $\{x_i\}_{i=1}^m$, called the *trace* of the solution. If $\{a_{i_l}, b_{i_l}\}$ ($l = 1, \dots, l(A, B)$) is the *LCS* constructed by the oracle, then $\forall j \in [1, m], x_j = b_{i_t}$, where t is the largest integer with $i_t \leq j$. The set of all traces is called the *solution space*.

Stage 3: The underlying empirical assumption of our strategy is the belief (supported by our experiments) that the traces of the solutions to inputs from a given domain cluster in an area significantly smaller than the whole dynamic programming $m \times n$ -matrix. Consequently, if solving the problem on inputs from the domain is confined to a search through this area, the running time is substantially reduced with an insignificant decrease in accuracy. The size and shape of the search area depend on the input domain, the degree of accuracy that we want to achieve, and the oracle’s bias. The latter is the particular way the oracle selects solutions among equally optimal ones. The goal of the processing of the “raw” database of the traces accumulated during the training stage is to extract the smallest area which would guarantee (according to the experiments) that, for an input in the domain, there

¹For the inputs with k strings, it is a k -dimensional matrix.

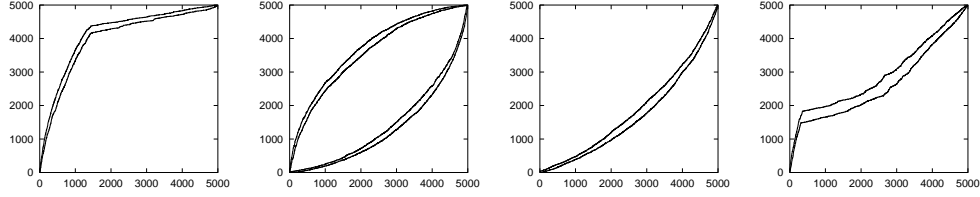


Fig. 1. Boundaries of search areas for different input domains

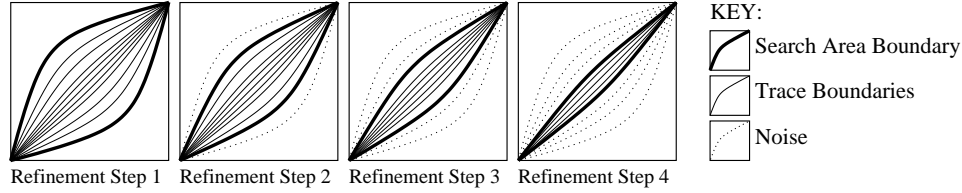


Fig. 2. Refining the search area

is a trace inside the area which yields a solution of acceptable accuracy. The construction of such a search area is done by three separate procedures: *clustering*, *refining*, and *scaling*.

Clustering. To identify clusters, we use L traces in the database to produce n sorted lists $\{list_i(\cdot)\}$, each of length L ; $list_i(\cdot)$ consists of the y -values that are the i^{th} entries of the corresponding traces. Two points, $list_i(a)$ and $list_i(a + 1)$, are included in the same cluster if

$$|list_i(a) - list_i(a + 1)| \leq tm/L,$$

where t is a tuning parameter. Executing this rule for all adjacent pairs of $\{list_i(a)\}$ ($a \in [1, L]$) may split the list into more than one cluster (Figure 1).

Refining. Each cluster is refined to reduce its size in order to extract an “essential” part of the search area. The points of the original cluster that are removed are assumed to be noise. The refinement operation cuts off the boundary points² in a sequence of steps (Figure 2). The number of refining steps is determined experimentally: after each refinement, the new algorithm with the current search area is tested; the refinement and testing are repeated until the accuracy exceeds the predetermined level.

Scaling. This operation is applied to a sequence of search areas $\{S_i\}$ obtained through learning on relatively short inputs of lengths $\{n_i\}$; ($n_i < n_{i+1}$) ($0 \leq i \leq k - 1$) in order to develop a search area S for longer inputs from the same input domain (Figure 3). Here, we consider the case of search areas described by their lower and upper bounds $\langle f_i(\cdot), g_i(\cdot) \rangle$ ($f_i(\cdot) \leq g_i(\cdot)$; $i \in [0, k - 1]$).

First, we approximate the sizes $\{|S_i|\}$ by the function $An_i^\alpha + B$, for which the parameters A , B , and α are determined using the least square method. To scale up the areas themselves, we select an integer $q \leq n_0$, and for each $i \in [0, k - 1]$, consider sets $\{f_i(x_j^i)\}$ and $\{g_i(x_j^i)\}$, where $x_j^i = \lceil j(n_i - 1)/q \rceil$ ($j = 0, \dots, q$). For

²Note that adjacent points of a cluster can have the same value.

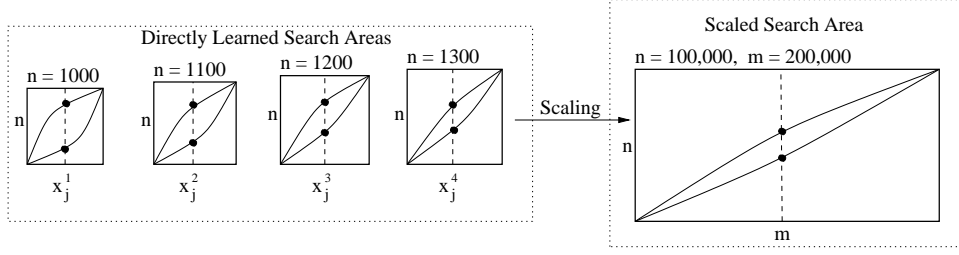


Fig. 3. Scaling search areas

each $j \in [0, q]$, the sequence $F^{(j)} = \{f_i(x_j^i)\}_{i=0}^{k-1}$ (resp. $G^{(j)} = \{g_i(x_j^i)\}_{i=0}^{k-1}$) is approximated by the function $F_*^j[n] = a[j]n - b[j]n^\gamma$ (resp. $G_*^j[n] = a[j]n + c[j]n^\gamma$). We select γ to be $\alpha - 1$ to guarantee that the size of the scaled up area would grow as n^α ; the other parameters are determined using the least square method.

Denote $f(x)$ and $g(x)$ the lower and upper bounds of the search area S on inputs (A, B) , where $|A| = n \leq |B| = m$. Then the values of these functions for $\{x_j = \lfloor j(m-1)/q \rfloor\}$ ($j \in [0, q]$) are set to be $a[j]n - b[j]n^\gamma$ and $a[j]n + c[j]n^\gamma$, respectively. The values of $f(x)$ and $g(x)$ for $x \notin \{\lfloor j(m-1)/q \rfloor\}$ are computed using linear extrapolations, that is, if $x_j < x < x_{j+1}$ for some $j \in [0, q-1]$, then we set

$$f(x) = (x - x_j) \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j}.$$

In our experiments, n_i ranges from 1000 to 10,000 and $q = n_0$.

Stage 4: The search area S developed at Stage 3 is a subset of the set of all $m \times n$ entries of the dynamic programming matrix. The new *LCS*-algorithm, called *restricted dynamic programming (rdp)*, executes the standard dynamic programming calculations restricted to S only. The volume of computation is $O(|S|)$.

Note that the description of the search area is stored separately from the main body of the program, hence switching from one domain to another does not require recompiling the program. This property allows for efficient combining learning with the execution and creates the possibility of a program which “perpetually learns” the problem.

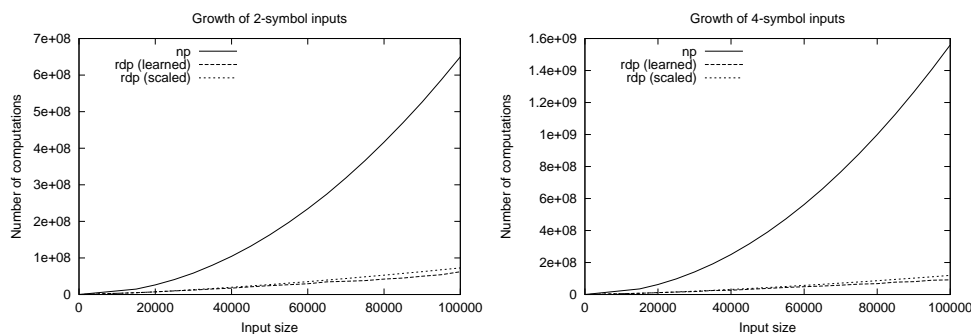
3. EXPERIMENTS

Since the learning is done on a limited number of training examples, *LA* is expected to generate a fast, but approximate procedure. To reduce experimentation to a manageable amount, we fix the target accuracy to 0.999 and evaluate the growth of the running time of the programs subject to this constraint.

The *rdp* program is trained on sequences of length up to 10,000 (either 2-symbol or 4-symbol inputs). To establish its accuracy, we compare the solution found by *rdp* with the optimal solutions found by *np* for 50 inputs from a given domain. These trials are used to measure both the gain in speed over *np* and the accuracy of the evolved program. The running times are extrapolated to project the asymptotic running time of the procedure. The areas obtained by direct learning are scaled up to construct search areas for large inputs (length in the millions). The training is done on the domain of equal length sequences generated uniformly at random.

Table 1. Asymptotic growth of the algorithms

Input Type	np (oracle)	rdp(learned)	rdp(scaled)
2-symbol	$0.065n^2$	$5.082n^{1.41}$	$5.883n^{1.42}$
4-symbol	$0.156n^2$	$6.237n^{1.43}$	$6.796n^{1.45}$

Fig. 4. Runtime growth of *rdp* and *np*

The evolved algorithm is tested on this domain, and two other input domains: (a) random inputs with non-uniform distributions described by linear functions; and (b) DNA-sequences taken from the DNA repository at TIGR³. We also test the effect of refining on the speed and the accuracy of *rdp*. Finally, we test the speed and accuracy of the scaled up *rdp*.

3.1 Runtime Experiments

The two plots in Figure 4 show the dependence of the average number of comparisons on the length of the input, respectively for 2- and 4-symbol alphabets. Notice, that for *rdp*, the number of comparisons equals the size of the search area. Our experiments, that we do not present here, show that for both programs, *rdp* and *np*, the *cpu* running time is proportional to the number of comparisons. The three curves on the plots correspond to

np program by Wu *et al.*, used as the oracle for learning and for testing the accuracy of *rdp*;

rdp(learned) program, which executes the restricted dynamic programming using the area generated by direct learning, when the accuracy is bounded from below by 0.999; the training for these experiments was done on sequences of length up to 100,000;

rdp(scaled) program, which executes the restricted dynamic programming using the area constructed by the scaler. The scaler uses search areas learned on inputs of length 1000 to 10,000 to obtain a search area for longer inputs; the accuracy of the program ≥ 0.999 .

The asymptotic growth of the running times of the algorithms was computed using the least square method and the results are presented in Table 1. To summa-

³<http://www.tigr.org/tdb/mdb/afdb/afdb.html>.

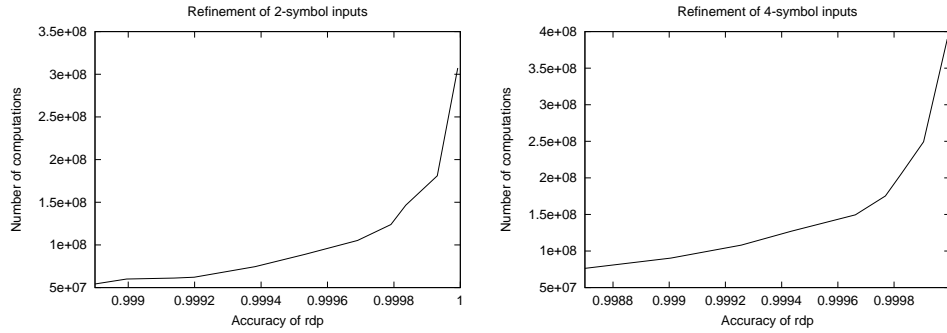


Fig. 5. Refining the search areas

size, the experiments indicate that learning and scaling yield a near-optimal LCS algorithm that is asymptotically faster than the oracle.

3.2 Refinement Experiments

The next set of experiments tests the evolution of the accuracy of the algorithm as the search area is reduced through refinement.

The two plots in Figure 5 show the dependence of the number of computations on the accuracy, respectively for 2- and 4-symbol inputs. As usual, the accuracy is defined as the ratio of the length of the common subsequence computed by *rdp* over the length the LCS. Both plots on Figure 5 present the average accuracy computed on 50 testing trials for inputs of length 100,000. The search areas were developed by collecting and clustering 300 traces. Afterwards, the search areas were reduced using the refinement process. The initial search areas yield greater than 0.9999 accuracy for both 2- and 4-symbol inputs. For 2-symbol inputs, *rdp* achieves a speedup of 2.1 over the *np*-algorithm. After refining the search area to the 0.999-accuracy, *rdp* achieves a speedup of 10.6. Similarly, for 4-symbol inputs, refinement to 0.999 accuracy increased the speedup from 4.0 to 17.3.

To summarize, the refinement significantly reduces the size of the search areas while introducing only a slight loss of optimality.

3.3 Scaling Experiments

The next set of experiments tests the accuracy of the scaling process. The scaled search areas are developed by learning on inputs of length 1000 to 10,000. Then the areas are scaled up to inputs of length 15,000 to 100,000. For all experiments, the scaled search areas achieves a slightly better than 0.999 accuracy: for 2-symbol inputs, the average accuracy is 0.9992 with a standard deviation $\sigma = 6.7 \times 10^{-5}$; and for 4-symbol inputs, the average accuracy is 0.9992 with $\sigma = 8.6 \times 10^{-5}$. In addition, we scale up to 1,000,000. Based on 20 trials, *rdp(scaled)* achieves an average accuracy of 0.9994 with $\sigma = 8.7 \times 10^{-5}$.

For 2-symbol inputs, the search areas for length 15,000 to 100,000 obtained through scaling are compared with the search areas constructed by direct learning. The *spatial* similarity of the areas is measured by the normalized intersection and difference of the corresponding areas:

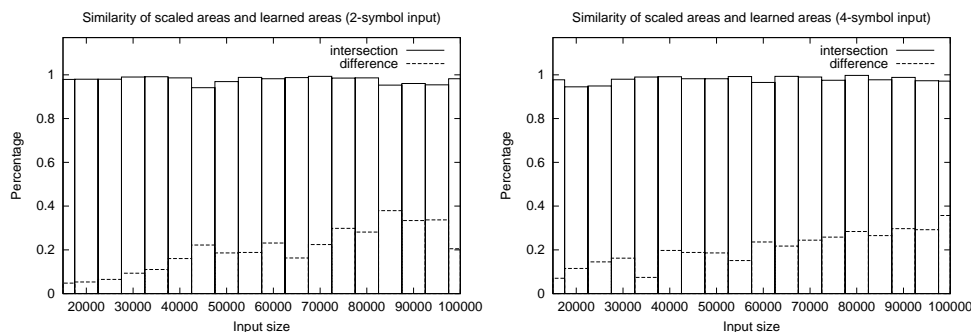


Fig. 6. Similarity of scaled and learned search areas

Table 2. The accuracy and speedup of *rdp* on DNA sequences

Input 1	Input 2	Length 1	Length 2	Accuracy	Speedup
a_fulgidus	t_maritima	2178400	1860725	0.9961	65.7
a_fulgidus	b_burgdorferi	2178400	1520758	0.9986	57.4
t_maritima	b_burgdorferi	1860725	1520758	0.9988	61.1
b_burgdorferi	t_pallidum	1520758	1138012	0.9992	59.1
h_influenzae	m_genitalium	594310	580074	0.9987	34.0
h_influenzae	h_pylori	594310	575629	0.9990	34.7
h_influenzae	m_jannaschii	594310	556034	0.9985	33.5
m_genitalium	h_pylori	580074	575629	0.9988	34.5
m_genitalium	m_jannaschii	580074	556034	0.9980	32.4
h_pylori	m_jannaschii	575629	556034	0.9981	34.1

The two bar-graphs in Figure 6 show the spatial similarity between the scaled search areas and the learned search areas for inputs of length 15,000 to 100,000.

Intersection fraction is the size of the intersection of the scaled area and learned area divided by the size of the learned area.

Difference fraction is the size of the symmetric difference of the two areas divided by the size of the learned area.

The experiments indicate that scaling provides a process for generating highly accurate search areas for large input sizes where direct learning would be infeasible.

3.4 DNA Experiments

The next set of experiments scaled search areas for large DNA sequences. The search areas were developed by scaling up the search area learned on 4-symbol random inputs of length 1000 to 10,000. Table 2 shows the results of 10 experiments. Every experiment yields an accuracy greater than 0.996. For the longest inputs, *rdp (scaled)* runs 65.7 times faster than the oracle and achieves an accuracy of 0.9961.

3.5 Non-uniform distributions

The *rdp* trained on inputs generated uniformly at random was applied to sequences generated randomly according to non-uniform distributions. Surprisingly, for all domains tested, the accuracy of *rdp* was > 0.99 . The details of these experiments

will be presented in the full paper.

4. CONCLUSION

Oracle-based learning addresses the practical need for very fast programs tailored to an input domain. Our experiments demonstrate the feasibility of this approach to the algorithm design.

The experiments raise several interesting theoretical questions. What are the probability distributions for which our learning algorithm creates “almost” optimal and fast algorithms? What are good provable estimations of the efficiency and accuracy of the evolved algorithms depending on the input domain? In particular, can it be proved that, on the inputs generated uniformly at random, the running time of the evolved algorithm is $O(n^{1.5-\epsilon})$, while the approximation ratio is $> 1 - \epsilon$ ($\epsilon > 0$)? How much does the accuracy of the default algorithm decrease when it is applied to a “wrong” input domain? In particular, can one use the learning on equal-sized inputs to produce highly accurate algorithms for inputs with substantially different lengths?

An interesting question that can be approached theoretically and/or experimentally is whether the learning algorithm can be improved by introducing a preliminary procedure which classifies the input. Given an input and a class of domains (for example, all linear distributions), how can one quickly determine the most “appropriate” domain into which the input should be placed? Finally, because of the generic nature of our learning algorithm, will it be effective for other applications of the dynamic programming paradigm?

REFERENCES

- BERRY, J. W. AND GOLDBERG, M. K. 1995. Path optimization and near-greedy analysis for graph partitioning: An empirical study. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* (1995).
- BONIZZONI, P., D’ALESSANDRO, M., VEDOVA, G. D., AND MAURI, G. 1998. Experimenting an approximation algorithm for the lcs. In *Proceedings of the Workshop on Algorithms and Experiments (ALEX 98)* (1998).
- BONIZZONI, P. AND EHRENFEUCH, A. 1996. Approximation of the shortest common super-sequence with ratio less than the alphabet size. In *Technical Report TR 149-95* (1996). Dipartimento di Scienze Della Informazione, Università Degli Studi di Milano.
- BREIMER, E. A. AND GOLDBERG, M. K. 1998. Case study of a learning algorithm for the longest common subsequence problem. In *Technical Report TR 98-3* (1998). Computer Science Department, Rensselaer Polytechnic Institute.
- CHVATAL, V., KLARNER, D. A., AND KNUTH, D. E. 1972. Selected combinatorial research problems. In *Technical Report STAN-CS-92* (1972). Computer Science Department, Stanford University.
- DANČÍK, V. AND PATERSON, M. 1994. Upper bounds for the expected length of a longest common subsequence of two binary sequences. In *Proceedings of 11th Annual Symposium on Theoretical Aspects of Computer Science* (1994), pp. 669–678.
- GOLDBERG, M. AND HOLLINGER, D. 1998. Designing algorithms by sampling. In *Proceedings of the Workshop on Algorithms and Experiments (ALEX 98)* (1998).
- HIRSCHBERG, D. S. 1975. A linear space algorithm for computing longest common subsequences. *Communications of the ACM* 18, 341–343.
- HIRSCHBERG, D. S. 1977. Algorithms for the longest common subsequence problem. *Journal of the ACM* 24, 664–675.

- HUNT, J. W. AND SZYMANSKI, T. G. 1977. A fast algorithm for computing longest common subsequences. *Communications of the ACM* 20, 350–353.
- JIANG, T. AND LI, M. 1995. On the approximation of shortest common supersequences and longest subsequences. *SIAM Journal on Computing* 24, 1122–1139.
- MASEK, W. J. AND PATERSON, M. S. 1980. A faster algorithm for computing string edit distances. *Journal of Computer and System Science* 20, 18–31.
- MITCHELL, T. M. 1997. *Machine Learning*. WCB/McGraw-Hill.
- MYERS, E. W. 1986. An $o(nd)$ difference algorithm and its variations. *Algorithmica* 1, 251–266.
- NAKATSU, N., KAMBAYASHI, Y., AND YAJIMA, S. 1982. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica* 18, 171–179.
- SANKOFF, D. AND KRUSKAL, J. B. 1983. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparisons*. Addison-Wesley, Reading, MA.
- SMITH, T. F. AND WATERMAN, M. S. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 195–197.
- UKKONEN, E. 1985. Algorithms for approximate string matching. *Information and Control* 64, 100–118.
- VALIANT, L. G. 1984. A theory of the learnable. *Communications of the ACM* 27, 1134–1142.
- WAGNER, R. A. AND FISHER, M. J. 1974. The string-to-string correction problem. *Journal of the ACM* 21, 168–173.
- WU, S., MANBER, U., MYERS, E., AND MILLER, W. 1990. An $o(np)$ sequence comparison algorithm. *Information Processing Letters* 35, 317–323.