

Discovering Optimization Algorithms Through Automated Learning

Eric Breimer, Mark Goldberg, David Hollinger, and Darren Lim

ABSTRACT. In this paper, we describe the supervised learning approach to optimization problems in the spirit of the PAC learning model. By this approach, we discover *domain-specific* algorithms by learning from an *oracle*, which is also an optimization algorithm for the problem in question. We describe examples of learning backtracking-based algorithms and algorithms that implement the dynamic programming paradigm.

1. Introduction

For many NP-hard optimization problems, heuristic algorithms perform reasonably well in practice. This suggests that linking NP-hardness and intractability is not always justified, or maybe even rarely so. A possible explanation for this phenomenon is that the input domains on which the heuristic algorithms are tested do not contain “really hard” inputs. One of the most transparent restrictions on inputs used for testing is their sizes. An algorithm designer might conclude that meeting the needs of “practical” computing can be achieved by developing libraries of procedures that are efficient on the domains of interest. This strategy is *de facto* adopted by numerous software packages, such as LEDA [38] and LINK [5]. The obvious difficulty with this approach is the multitude of optimization problems and domains of interest: there are too many of both. Typically, the design and improvement of algorithms is performed by algorithm designers, and is based on intuition about the problem, knowledge of various algorithmic techniques, and an understanding of specific input features. Designing algorithms tuned to work well on specific input domains can be tedious and time consuming, and may result in a redundant and unmanageable set of specialized algorithms. The main point of this paper is that a good way of dealing with these problems is to design learning schemes, or *learning algorithms*, that automatically learn domain-specific optimization procedures.

Automating the algorithm design process has been a focus in the machine learning and artificial intelligence research community (see [14, 31, 40, 41]). The



2000 *Mathematics Subject Classification.* **To be completed by the authors.**

Key words and phrases. **To be completed by the authors.**

The second author was partially supported by an internal RPI grant.

idea of automated algorithm development was perhaps first realized in the field of genetic programming [34]. However, since genetic programming does not focus on efficiency, so far it appears that, with some notable exceptions (see [26]), evolving programs—the main idea of genetic programming—is not yet practical. The learning approach to designing combinatorial optimization algorithms has been extensively developed in the context of reinforcement and statistical learning (see [8, 9, 10, 45, 51]). An algorithm implementing the reinforcement learning strategy keeps track of the state-space of its search and moves from state to state according to a pre-designed set of objective functions that, in effect, control the search. The objective functions are manually designed based on general and problem-specific experience.

In this paper, we describe the supervised learning approach to optimization problems in the spirit of the PAC learning model [48]. By this approach, we *learn domain-specific* algorithms by consulting an *oracle*, which is also an optimization algorithm for the problem in question. Designing optimization algorithms by means of supervised learning was initiated in [4, 24, 19] and was further applied in [11, 12, 13, 20, 21, 22, 23]. Briefly, the PAC model applied to learning combinatorial algorithms is comprised of four stages:

- (1) Algorithm training is performed on an input domain determined by a probability distribution or a suitable input generator;
- (2) The evolved algorithm is tested on the same domain on which it was trained (and is expected to perform “well”) ;
- (3) The performance of the evolved procedure is characterized by a profile, comprised of the accuracy of its output and the probability of achieving that accuracy; and
- (4) The performance of the learning process is characterized by the number of training samples needed to construct a new algorithm with a given performance profile.

For a practically feasible implementation of the supervised learning model, one needs to supply a reasonably efficient oracle-algorithm whose performance is analyzed by the learning algorithm in order to construct a more efficient target-algorithm, which we also call the evolved algorithm. For example, in order to design an algorithm for the longest common subsequence problem [13], we use the algorithm from [50] as an oracle. To design a fast heuristic for the Maximum Clique Problem [19], we use a procedure which implements a restricted backtracking algorithmic paradigm. In both cases, the usage of the oracle for learning and for testing involves time-demanding experiments. In [20, 21, 22], we employ a method of *reverse learning algorithm engineering*, for which the learning is done on inputs generated from a given set of outputs. When reversed learning algorithm engineering is possible, we completely avoid the need for an efficient oracle algorithm. Another way of avoiding the time-consuming learning stage of the design process is to do the learning on inputs of a small to moderate size, and “generalize” (scale up) the results to the inputs of a large size. This technique is successfully used in [13].

Supervised algorithm learning is essentially generic. The examples of learning presented in this paper deal with algorithmic strategies: backtracking (Section 2) and dynamic programming (Section 3). We describe the learning algorithms and



illustrate them with several developed applications. The last section of the paper outlines possible implications of learning algorithms on practical computing.

2. Learning strategy for backtracking

Backtracking is an algorithmic paradigm that can be applied to virtually any discrete optimization problem, but as is well known, it is frequently inefficient for even moderate-size inputs. Nevertheless, experiments show [4, 24] that optimal solutions can often be obtained by traversing just a small portion of the whole backtracking tree. Thus, “learning” an area containing optimal solutions, *the search area* of the domain, may lead to a backtracking-based algorithm that is efficient on the domain. Having developed a description of the appropriate search area, the new algorithm would search through those nodes of the backtracking tree that belong to the search area; the output would be the best solution found. The success rate, *i.e.*, the probability of finding an optimal solution, can be evaluated by testing, if not theoretically.

The challenge is to develop a general strategy for determining (learning), representing, and using a small search area where good solutions to inputs from a given domain can be found. In this section, we describe one such strategy which uses the language of *backtracking coordinates* for restricting the area of the search. In the following sections, we present the experimental results of two implementations of the strategy: the Maximum Independent Set problem and the Shortest Superstring problem.

2.1. Backtracking coordinates. The search performed by a backtracking algorithm can be viewed as scanning the nodes of the associated *backtracking tree*. Backtracking coordinates are strings of non-negative integers that describe the locations of the nodes in the tree, provided the order in which the algorithm visits the branches of the tree is fixed. Given two strings of non-negative integers,

$$\alpha = (a_0, \dots, a_{p-1}) \text{ and } \beta = (b_0, \dots, b_{q-1}),$$

we say that α *covers* β and write $\beta \preceq \alpha$, if appending α with $q - p$ 0's (only if $q > p$) yields

$$b_i \leq a_i, (i = 0, \dots, p - 1).$$

For a given string α , the α -box $B(\alpha)$ is defined to be the set of all strings β covered by α . Given an integer $t \geq 0$ and a string $\alpha = (a_0, \dots, a_{p-1})$, the $(t; \alpha)$ -box $B(t; \alpha)$ consists of all strings $\beta = (b_0, \dots, b_{q-1})$ covered by α and such that $\sum_{i=0}^q b_i \leq t$. When α and t are understood from the context, we will call the set a box.

Let T be a rooted directed tree and let $v \in V(T)$. The subtree $T(v)$ is defined as a subgraph of T induced on all vertices in T that can be reached from v by a directed path. A branch of v is a connected component of $T(v) - v$. A rooted directed tree with all edges directed from the root is called a *preorder tree* if, for every internal node, its children are linearly ordered. Let (v_0, v_1, \dots, v_k) be the directed path from the root $r = v_0$ to a node $v = v_k$ in a preorder tree T and let $d_i \geq 0$ be the index of v_{i+1} in the ordered set of its siblings ($i \in [0, k - 1]$). We call string $(d_0, d_1, \dots, d_{k-1})$ the T -coordinates of v in T .

Every backtracking tree is an example of a preorder tree. If E is a backtracking algorithm, T is its backtracking tree, and I is an input to the problem under consideration, then there is a node $v \in V(T)$ corresponding to a solution S of I .

The T -coordinates of v are called the *backtracking coordinates* of S with respect to E , or simply the coordinates of S when E is understood from the context. The sum of the coordinates is called *total*. Backtracking coordinates of an optimal solution indicate the deviation of an optimal solution from the greedy solution, which corresponds to the 0-string. For a string $\alpha = (a_0, \dots, a_{p-1})$, the *search area* $S(\alpha)$ covered by α is the set of all nodes of T whose backtracking coordinates (x_0, \dots, x_{s-1}) belong to the box $B(\alpha)$. The search area covered by the pair $(t; \alpha)$ is the $(t; \alpha)$ -box $B(t; \alpha)$. In general, we will consider search areas that are covered by the union of $(t; \alpha)$ -boxes. Then the scanning of such a search area involves testing membership of a node to the area. In the straightforward approach, the complexity of the testing is proportional to the size of the database containing all $(t; \alpha)$ -boxes.

2.2. Database learning. The essence of database learning is to accumulate backtracking coordinates of solutions to inputs from a given domain and then to “re-interpret” the database of the coordinates as a description of a search area for the new algorithm. In the simplest case, the backtracking coordinates of each solution is treated as a $(t; \alpha)$ -box and the union of all such $(t; \alpha)$ -boxes is used to determine the search area. The algorithm is expected to perform “well” on the inputs in the domain, but it can be applied to arbitrary inputs.

To be able to construct a database (a collection of $(t; \alpha)$ -boxes), we need an oracle—an algorithm which can find an optimal solution to a given input to the problem. In the absence of a ready-to-use oracle, a variation of backtracking can be employed (see below the explanations to Stage 3). As a rule, an oracle is not very efficient, hence, the task of learning can also be viewed as boosting the efficiency of an oracle. Note that every combinatorial optimization problem can be efficiently reduced to that with a given target value of the objective function; often, the target value is a part of the input. The algorithms designed by database learning construct, with the prescribed probability, solutions whose value of the objective function is equal to the target. The learning comprises the following five stages:

- Stage 1:** Define a backtracking tree T for the problem;
- Stage 2:** Use a given instance generator to supply inputs to oracle \mathcal{O} ;
- Stage 3:** Apply \mathcal{O} to the generated instances and store in $D.INITIAL$ the coordinates of the solutions;
- Stage 4:** Let S be the union $\cup_{\alpha} B(\alpha)$, where α ranges over the initial database $D.INITIAL$. Define the search area of the algorithm as a set SA containing S .
- Stage 5:** Set up the output algorithm to be the procedure which searches for a solution by scanning the nodes of T whose backtracking coordinates are in SA .

Thus, the learning proper is being done at Stage 4, where the accumulated database $D.INITIAL$ is re-interpreted and generalized as a search area SA described with the use of another database, termed $D.FINAL$.

2.3. Restriction trees. A more efficient representation of the union of boxes is achieved with the use of *restriction trees* defined in [19]. They enable a constant amortized time for testing the membership of a node in a subtree of a backtracking tree which is the union of boxes. The idea of the representation implemented by restriction trees is, simply, to consistently replace identical branches of boxes by using integer labels on the edges, that show the range of the corresponding

backtracking coordinates. The information about the *totals* is given by labels on the vertices. The simplest case of an *r-tree* is a restriction path, *r-path*, $P = \{(t, d_0), (t, d_1), \dots, (t, d_p)\}$, where t is the label of the vertices (identical to all), and $\{d_0, \dots, d_p\}$ are the labels of the edges. P represents the set $A(P)$ of all nodes of the backtracking tree whose coordinates belong to a $\{t; d_0, \dots, d_k\}$ -box. In general, an *r-tree* T is a rooted, directed (from the root) tree such that

- every vertex and every edge has an integer label;
- for every vertex v , a linear order is defined on the set $E(v)$; all labels on edges in $E(v)$ with a possible exception for the first one are positive.

The properties of restriction tree and algorithms for constructing restriction tree that represent the search area described by a database of solutions can be found in [19].

Creating the restriction tree can be done off-line and need only be done once for a given database. At run-time, the search algorithm keeps a pointer to a restriction tree and updates this pointer each time the search procedure moves in the search tree. The restriction relevant to the current search node is thus immediately available by looking at the restriction tree. Since the labels of the edges and nodes of the restriction tree are upper bounds for the out-degrees of the nodes of the search tree, the size of the former is usually only a fraction of that of the latter (for a complete description see [19]). However, large databases that define complex search areas may result in large restriction trees. Although the overhead associated with the restriction lookup operation is reduced to constant factor, the size of the restriction tree may grow beyond practical limits. In general, for large problems, the database required to support a high degree of accuracy contains many entries accumulated from many training samples and the size of the resulting restriction trees becomes too large to be practical.

2.4. Maximum independent set problem. The learning approach is applied in [24] and [19] to the *Maximum Independent Set* problem (linearly equivalent to the *Maximum Clique* problem). The language for describing search areas—the *backtracking coordinates*—is used in [24] to build programs that implement the standard backtracking algorithm whose search is restricted by imposing bounds on the backtracking coordinates of the tree-nodes that are to be scanned by the program. In [24], the selection of the bounds is given to the user; in [19], the notion of bounds is generalized to that of *restriction trees*, and an algorithm is described, which learns the restriction tree corresponding to a given input domain. It turns out that the language of *backtracking coordinates* is also useful for efficiently traversing the relevant part of the backtracking tree. The oracle used for our learning algorithm is a modification of the standard backtracking algorithm based on the following empirical observation: for many problems and many input domains, the individual backtracking coordinates of optimal solutions, as well as their sum, are small; in particular, the coordinates with large indices are 0's (greedy ending). By utilizing this observation, we created a procedure which is computationally acceptable as an oracle for learning.

The performance of the program implementing the evolved algorithm, called RB (for *restricted backtracking*), was tested on the domain of random graphs with edge probabilities 0.3, 0.5, and 0.7; the respective domains were denoted $\mathcal{G}(n, 0.3)$, $\mathcal{G}(n, 0.5)$, and $\mathcal{G}(n, 0.7)$. All experiments were done using a Sparc Ultra 2 Model

220, and the software is available from <http://www.cs.rpi.edu/~hollingd/rb/>. The purpose of the testing is to establish the learning rate of RB and to compare its performance with that of RS [3], which implements the reactive search strategy proposed by Battiti in [2]. Our experiments confirmed that the size of the area that with high probability contains solutions to all inputs is significantly smaller than that of the whole tree. Furthermore, the area can be “learned” using a relatively small number of examples. Our experiments also show that the rate and quality of learning are superior for denser graphs, for which the length of the solution is shorter.

Although RB’s performance is superior to that of RS on $\mathcal{G}(0.5)$ and $\mathcal{G}(0.7)$, the reverse is true for $\mathcal{G}(0.3)$. Tables 1–3 present the results of the comparison. The numbers in every block are, respectively, the average run-time in seconds, and the rate of success, *i.e.*, the proportion of the graphs for which the targets are found.

Our main objective is the improvement of RB. We identify the reason for the failure of RB to outperform RS on $\mathcal{G}(0.3)$ to be the “coarseness” of the clustering procedure for post-processing of the solutions accumulated in the training database. The search area constructed by the current clustering procedure contains too much “useless” space traversed by the evolved algorithm. Our idea is illustrated in Figure 1. The points in Figure 1(A) are collected through learning; the shaded area in Figure 1(B) is the area traversed by the standard backtracking algorithm; the search area produced by the current procedure is illustrated in Figure 1(C); and the search area that should be used by the algorithm is in Figure 1(D). Time-saving will be much higher for the k -dimensional integer inputs that arise from independent sets of the expected size k .

Note that the clustering problem we are dealing with is different from and is, in fact, easier than the classical clustering of sets in k -dimensional spaces for large k . An additional feature is that the description of the search area must be developed so that the evolved algorithm can efficiently traverse the area. The main reason this version of clustering is easier is that the effect of “non-perfect” clustering on the efficiency of the evolved algorithm is only marginal. Our current clustering algorithm is almost the simplest possible. We envision substantial improvement in this key element of the learning algorithm. More accurate clustering combined with efficient traversing can be done with the efficient use of *restriction trees* introduced in [19].

2.5. Shortest superstring problem. Database learning was applied in [20, 22] to the *Shortest Superstring Problem*, *SSP*. The instance of *SSP* is a finite collection $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ of strings over 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 [36]. A number of approximation algorithms with a fixed approximation ratio have been developed that use different variations of the greedy

TABLE 1. $q = 0.5$; Target = expected

	$n = 816$		$n = 1122$		$n = 1591$	
RS	1.21	1.0	19.9	1.0	330.5	0.8
RB	0.18	1.0	5.4	1.0	93.6	1.0

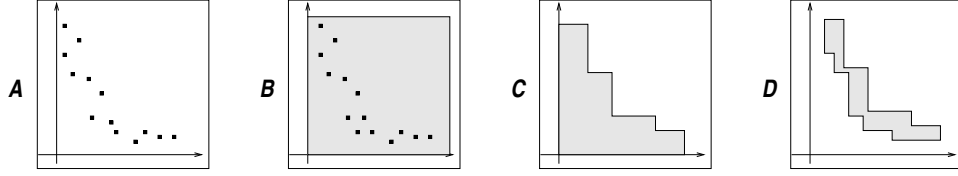


FIGURE 1. Database Clustering Illustration

strategy [1, 15, 17, 33]. *SSP* is closely related to DNA-sequencing, which aims to sequence a strand of DNA from a set of “reads” (short fragments of the future superstring) whose locations in the strand are not known [49].

Since the applications of *SSP* involve very large inputs, the only practically feasible strategy of implementing restricted backtracking appears to be to start with greedy merging, *e.g.* merging two input strings with the longest overlap. The greedy merging is applied a sufficient number of times until the number of remaining strings is sufficiently small. After that, it is replaced with a more elaborate, although more time-consuming search for the shortest superstring. For many applications, it was experimentally discovered that the need for non-greedy steps emerges only after a long sequence of greedy ones. The moment when the greedy strategy should be abandoned to achieve optimality depends on the input domain. For example, for the set of randomly generated strings the greedy strategy happen to be optimal almost till the very end; for the domain of DNA strings of the length up to 500,000, the first non-greedy merging needed for optimality was usually when close to 70 fragments left (out of an initial 5000) after the preceding merging. In our approach [20, 22], the threshold is learned by experimentation.

The important property of *SSP* is that it permits *reverse learning algorithm engineering*. Often, the set of input strings is obtained from an (unknown) superstring by a random, or quasi-random fragmentation. This is assumed to be the case for the set of reads that are obtained for the DNA-assembly problem. Thus, we can learn the “parameters” of an assembly-strategy by selecting a long string u to serve as a target-superstring, and developing, at random, a set $\{s\}$ of substrings of u , to serve as an instance of the problem. Knowing “an answer” to a given input

TABLE 2. $q = 0.5$; Target = expected - 1

	$n = 2293$		$n = 3329$		$n = 4851$	
RS	23.9	1.0	164.1	0.9	512.1	0.3
RB	1.59	1.0	13.0	1.0	171.61	1.0

TABLE 3. $q = 0.3, 0.7$

(q, n, Target)	$(0.7; 2700; 11)$		$(0.3; 1262; 26)$	
RS	378	1.0	49	1.0
RB	11	1.0	82	1.0

allows checking the correctness of the possible merges. Such a construction is easily done for the case of randomly generated strings u ; furthermore, many repositories contain already sequenced DNA-strings, that can be used for training and testing the evolved algorithms. Having created the set $\{s_i\}$ and u , one can easily determine if a greedy merging, is optimal. More generally, for a known superstring, it is computationally simple to determine backtracking coordinates of a sequence of mergings that would have yielded the superstring u from $\{s_i\}$.

This learning strategy was implemented [20] for *SSP* and in [22] for a more general variation of the string assembly problem. All inputs in our experiments are 4-symbol strings. The algorithms were tested on three domains, comprised of randomly generated strings, and several domains defined by DNA strings that were collected from various web-sites. The random domains are *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 $\{0.261, 0.239, 0.239, 0.261\}$ which is the set of frequencies of based pairs A, C, G, T in the Human Chromosome 22 [43]. The generator for domain *Markov* creates strings by a Markov process using the table computed from Human Chromosome 22. The DNA-strings that we used for the experiments are helicobacter pylori (H. pylori), (see [46]) and Human Chromosome 22. The length of the former is close to 500,000 and that of the latter is close to 32,000,000. For our learning experiments, we used randomly selected superstrings of lengths ranging from 200,000 to 500,000 at the interval of 50,000; for each value of the length and each domain, one hundred superstrings were used for learning. Each superstring was a substring of H. pylori or Human Chromosome 22; those originated from H. pylori (resp. Chromosome 22) form the domain P (resp. H). For each domains, fifty strings were used for testing. Most of the experiments were performed on ten Ultra-SPARC workstations running Solaris 8; early experiments were run using Solaris 7.

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 strings were obtained by randomly splitting each of the superstrings. The length of the strings was fixed at 500 and the cover ratios (the average coverage rate of the strings) used were 5.0 and 6.0. Every version of the evolved algorithm **Assembly** was tested on the domain that was used for its training, but also on “wrong” domains, for which it was not trained. For testing, the algorithms were given the collections of reads, 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* (R). The two main objectives of the experiments were testing the accuracy of the algorithms on the domains and measuring the running time as a function of the number of operations, as well as the actual execution time.

TABLE 4. 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:

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

where N is the number of test-inputs (in our case $N = 50$); u_i is the i^{th} superstring used for testing; \mathcal{T}_i is the superstring constructed by **Assembly** for the i^{th} test. The accuracy results are shown in Table 4 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**.

The running time of **Assembly** is determined by that of subroutines **Greedy**, **B_Greedy**, (both are provably polynomial) and **Restricted_backtracking**. The Figures 2–5 show the growth of the computation bounds versus the length of input. A detailed explanation of the bounds **Bound** and **B_Bound** can be found in [20]. Figure 2 presents the growth of the parameter **Bound** of **Greedy** for four domains. Similarly, Figure 3 presents the growth of the parameter **B_Bound** for **B_Greedy** on four domains and for cover ratios 5.0 and 6.0. 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 on the DNA-domains the version **Assembly** trained on R.

Figure 4 presents 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 which is accumulated during learning. The running time of the procedure is proportional to the size of the search space. Figure 5 presents 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 would need more experiments with longer superstrings (maybe up to several million) to conjecture a polynomial growth.

3. Oracle-based learning for dynamic programming

3.1. Introduction. Dynamic programming is a bottom-up algorithmic strategy which achieves efficiency by systematically recording solutions for small subproblems and using them to find solutions for larger subproblems. The re-computation of smaller subproblems is eliminated through the use of additional memory, which

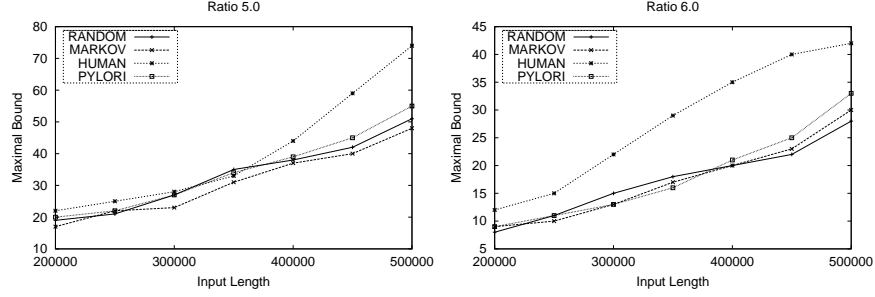


FIGURE 2. Non-Greedy Height

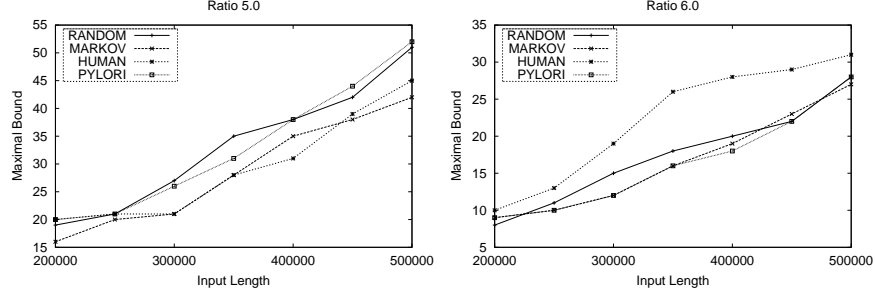


FIGURE 3. B_Bound

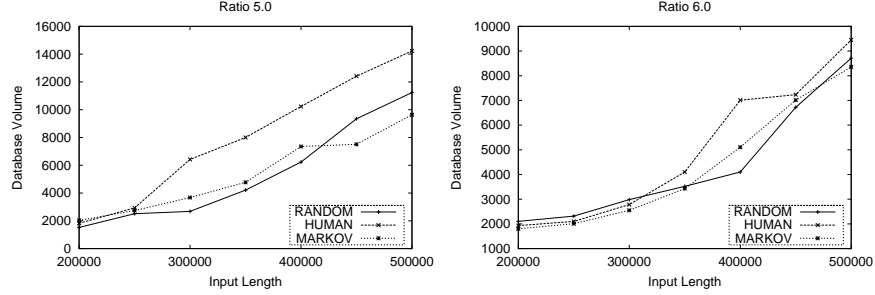


FIGURE 4. Database Volume

can be an expensive trade-off. When the set of different subproblems is reasonably small, dynamic programming is the technique of choice. Many sequence comparison problems are efficiently solvable through the use of dynamic programming; among them are files comparison [39], sequence alignment [16, 44], and the longest common subsequence problem, *LCS* [37].

While the computational benefits of dynamic programming are generally significant, for many “real-world” problems the inputs of interest are too large to be solved using this method. Whether it is possible to further reduce unnecessary or redundant computation for these types of problems is of great practical interest. Many attempts have been made to develop more efficient algorithms for *LCS*

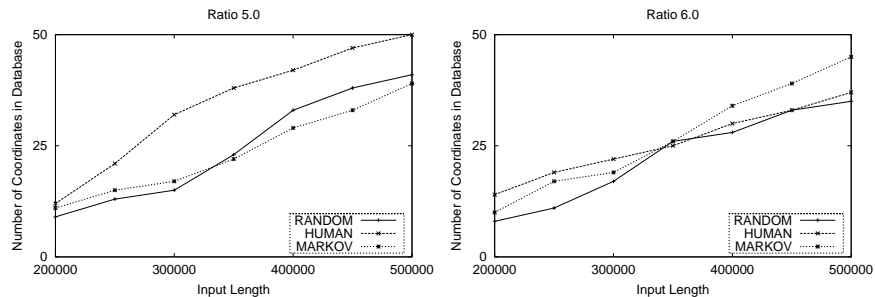


FIGURE 5. Database Size

[28, 29, 42, 47, 50]. A learning approach for designing faster *LCS*-algorithms was used in [13]. Similar to learning for backtracking, learning for dynamic programming attempts to discover the “search area” which is essential for finding solutions to instances from the input domain in question. If the solutions to inputs from a given source are “close” to each other, learning may discover a small search area which would yield a more efficient algorithm. It is important to note that as for learning for backtracking, the goal of the learning approach here is not to discover “learnable” input domains, but rather to design a framework for automatically designing an algorithm tailored for a given input domain.

An algorithm implementing the dynamic programming strategy finds a solution to a problem instance by constructing a set \mathcal{S} of solutions to all subproblems associated with the given instance. The principle of optimality asserts that if a solution to a problem is optimal, then the solution to its subproblem is also optimal. Thus, only optimal solutions must be stored. Each entry of \mathcal{S} is found as an optimal extension of the solution to a candidate subproblem. For each solution from \mathcal{S} , the algorithm would “know” the particular subproblem whose extension yields this solution. Thus, having developed a solution to a problem instance, one can backtrack through a sequence of diminishing subproblems that lead to the solution. This defines the **solution trace** related to the instance of the problem. If the set \mathcal{S} is encoded as a multi-dimensional matrix M of reals (as often the case), a solution trace is a one-dimensional sequence $\{a_{i,k_i,l_i,\dots}\}$ ($i = 1, 2, \dots$) of entries of M . The **search area** of an input domain is then the union of all solution traces¹ over all inputs from the domain. The determination of the search area (given the encoding) is the objective of learning. An **oracle algorithm** is any algorithm for the problem at hand which is utilized for collecting the solution traces for inputs from the domain of interest. The search area may depend on the oracles **bias**, that is the particular way it selects solutions among equally optimal ones.

Learning, as implemented in [13], consists of three major steps: (a) generating uniformly at random a finite number of instances from a given domain; (b) applying an oracle to each instance, in order to produce a set of solution traces; and (c) refining and scaling the search area. The algorithm which emerges from learning implements the restricted dynamic programming strategy, for which the full search is reduced to the search within the search area learned.

¹Here we neglect the fact that there can be more than one optimal solution for a given input.

The goals of the third step, refining and scaling, are (c.1) to develop a method for creating a “small” search area which would contain solution traces for “most” inputs of a given domain; and (c.2) to develop a method for scaling up the search area learned from inputs of “small” size to that for inputs of any given size. A practically acceptable relaxation of c.1 is to aim for the search area which contains solution traces of “most” of “almost” optimal solutions. The first of these tasks is close to the *rectangle learning game* described in [32]. According to our experiments with the *LCS*-problem, even reasonably simple algorithms for both tasks yield impressively accurate and fast *LCS*-algorithms.

3.2. The longest common subsequence problem. The standard quadratic dynamic programming algorithm for *LCS* constructs solutions for all subproblems defined on prefixes $\{\langle S_i, T_j \rangle\}$ of the input pair $\langle S, T \rangle$. Here, for two sequences S and T , S_i and T_j denote the prefix of S of length i and the prefix of T of length j , respectively. Let $n = |S| \geq m = |T|$. If $l(i, j)$ is the length of the shortest common subsequence of S_i and T_j , then the $n \times m$ matrix $M = [l(i, j)]$ is called the dynamic programming matrix. The algorithm computes all entries of M , in particular, $l(n, m)$, which is the real target. The first row, $\{l(i, 1)\}_{i=1}^n$, and the first column, $\{l(1, j)\}_{j=1}^m$, are computed directly. Every other entry is computed using the following reduction:

$$\begin{aligned} &\text{if } S[i] = T[j], \\ &\quad \text{then } l(i, j) = l(i-1, j-1) + 1, \\ &\text{else } l(i, j) = \max(l(i, j-1), l(i-1, j)). \end{aligned}$$

After M is computed, the solution trace and the longest common subsequence are determined by tracing back the computation using the same reduction above. The solution trace is a sequence $\{(s_i, t_i)\}$ ($i = 1, \dots, n$), where $s_1 = t_1 = 1$, $s_n = n$, $t_n = m$, and for $i > 1$, the pair (s_{i-1}, t_{i-1}) is defined by

$$\begin{aligned} &\text{if } l(s_i, t_i) = l(s_{i-1}, t_{i-1}) + 1, \\ &\quad \text{then } \{s_{i-1} = s_i - 1 \text{ and } t_{i-1} = t_i - 1\}; \\ &\text{else if } l(s_i, t_i) = l(s_i, t_i - 1), \\ &\quad \text{then } \{s_{i-1} = s_i \text{ and } t_{i-1} = t_i - 1\}; \\ &\text{else } \{s_{i-1} = s_i - 1 \text{ and } t_{i-1} = t_i\}. \end{aligned}$$

In our experiments, the oracle was an implementation of the $O(np)$ -time algorithm described in [50]. Our implementation employs the technique from [27, 28] to create a linear memory algorithm.

For each input, the oracle solves the problem and returns an optimal solution trace shown in Figure 6(a). Multiple inputs are solved to form a collection of solution traces shown in Figure 6(b). Afterwards, the collection can be refined into a dynamic programming search area shown in Figure 6(c). To identify clusters of solution traces, L traces from the database are used to produce n sorted lists $\{list_i\}$, each of length L ; $list_i$ 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 7 shows the cluster boundaries for several different randomly generated input domains.

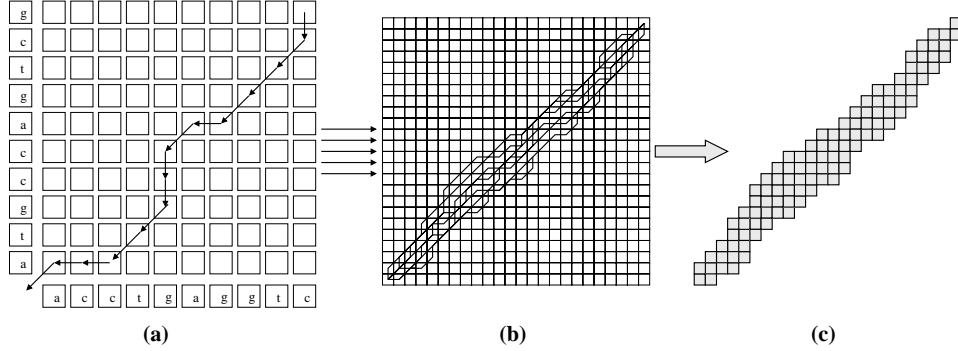


FIGURE 6. Search area formation for the *LCS* problem: (a) Solution trace, (b) Collection of solution traces, and (c) Search area. Note that the bottom left cell of matrix is defined as $l(1, 1)$.

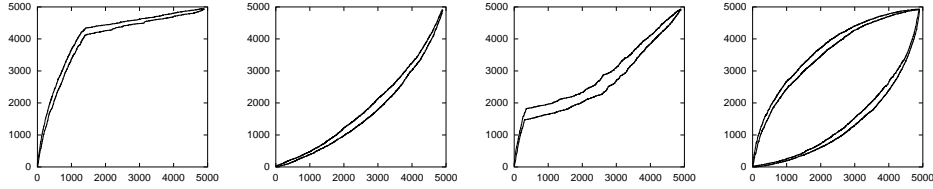


FIGURE 7. Search area boundaries for different input distributions

In our experiments, the number of training examples is fixed to 300. Each cluster is refined 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 points in a sequence of steps shown in Figure 8. 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.

A scaling 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 9). For this particular application it is advantageous to consider search areas described by their lower and upper bounds $\langle f_i, g_i \rangle$ ($f_i \leq g_i$; $i \in [0, k-1]$).

The sizes $\{|S_i|\}$ are approximated by the function $An_i^\alpha + B$, for which the parameters A , B , and α are determined using the least squares method. To scale up the areas themselves, an integer $q \leq n_0$ is selected, and for each $i \in [0, k-1]$, sets $\{f_i(x_j^i)\}$ and $\{g_i(x_j^i)\}$ are considered, where $x_j^i = \lceil j(n_i - 1)/q \rceil$ ($j = 0, \dots, q$). For 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. $G_*^j[n] = a[j]n + c[j]n^\gamma$). The value γ is selected 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 squares method.

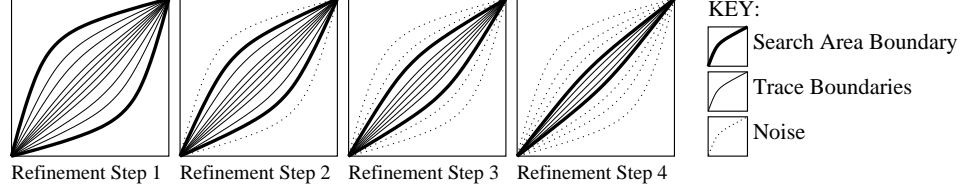


FIGURE 8. Refining the search area

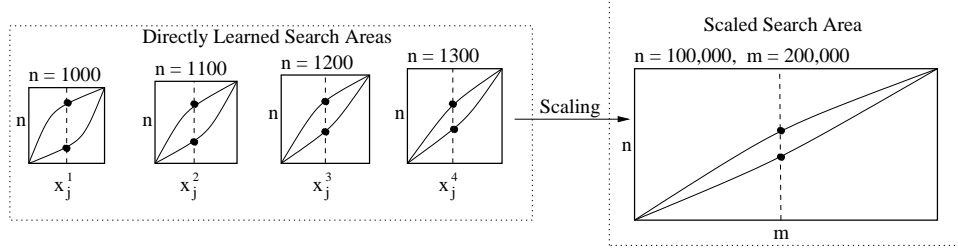


FIGURE 9. Scaling search areas

Denote $f(x)$ and $g(x)$ the lower and upper bounds of the search area S on inputs (S, T) , where $|AS| = n \leq |T| = m$. Then the values of these functions for $\{x_j = \lceil j(m-1)/q \rceil\}$ ($j \in [0, q]$) 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 $x \notin \{\lceil j(m-1)/q \rceil\}$ are computed using linear extrapolations; that is, if $x_j < x < x_{j+1}$ for some $j \in [0, q-1]$, then

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

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

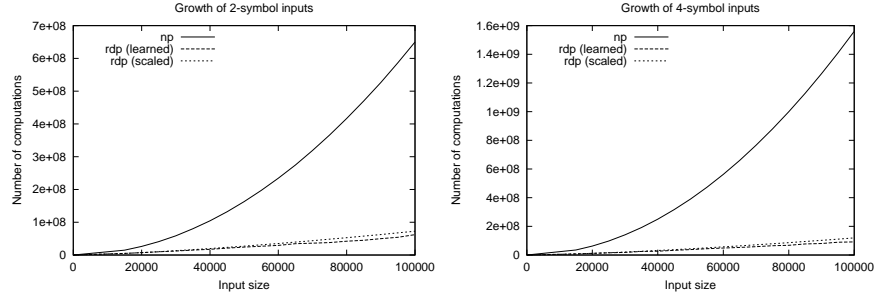
3.3. Selected experiments. To reduce experimentation to a manageable amount, we fix the target accuracy to 0.999 and evaluate the growth of the running time of the programs subject to this constraint. The *rdp*-program is trained on sequences of length up to 10,000 (either two symbol or four symbol inputs). To establish its accuracy, the solutions found by *rdp* are compared with the optimal solutions found by *np* for 50 inputs. These trials are used to measure both the speedup 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. The training is done on the domain of equal length sequences generated uniformly at random.

The two plots in Figure 10 show the relationship between the average number of comparisons and the length of the input for the following algorithms:

- *np* by Wu *et al.*: Used as the oracle for both learning and accuracy testing.
- *rdp(learned)*: Uses restricted dynamic programming on search areas that were directly learned and refined to produce 0.999 accuracy.

TABLE 5. Asymptotic growth of the algorithms

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

FIGURE 10. Runtime growth of *rdp* and *np*

- *rdp(scaled)*: Uses restricted dynamic programming on search areas constructed by scaling. Search areas learned on inputs of length 1000 to 10,000 are scaled for larger input lengths.

The measurements were done on the lengths $1000 + i \times 200$, for $i = 0, \dots, 45$. Additional experiments not present here show that for both programs, *rdp* and *np*, the cpu running time is proportional to the number of comparisons. Furthermore, the number of comparisons for *rdp* is very close to $3 \times$ the size of the search area.

The asymptotic growth of the running times of the algorithms was computed using the least squares method and the results are presented in Table 5. To summarize, the experiments indicate that learning and scaling yield a near-optimal *LCS* algorithm that is asymptotically faster than the oracle.

The two plots in Figure 11 show the effects of relaxing the accuracy constraint on the number of computations. Both plots in Figure 11 present the average accuracy computed on 50 testing trials for inputs of length 100,000. The search areas are developed by collecting and clustering 300 traces. Afterwards, each search areas is reduced using the refinement process. The two initial search areas yield greater than 0.9999 accuracy for two and four symbol inputs. For two symbol inputs, *rdp* run on the unrefined area 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 four symbol inputs, refinement to 0.999 accuracy increased the speedup from 4.0 to 17.3. Refinement significantly reduces the size of the search areas while introducing only a slight loss of optimality.

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 two symbol inputs, the average accuracy is 0.9992 with a standard deviation $\sigma = 6.7 \times 10^{-5}$; and for four symbol inputs, the average accuracy is 0.9992 with

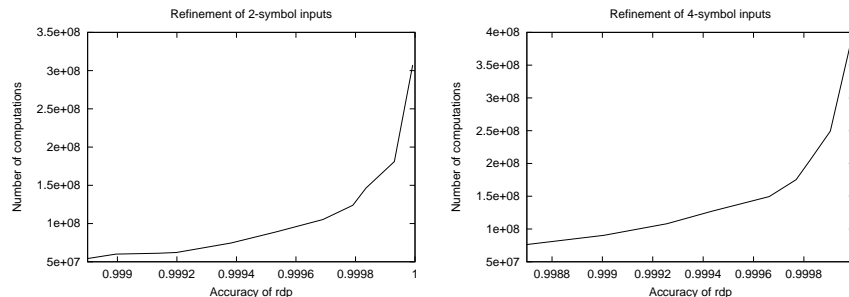


FIGURE 11. Refining the search areas

$\sigma = 8.6 \times 10^{-5}$. In an additional experiment, we further scaled the two symbol search area for inputs of length 1,000,000. Based on 20 trials, $rdp(scaled)$ achieves the average accuracy of 0.9994 with $\sigma = 8.7 \times 10^{-5}$. Scaling provides a process for generating highly accurate search areas for large input sizes where direct learning would be infeasible.

4. Conclusion

The development of the learning approach to the problem of designing optimization algorithms may have a significant impact on the model of practical computing. We envision future systems, similar to LEDA and LINK, containing learning algorithms that “perpetually” learn and store search areas for diverse domains of inputs to the corresponding problems. In addition to the collection of databases with search areas, the algorithms are supplied with classification algorithms that determine, given a specific input, the most appropriate search area or areas to be used by the programs on the input. The effectiveness of the whole system, including the learning algorithm and the classification algorithm, will in time increase when more search areas are identified and/or more efficient learning strategies are discovered.








The need for massive background and/or online experimental work necessitates the development of software support for creating, running and post-processing experiments. Such a system should be able to facilitate the monitoring of ongoing experiments and the analysis of the results. This system will use XML or a similar tool for a common data representation; automatically build experiments; and support a variety of methods for viewing graphical and textual results, as well as generating summaries.

Acknowledgements

The authors are grateful to the reviewers for useful comments.

References

- [1] C. Armen and C. Stein, A 2.75 approximation algorithm for the shortest superstring problem, *DIMACS Workshop on Sequencing and Mapping* (1994).
- [2] R. Battiti, Reactive search: toward self-tuning heuristics, in (V. J. Rayward-Smith, ed.), *Modern Heuristic Search Methods*, John Wiley and Sons, 1996, 61–83.
- [3] R. Battiti and M. Protasi, Reactive local search for the maximum clique problem, *Algorithmica* **29**(4) (2001), 610–637.

- [4] J. Berry and M. Goldberg, Path optimization and near-greedy analysis for graph partitioning: an empirical study, *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms SODA* (1995).
- [5] J. Berry, LINK: A system for experimentation with graphs and hypergraphs, *SIAM Discrete Mathematics Newsletter* **2**(7) (1997), <http://dimacs.rutgers.edu/berryj/LINK.html>.
- [6] J. Berry and M. Goldberg, Path optimization for graph partitioning problems, *Discrete Applied Mathematics* **90** (1999), 27–50.
-  [7] M. Berry and G. Linoff, *Data Mining Techniques*, John Wiley and Sons, 1997.
- [8] K. Boese, A. Kahng, and S. Muddy, A new adaptive multistart technique for combinatorial global optimization, *Operation Research Letters* **16** (1994), 101–113 .
- [9] J. Boyan and A. Moore, Learning evaluation functions to improve optimization by local search, *Journal of Machine Learning Research* **1** (2000), 77–112.
- [10] J. Boyan, W. Buntine, and A. Jagota, eds., Statistical machine learning for large-scale optimization, *Neural Computing Surveys* **3** (2000).
- [11] E. Breimer and M. Goldberg, A supervised learning approach for detecting significant local alignments, *Currents in Computational Molecular Biology, RECOMB 2002* (2002), 26–27.
- [12] E. Breimer, M. Goldberg, B. Kolstad, and M. Magdon-Ismail, On the height of a random set of points in a d-dimensional unit cube, *Journal of Experimental Mathematics* **10**(4) (2001), 583–597.
-  [13] E. Breimer, M. Goldberg, and D. Lim, A learning algorithm for the longest common subsequence problem, *Proceedings of the Second Workshop on Algorithm Engineering and Experiments, ALENEX* (2000).
- [14] J. Crutchfield and M. Mitchell, The evolution of emergent computation, *Proceedings of the National Academy of Sciences* **23** (1995), 10742–10746.
-  [15] A. Czumaj, L. Gasieniec, M. Piotrow, and W. Rytter, Sequential and parallel approximation of shortest superstrings, *4th Scandinavian Workshop on Algorithm Theory SWAT* (1994), 95–106.
- [16] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis*, Cambridge University Press, 1998.
-  [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).
- [18] M. Gelfand, A. Mironov, and P. Pevzner, Gene recognition via spliced sequence alignment, *Proc. Natl. Acad. Sci. USA* **93** (1996), 9061–9066.
-  [19] M. Goldberg and D. Hollinger, Designing algorithms by sampling, *Discrete Applied Mathematics* **110** (2001), 59–75. (Also, M. Goldberg and D. Hollinger, Designing algorithms by sampling, *Proceedings of Algorithms and Experiments*, Trento, Italy (1998).)
- [20] M. Goldberg and D. Lim, A learning algorithm for the shortest superstring problem, *Proceedings of the Atlantic Symposium on Computational Biology and Genome Information and Technology* (2001), 171–175.
- [21] M. Goldberg and D. Lim, A learning approach to shotgun sequencing, *Currents in Computational Molecular Biology, RECOMB 2002* (2002), 72–73.
- [22] M. Goldberg, D. Lim, and M. Magdon-Ismail, A learning algorithm for string assembly, *Workshop on Data Mining in Bioinformatics BIODDD, 7th International Conference on Knowledge Discovery and Data Mining, ACM SIGKDD*, 2001.
-  [23] M. Goldberg and D. Lim, Designing and Testing a New DNA Fragment Assembler VEDA-2, (to appear).
- [24] 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*, **26**, 1996, 75–88.
- [25] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
-  [6] W. E. Hart, Comparing Evolutionary Programs and Evolutionary Pattern Search Algorithms: Drug Docking Application, in (W. Banzhaf, J. Daida, A. Eiben, M. Garzon, V. Honavar, M. Jakiela, and R. Smith, eds.), *Proceedings of the Genetic and Evolutionary Computation Conference*, **1** (1999), 855–862.
- [27] D. Hirschberg, A linear space algorithm for computing longest common subsequences, *Communications of the ACM* **18** (1975), 341–343.

- [28] D. Hirschberg, Algorithms for the longest common subsequence problem, *Journal of the ACM* **24** (1977), 664–675.
- [29] J. Hunt and T. Szymanski, A fast algorithm for computing longest common subsequences, *Communications of the ACM* **20** (1977), 350–353.
- [30] A. Jain and R. Dubes, *Algorithms for Clustering Data*, Prentice Hall, 1988.
- [31] J. Jang, C. Sun, and E. Mizutani, *Neuro-fuzzy and Soft Computing*, Prentice Hall, 1997.
- [32] M. Kearns and U. Vazirani, *Computational Learning Theory*, The MIT Press, 1994.
- [33] R. Kosaraju, J. Park, and C. Stein, Long tours and short superstrings, *35th IEEE Symposium on Foundation of Computer Science* (1994).
- [34] J. Koza, Hierarchical genetic algorithms operating on populations of computer programs, *Proceedings of the 11th International Joint Conference on Artificial Intelligence* **1** (1989), 768–774.
- [35] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, The MIT Press, 1992.
- [36] D. Maier, The complexity of some problems on subsequences and supersequences, *Journal of ACM* **25** (1978), 322–336.
- [37] W. Masek and M. Paterson, A faster algorithm for computing string edit distances, *Journal of Computer and System Science* **20** (1980), 18–31.
- [38] K. Mehlhorn and S. Naher, LEDA: A library of efficient data types and algorithms, in (A. Kreczmar and G. Mirkowska, eds.), *Proceedings of the 14th International Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science* **379** (1989), 88–106, <http://www.algorithmic-solutions.com>.
- [39] W. Miller and W. Myers, A file comparison program, *Software-Practice & Experience*, **15** (1985), 1025–1040.
- [40] M. Mitchell, J. Crutchfield, and R. Das, Evolving cellular automata with genetic algorithms: a review of recent work, *Proc. of the First International Conference on Evolutionary Computation and Its Applications EVCA* (1996).
- [41] T. Mitchell, *Machine Learning*, WCB/McGraw-Hill, 1997.
- [42] E. Myers, An $O(ND)$ difference algorithm and its variations, *Algorithmica* **1** (1986), 251–266.
- [43] Sanger Institute: Human chromosome 22, <http://www.sanger.ac.uk/HGP/Chr22>.
- [44] T. Smith and M. Waterman, Identification of common molecular subsequences, *Journal of Molecular Biology* **147** (1981), 195–197.
- [45] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, MIT Press, 1998.
- [46] TIGR: The Institute for Genomic Research, <http://www.tigr.org>.
- [47] E. Ukkonen, Algorithms for approximate string matching, *Information and Control* **64** (1985), 100–118.
- [48] L. Valiant, A theory of the learnable, *Communications of the ACM* **27** (1984), 1134–1142.
- [49] J. Venter et al., The sequence of the human genome, *Science* **291** (Feb. 2001), 1304–1351.
- [50] S. Wu, U. Manber, E. Myers, and W. Miller, An $O(NP)$ sequence comparison algorithm, *Information Processing Letters* **35** (1990), 317–323.
- [51] W. Zhang, and T. Dietterich, Solving combinatorial optimization tasks by reinforcement learning: a general methodology applied to resource-constrained scheduling, (submitted), <http://www.cs.prst.edu/~tgd>.

 COMPUTER SCIENCE DEPARTMENT, RENSSELAER POLYTECHNIC INSTITUTE, 110 EIGHTH STREET, TROY, NY 12180

E-mail address: {breime, goldberg, hollingd, limd}@cs.rpi.edu