Designing algorithms by sampling

Mark K. Goldberg,[1] David L. Hollinger
goldberg@cs.rpi.edu hollingd@cs.rpi.edu

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
U.S.A.

Abstract

The paper describes a method for empirical algorithm design, called database learning, using which an algorithm is constructed based on the solutions produced by an oracle on instances from a given input domain.

We present experimental results of applying the strategy to designing heuristics for the problem of constructing a maximum independent set of a given graph. The heuristic designed by database learning is compared with the best general-purpose heuristic for the problem.

1 Introduction

Suppose we are to solve an NP-hard combinatorial optimization problem Π on inputs from a given class $\mathcal{C}$. Further suppose that an algorithm $\mathcal{O}$ (*an oracle*) is available for solving Π and there is a source of instances from $\mathcal{C}$. How can we use the information about solutions obtained by applying $\mathcal{O}$ to inputs from $\mathcal{C}$ in order to construct, after a period of learning, a robust and efficient procedure working well on $\mathcal{C}$? A more practical version of this question is how to speed up an inefficient algorithm by using it a limited number of times for learning.

Backtracking is a natural example of an oracle. It can be applied to any combinatorial optimization problem, but as is well known, for most cases, it is inefficient for even moderate-size inputs. Thus, a variation of our question is: can backtracking be significantly improved through learning? In this

[1]The work of this author was supported in part by NSF Grants #CCR-9214487 and #CDA-9634485.

paper, we describe a method for empirical algorithm design, termed database learning, which offers an answer to the questions above.

Experiments show ([3],[6]) that for some problems, given an input domain, optimal solutions for most of them are "located" in an area which is significantly smaller than the full backtracking search tree. Thus, "learning" the area containing solutions to all, or almost all inputs—*the search area* of the domain—may lead to efficient backtracking-based algorithms for the domain. Our model of learning is a variation of the PAC learning model ([10]). We assume that the problem is solvable by backtracking and the class of inputs is effectively given by a procedure which generates inputs from the class at random according to a given probability distribution. The idea of database learning is to use sampling to describe the algorithm's search area as a part of its backtracking tree. Having developed a description of the search area, the new algorithm searches only through the nodes of the backtracking tree that belong to the search area; the output is the best solution found. The success rate, *i.e.* the probability of finding an optimal solution, can be evaluated by testing.

The language used for describing search areas is that of backtracking coordinates. It was previously used in [6] 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 considered by the program. The selection of the bound is given to the user of the program. In this paper, we generalize the notion of bounds to that of restriction trees, and describe an algorithm for learning the tree corresponding to a given input domain. It turns out that the language of backtracking coordinates is also useful for efficient scanning of the relevant part of the backtracking tree.

The idea of learning by sampling with an oracle (*supervised learning*) is complementary to *self-learning.* An example of a heuristic strategy employing self-learning is the Reactive Search method proposed by Battiti in [1]. The implementation of the strategy for the maximum clique problem ([2]) is currently the most efficient[2] general-purpose heuristic for the problem. The binary code of the program, further on in the paper called `RS`, is available on the Internet (`http://rtm.science.unitn.it/~battiti/reactive.html#RLS-MAX-CLIQUE`); the user has a choice of using the program up to 10 minutes. `RS` implements the reactive search strategy, which is a variation of the randomized taby search where the restriction on the search depends on the "history" of the search; the program periodically re-starts the search from a new randomly selected point.

[2] Based on the experiments with the DIMACS benchmarks, see [2] and [7].

We tested database learning on the problem of finding a maximum independent set in a given graph (linearly equivalent to the clique problem). The class of graphs used for learning and comparison was the set of random graphs with a given edge-probability. We consider three values for edge probabilities: 0.5; 0.3; and 0.7; the respective classes are denoted $\mathcal{G}(n, 0.5)$, $\mathcal{G}(n, 0.3)$, and $\mathcal{G}(n, 0.7)$. Our experiments suggest that the size of the area containing, with high probability, solutions to all inputs is significantly smaller than that of the whole tree. Furthermore, it can be "learned" using a relatively small number of samples. The experiments also show that the rate and the quality of learning are superior for denser graphs, for which the length of the solution is shorter.

The experimental comparison of `RS` with our program, which we call `RB` (for restricted backtracking) is presented in the section "Experiments." According to our testing, `RB` clearly outperforms `RS` on graphs with edge probabilities 0.5 and higher, while `RS` is clearly more efficient for graphs with edge-probabilities 0.3 and below. We discuss a probable reason for this in the section "Conclusions."

The source code for our program is available via the web at: `http://www.cs.rpi.edu/~hollingd/rb`. A number of perl scripts are also available to automate the running of experiments using our search program.

We are grateful to anonymous referees for their comments that significantly improved the presentation of the paper.

2 Notations

In this paper, all strings are sequences of non-negative integers. Given two strings $\alpha = \{a_0, \ldots, a_{p-1}\}$ and $\beta = \{b_0, \ldots, b_{q-1}\}$, we say that α *covers* β and write $\beta \preceq \alpha$, if appending α with $q - p$ 0's (only if $q > p$) yields $b_i \leq a_i, (i = 0, \ldots, p-1)$. For a given string α, the α-box $\mathrm{B}(\alpha)$ is defined to be the set of all strings β covered by α. Given an integer $t \geq 0$ and a string $\alpha = \{a_0, \ldots, a_{p-1}\}$, the $(t; \alpha)$-box $\mathrm{B}(\alpha; t)$ consists of all strings $\beta = \{b_0, \ldots, b_{q-1}\}$ covered by α and such that $\sum_{i=0}^{q} \beta_i \leq t$. When the information about t and α is not known, or irrelevant, we will, simply, call the set a t-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 will be called a *preorder* tree if, for every internal node, its children are linearly

ordered. Let $\{v_0, v_1, \ldots, v_k\}$ 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 $(d_0, d_1, \ldots, d_{k-1})$ 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 will be 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 $\alpha = (a_0, \ldots, a_{p-1})$, the *search area* $\mathrm{S}(\alpha)$ covered by α is the set of all nodes of T whose backtracking coordinates $(x_0, \ldots, x_{s-1})$ belong to the box $\mathrm{B}(\alpha)$. The search area covered by the pair $(t; \alpha)$ is the $(t; \alpha)$-box $\mathrm{B}(t; \alpha)$.

3 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. 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, 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 the explanations to Stage 3 below.) 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. For our experiments, we implemented four methods for constructing D.FINAL. Any of these methods can be used "continuously": the learning program could be run as long as the resulting algorithms are in use.

Explanations.

→ **Stage 1:** Usually, a backtracking algorithm executes the depth-first strategy yielding the backtracking tree, which is a preorder tree. The order in which the branches of every internal node are explored can significantly change the efficiency of the algorithm. Our experiments show that even simple (polynomial) reorderings may significantly improve the running time of the resulting algorithms.

→ **Stage 2:** The main question is how many inputs are needed to "learn" a satisfactory algorithm. This is to be answered experimentally. Our experiments show that the number of required samples is surprisingly small.

→ **Stage 3:** Although an oracle is not expected to be an efficient procedure, still it must be efficient enough to process a relatively small number of inputs to be used for developing a better algorithm. The oracle used in our experiments is based on the following empirical observation: *for many problems and many domains of inputs, the* `coordinates` *and the* `total` *of optimal solutions are small; in particular, the* `coordinates` *with large indices are 0's (greedy ending).* It turns out that instead of performing the standard backtracking, it is more efficient to execute a sequence of its restricted versions, each doing the backtracking search among the nodes with `total` = 1,2, ..., until a solution with a target objective function is found. This splits the search into a number of `STEPS`, each based on larger `total` and/or `coordinates`. Yet another improvement of the oracle is to restrict the `coordinates` themselves and "grow" them at a rate slower than that of `total`. The strategy employed for our experiments used a two-line file `inc-template`, which governs the changes of the parameters in the `database`-file. The table below shows an example `inc-template`.

1	1	1	1	1	2	2	2	2	4	4	4	4	8	8	8	8	8
1	1	1	1	1	2	2	2	2	-2	-2	-2	-2	-1	-1	-1	-1	-1

Table 2

Every entry of the table indicates how often the corresponding parameter of the database is increased or decreased when the `STEP`s are executed. The leftmost 1s show that the value of `total` increases by one each time we move to the next `STEP`. All other numbers show the changes to be made in the values of the corresponding `coordinates`. The number, a, in the ith column indicates that $i-1$ coordinate d_{i-1} increases by exactly 1 every a'th `STEP`. The oracle successively applies `STEPS` to a fixed number of samples until, for a prescribed portion of them, target solutions are found. If the current `STEP` fails to find target solutions for new inputs, then second line of `inc-template` is applied; here a negative number means a decrease of the corresponding parameter, provided the parameter is[3] positive.

Note, that an oracle need not be a variation of backtracking, since for a given solution, it is computationally simple to compute its backtracking coordinates with respect to a predefined backtracking tree.
→ **Stage 4:** The search area related to a given class of inputs contains all points described by the strings from the database. Hence, we can view the learning problem as that of appropriate extrapolation of the database. Replacing every string $\alpha \in$ D.INITIAL with the box $\mathrm{B}(\alpha)$ is an extreme strategy of extrapolating the initial database. This partially reflects the specifics of the maximum independent set problem; however, we suspect that this transformation is justifiable for many other applications as well. Three methods for constructing SA from $\cup_\alpha \mathrm{B}(\alpha)$ were explored.

Method 1. SA $\equiv \cup_\alpha \mathrm{B}(\alpha)$, where α ranges over D.INITIAL. ▮

Method 2. SA $\equiv \mathrm{B}(t,\gamma)$ for t and γ defined as follows.
Let D.INITIAL $= \{\alpha_i\}$, where $\alpha_i = \{\alpha_{i,0}, \alpha_{i,1}, \ldots, \alpha_{i,s}\}$ $(i = 1, \ldots, N)$. Then

$$t \equiv \max_i \Big(\sum_{j=0}^{s} \alpha_{i,j}\Big) \quad \text{and} \quad \gamma \equiv (\gamma_0, \ldots, \gamma_s), \quad \text{where} \quad \gamma_j = \max_i(\alpha_{i,j}). \qquad ▮$$

Method 3. Let D.INITIAL be lexicographically sorted and let K be an arbitrary positive integer (K is a parameter to be chosen experimentally). Construct a set of length K sub-intervals of the list D.INITIAL that uniformly (with possible overlap) cover the list. For each interval $\mathrm{D}^{(r)}$, compute the pair (t^r, γ^r) as in Method 2, and define SA$\equiv \cup_r \mathrm{SA}(t^r, \gamma^r)$. The operation will be termed *compressing with factor K*. Note that compressing reduces the size of the database while increasing the size of the area described by the database. ▮

Obviously, Methods 1 and 2 are two extreme cases of Method 3, corresponding to $K = 1$ and $K = |\text{D.INITIAL}|$, respectively. In general, for $K = 1$, the

[3]Obviously, the idea of the file and the specific values of the entries in it reflect our empirical and very approximate "image" of the search area.

learning is slow, hence, the need for the consideration of $K > 1$. On the other hand, for $K = |\text{D.INITIAL}|$, the learning is fast, but the resulting search area can be excessively large.

NOTE: In the section describing our experiments we use the following notations to indicate which of the above methods was used: experiments that are based on method 1 as `RB(1)`, those based on method 2 as `RB(`K`)` and method 3 as `RB(MAX)`.

$\rightarrow$ **Stage 5:** Since the output algorithm is a search through a portion of the backtracking tree, any pruning of the tree that can be used for the total search can be applied in this case as well.

4 Implementation

The database developed during the learning phase is used to augment a backtracking procedure by imposing some restrictions on the search space visited during a depth-first search. Our current implementation restricts the search to only those nodes in the backtracking tree that are *covered* by at least one line in the database. Thus, the database provides an upper bound on the non-greediness of the resulting search. An alternative strategy is to search only those nodes in the search tree that are *covered* by a line in the database and *cover* a line in the database, providing both upper and lower bounds on the non-greediness of the search. We believe that including lower bounds on greediness will improve the efficiency of our algorithms for some special classes of inputs, however this strategy has not been implemented at this time.

The backtracking algorithm itself is described in [6], here we describe only the implementation issues related to the use of the database. At each node in the search tree, the backtracking algorithm must determine what actions are possible given the restrictions imposed by the database. The restrictions do not impose any limit on the depth of the search, so moving down in the search tree (this corresponds to the selection of a vertex from the graph and inclusion of that vertex in the current independent set) are not limited. However, when a backtrack occurs and the natural action of the algorithm is to move breadth-wise in the search tree, the program must check the database to determine whether this move violates the restrictions. If this breadth-wise movement would result in the generation of a node whose backtracking coordinates are not *covered* by any line in the database, the algorithm backtracks rather than to explore any more nodes at the current level in the search tree.

Determining whether a node in the search tree is *covered* by the database is

a fundamental operation in our algorithm; it is performed at each non-leaf node in the search tree and its run-time has a large impact on the total search time. We describe two implementations for this operation, one has very low memory requirements although the time for each operation is based on the size of the database, the other provides constant time, although the memory requirements are high.

4.1 Restriction lookup operation

The simplest strategy is as follows: given the backtracking coordinates of a node in the search tree, scan the entire database to see if any lines in the database *cover* the coordinates of the search node. This strategy requires only enough memory to hold the database, although the time for each operation is $O(|\mathrm{DB}|)$. We can reduce the average time for this operation by providing a mechanism for incrementally reducing the database to only the relevant lines each time a move is made in the search tree. Thus, at each node in the search tree we only need look at a subset of the entire database. Each time a breadth-wise move is made in the search tree, the current database is examined and any lines that do not cover the current position are discarded. When backtracking occurs, these lines will need to be recovered and added back to the database. In practice this strategy provides a significant improvement, although the overhead is still $O(|\mathrm{DB}|)$ for each search node.

Another strategy is to pre-compute the reductions to the database made by the previous strategy, and store the results in a suitable data structure. We implement this strategy using a data structure we call a restriction tree, *r-tree*. A restriction tree holds a symbolic representation of the search area defined by the database.

4.2 Restriction Trees

Restriction trees are used to compactly represent the union of t-boxes. They enable a constant amortized time for testing the membership of a node in this type of subtrees of a backtracking tree. The idea of the representation implemented by restriction trees is, simply, to consistently replace identical branches of t-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*, $P = \{(t, d_0), (t, d_1), \ldots, (t, d_p)\}$, where t is the label of the vertices (identical to all), and $\{d_0, \ldots, d_p\}$ are the labels of the edges. P represents the set $A(P)$ of all nodes of the backtracking tree whose coor-

dinates belong to a $\{t; d_0, \ldots, d_k\}$-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.

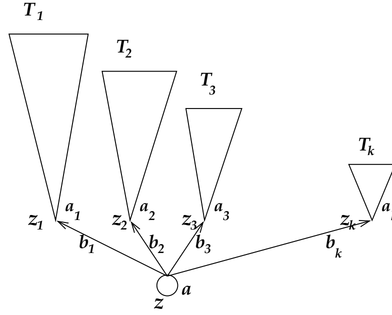

Figure 1: Restriction tree

Obviously, any branch of a node in a *r-tree* is also an *r-tree.* If $T_1, \ldots, T_k$ are the branches of the root z of T, listed according to the linear order on $E(z)$, then T can be expressed as a string

$$\langle (z, a);\ (z_1, b_1, a_1, T_1), \ldots, (z_k, b_k, a_k, T_k)\rangle.$$

Here a is the label of root z; z_i , b_i, and a_i are the root of T_i, the label of (z, z_i), and the label of z_i, respectively ($i = 1, 2, \ldots, k$). Label b_1 is non-negative; it shows the range $[0, b_1]$ of the 1st backtracking coordinate; for $i > 1$, label b_1 shows the additional range $[b_0+b_1+\ldots b_{i-1}+1, b_0+b_1+\ldots b_{i-1}+b_i]$ of the 1st backtracking coordinate. For a given *r-tree* T, $A(T)$ denotes the subtree of the backtracking tree described by T.

In this paper, we deal exclusively with restriction trees that represent the union of t-boxes. The construction of an *r-tree* is done by repeated application of a procedure `MERGE` that merges an *r-tree* T and an *r-path* P (equivalently, a t-box) into a new *r-tree* which describes $A(T) \cup A(P)$. It turns out that *r-trees* obtained by merging satisfy the "monotonicity" property:

Proposition 1. If a restriction tree $\langle (z, a);\ (b_1, T_1), (b_2, T_2), \ldots, (b_k, T_k)\rangle$ is obtained by merging t-boxes, then $\forall i = 1, \ldots, k-1$, T_i contains T_{i+1}.

The statement will become obvious after we describe the procedures `CONTAIN` (resp. `IS_CONTAINED`) for checking if an *r-tree* contains (resp. is contained

in) an *r-path* and a procedure for merging an *r-tree* with an *r-path*. Notice, that the description of procedures `CONTAIN` and `IS_CONTAINED` serves as a definition of the containment notion.

```
Procedure CONTAIN (T,P; answer)
```
/* T is an *r-tree*; P is *r-path*; `answer` is boolean */

1. `let` $T = \langle (z,a);\ (b_1, z_1, a_1, T_1), \ldots, (b_k, z_k, a_k, T_k)\rangle$;
2. `let` $P = \{(t,d_0),(t,d_1),\ldots,(t,d_p)\}$, and $P' = \{(t,d_1),\ldots,(t,d_p)\}$,
3. `if` $t > a$ then {`answer = 0`; Halt};
5. `find the smallest` l `such that` $d_0 \leq \sum_1^l b_i$
6. `if` $l > k$
7. then {`answer = 0`; Halt};
8. `else` {`answer = CONTAIN(`T_l, P'`)`; Halt};

```
Procedure IS_CONTAINED (T,P; answer)
```
/* T is an *r-tree*; P is *r-path*; `answer` is boolean */

1. `let` $T = \langle (z,a);\ (b_1, z_1, a_1, T_1), \ldots, (b_k, z_k, a_k, T_k)\rangle$;
2. `let` $P = \{(t,d_0),(t,d_1),\ldots,(t,d_p)\}$, and $P' = \{(t,d_1),\ldots,(t,d_p)\}$,
3. `if` $a \leq t$ `and` $\sum_1^k b_i \leq d_0$
4. then {`answer = IS_CONTAINED(`T_1, P'`); Halt`};
5. `else` {`answer = 0; Halt`};

The input to `MERGE` is a pair of an *r-tree* and an *r-path*. The procedure starts out by checking if one part of the input contains the other. If `answer = 1`, `MERGE` outputs the bigger part. Otherwise, the procedure tries to map the first "layer" of P onto that of T. The part which "sticks" out, either from P or from T, is appended to the part of the new tree which is obtained by merging branches of the roots of T and P.

`Procedure MERGE`$(T; P)$

1. `let` $T = \langle (z,a);\ (b_1, z_1, a_1, T_1), \ldots, (b_k, z_k, a_k, T_k)\rangle$;
2. `let` $P = \{(t,d_0),(t,d_1),\ldots,(t,d_p)\}$ and $P' = \{(t,d_1),\ldots,(t,d_p)\}$,
3. `if (CONTAIN(`T,P`)`$== 1$`) {output` T`; Halt}`;
4. `if (IS_CONTAINED(`T,P`)`$== 1$`) {output` P`; Halt}`;
5. `if` $(\sum_1^k b_j \leq d_0)$ $r = k$;
6. `else` $r = \max\{l : \sum_1^l b_j \leq d_0\}$;
7. $\Delta = d_0 - \sum_1^r b_j$; $bound = r + 1$;
8. `for` $(j = 1; j \leq r; j++)$
9. { $T_j =$ `MERGE(`T_j, P'`)`
10. `if` P' `contains` T_j`, then` $bound = \min\{bound, j\}$;
11. $a'_i = \max(t, a_i)$; }
12. `if` $(\Delta == 0)$ { $b'_r = \sum_{bound}^r b_j$;

13. $T = \langle (z, a'), (z_1, b_1, a'_1, T_1), \ldots, (z_{r-1}, b_{r-1}, a'_{r-1}, T_{r-1}),$
14. $(z_r, b'_r, t, P'), \ldots (z_k, b_k, a_k, T_k) \rangle$; }
/*the branches that are not contained by P' are merged with P'; all branches contained in P' are replaced by P' with the corresponding modification of the multiplicity of (z, z'); the remaining branches of T are appended. */
15. `else` { $T'_{r+1} =$ `MERGE`(T_{r+1}, P'); $b'' = b_{r+1} - \Delta$;
16. $T = \langle (z, a'), (z_1, b_1, a'_1, T_1), \ldots, (z_{r-1}, b_{r-1}, a'_{r-1}, T_{r-1}),$
17. $(z_r, b'_r, t, P'), (z_{r+1}, b', a_{r+1}, T_{r+1}), \ldots (z_k, b_k, a_k, T_k) \rangle$; }
/* similar to the case of $\Delta = 0$, except for the branch T_{r+1} which is split. */

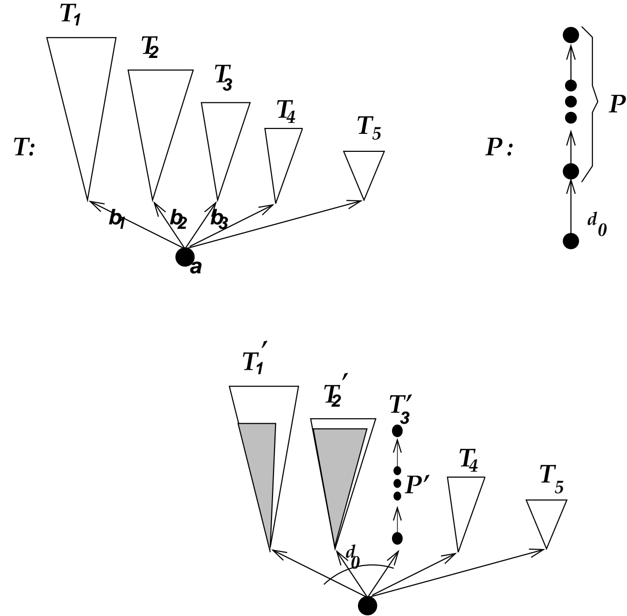

Figure 2: Illustration of merging

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 into a restriction tree and updates this pointer each time the search procedure moves in the search tree. The restriction immediately relevant to the current search node is thus available immediately by looking in the restriction tree. The size of the restriction tree is only a fraction of the size of the search tree, as each node in the restriction tree corresponds to a change in the relevant database. 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. We currently use restriction trees for all our experiments, although for some experiments we must reduce the complexity of the database (and lose some efficiency) by using the maximal extrapolation techniques described in the previous section.

5 Experiments

Database learning was experimentally tested on the maximum independent set problem ([9]); the results presented here include sets of random graphs with a given probability of an edge, $\mathcal{G}(n,p)$; we considered $p = 0.5$; $p = 0.3$ and $p = 0.7$. The program `RB` constructed by learning on these classes was compared with `RS`. The binary code for `RS` was down-loaded from the address mentioned in the introduction. The objective of the comparison was to experimentally check if and when learning on a given class of inputs yields programs that are more efficient on the class than general-purpose programs. Although we expected that `RB`, as a specially trained program, would be superior to `RS` on all classes, it turns out to be true for classes $\mathcal{G}(n, 0.5)$ and $\mathcal{G}(n, 0.7)$ only.

Additional experiments with `RB` were conducted in order to test the rate of learning of the method on class $\mathcal{G}(n, 0.5)$. The results are presented in plots.

All experiments were performed on a Sun Ultra 2, Model 2200.

5.1 Selecting the target.

It is known ([4], [5], [8]) that the expected size of the maximum independent set in $\mathcal{G}(n, 0.5)$ is $\approx 2\log_2 n$. Thus, the size of the maximal independent set increases by one when the number of vertices increases roughly by a factor of $\sqrt{2}$. For every target value t and for every q $(0 < q < 1)$, we define $n(t, q)$ to be the smallest integer such that the size of the maximum independent set in $G \in \mathcal{G}(n(t), 0.5)$ is at least t with probability $\geq q$. For our experiments, we selected random graphs with $n = n(t, q)$ vertices, where $t = 14, \ldots, 19$ and $q = 0.95$. These values, $n_t = n(t, 0.95)$, were computed with the use of

the bounds proved by Matula in [8]:

$$\sum_{j=0}^{k} \frac{\binom{n}{k}\binom{n}{k}}{\binom{n}{k}}(1-p)^{-\frac{j(j-1)}{2}} < Prob(Z_{n,1-p} > k) < \binom{n}{k}(1-p)^{\frac{k(k-1)}{2}}. \quad (1)$$

Here $Z_{n,p}$ denotes the expected size of the largest independent set in a random graph with n vertices and edge probability $p > 0$. The following table contains some of the n_k's for $p = 0.5$.

$k =$	10	11	12	13	14	15	16	17	18	19
$n_k =$	332	412	506	627	816	1122	1591	2293	3329	4851

Table 1

5.2 Comparisons with RS.

For the class $\mathcal{G}(n, 0.5)$, we considered the values of n for which the corresponding value of k from Table 1 are in the interval [14,19]. For each $k \in [14, 19]$, both RS and RB were run on the same set of ten randomly generated graphs from $\mathcal{G}(n_i, 0.5)$. Given an input, RB searched for an independent set; the corresponding input to RS is the complement of the same graph. The clique found by RS is an independent set in the original graph.

The code for RS available to us is tuned so that the program runs for at most 600 seconds. Hence, for the first set of experiments, we gave each program 10 minutes[4] and compared the actual times and the sizes of the sets found. RB was trained on 600 random graphs (the seeds used to generate training graphs and test graphs were different); the accumulated database was compressed using Method 3 with the compression factor of 10. Other experiments were conducted with the program obtained by using method 2, for which the compression factor is the number of lines in the database.

Because of the time restriction, the targets were selected so as to allow every program to finish the search within the time limit: for $n = 816,\ 1122$, and 1591, the targets are those from Table 1; for $n = 2293,\ 3329$, and 4891, the targets are one less than those theoretically expected. In Tables 2 and 3, the three numbers in every block are, respectively, the *average run-time* in seconds, the values of the *standard deviation*, and the *rate of success*, i.e., the proportion of graphs for which the targets were found.

[4]The time for generating the inputs was not counted for either program; neither was the time for training RB.

	Target = expected								
	816			1122			1591		
RS	1.21	1.25	1.0	19.9	20.9	1.0	330.5	228.6	0.8
RB	0.18	0.19	1.0	5.4	4.0	1.0	93.6	73.1	1.0

Table 2

	Target = expected - 1								
	2293			3329			4851		
RS	23.9	25.3	1.0	164.1	191.8	0.9	512.1	240.9	0.3
RB	1.59	0.80	1.0	13.0	9.58	1.0	171.61	126.10	1.0

Table 3

`RS` is a randomized procedure which re-starts the search from a new, randomly selected point, if the progress is not sufficient. Thus, re-runs[5] on the same graph may improve the success rate. Such experiments were conducted on graphs in $\mathcal{G}(2293, 0.5)$ with the target equal 17. The results were compared with `RB(10)` (the compression factor equals 10) and `RB(max)` (the compression factor equals the number of lines in the database). To achieve the success rate of 1.0, `RS` was allowed to run up to 20 times for each graph. These data, rounded off to 10 seconds, are presented in Table 4.

Number of vertices = 2293; target = 17								
RS(20)			RB(10)			RB(max)		
5930	3960	1.0	1820	1580	0.9	2550	2360	1.0

Table 4

In the case of class $\mathcal{G}(n, 0.7)$, the only value of n used for the comparison was $n = 2700$, which was selected based on the following inequalities obtained from (1).

$$0.949817 \leq Prob(Z_{2700,0.7}) \geq 11) \leq 23.804358;$$
$$0.008837 \leq Prob(Z_{2700,0.7}) \geq 12) \leq 0.009449.$$

The results of experiments with class $\mathcal{G}(n, 0.7)$ are presented in Table 5.

Number of vertices = 2700; target = 11								
RS(30)			RB(2)			RB (max)		
378	429	1.0	9	11	0.8	11	13	1.0

Table 5

[5] The condition of the re-start may not coincide with the 10-minute time-restriction, so without it, the program may perform better.

In contrast with the data above, the performance of RS on graphs in $\mathcal{G}(n, 0.3)$ is superior to that of RB. For this case, we use 999 samples for training. Using the inequalities (1), we have[6]

$$0.921056 \leq Prob(Z_{1276,0.3}) \geq 26) \leq 3682.84;$$

$$0.545188 \leq Prob(Z_{1276,0.3}) \geq 27) \leq 15.826580.$$

Number of vertices = 1262;			target = 26					
RS(10)			RB(2)			RB(max)		
49	42	1	82	56	1.0	774	648	1.0

Table 6

5.3 Rate of Learning.

For the class $\mathcal{G}(n, 0.5)$, we measured the success rate by testing the algorithms produced after learning from 50 and from 500 samples. Figures 1 (resp. 2) show the success rates measured for the algorithms resulting from learning on 50 (resp. 500 graphs); similarly, Figures 3 and 4 show success rates when the target is set to be one less than the expected maximum.

According to the experiments, the success rate for RB(max) down to RB(80) based on sampling on 500 graphs is close to 0.95. It only slightly declines with the increase of the graph size; the decline is undetected for the case of *one-less*-target. If the success rate is reduced to ≈0.90 from 0.95, the size of the area to be searched is reduced by almost four times (see that data for RB(5) in Tables 5 and 6). To achieve a given rate of success, we can either increase the number of samples, or increase the value of K for the RS(K)-family of the algorithms, or do both. It appears that more efficient algorithms result from increased sampling size. For example, if the learning is sufficiently long for RB(1) (the sampling size is sufficiently large), the resulting algorithms are more efficient and have a prescribed rate of success. A pitfall of this strategy is that the size of the database may become too large, causing one of two problems: either scanning of the search area is inefficient, or the representation of the database by a restriction tree (which makes scanning efficient) requires storage of a huge size, unavailable on small computational platforms.

[6]The accuracy of the bounds (1) apparently deteriorate with the decrease of the density; the target 26 was selected because finding an independent set of size 27 for these parameters is difficult for both programs.

# of Examples	Algorithm				
	RB(1)	RB(5)	RB(80)	RB(120)	RB(max)
5	0.04 / 0.01	0.35 / 0.33	0.35 / 0.33	0.35 / 0.33	0.36 / 0.33
50	0.27 / 0.14	0.69 / 2.25	0.83 / 16.1	0.83 / 16.1	0.83 / 16.1
100	0.33 / 0.26	0.74 / 5.29	0.93 / 31.8	0.93 / 31.8	0.93 / 31.8
200	0.34 / 0.27	0.81 / 6.60	0.94 / 38.5	0.94 / 44.7	0.93 / 44.7
500	0.55 / 0.57	0.88 / 9.89	0.95 / 38.9	0.96 / 42.4	0.95 / 45.0

Table 7; Success Rate and Database Volume (in Millions of nodes) for 1591 target=expected value

# of Examples	Algorithm				
	RB(1)	RB(5)	RB(80)	RB(120)	RB(max)
5	0.04 / 0.02	0.43 / 0.94	0.43 / 0.94	0.43 / 0.94	0.43 / 0.94
50	0.14 / 0.09	0.62 / 2.90	0.92 / 7.96	0.92 / 7.96	.92 / 7.96
100	0.21 / 0.20	0.81 / 3.78	0.94 / 12.2	0.94 / 12.2	.94 / 12.2
200	0.29 / 0.29	0.86 / 5.25	0.95 / 13.4	0.95 / 15.1	.95 / 15.1
500	0.40 / 0.51	0.88 / 7.63	0.95 / 14.0	0.95 / 14.3	.95 / 15.1

Table 8; Success Rate and Database Volume (in Millions of nodes) for 4851 target=expected value -1

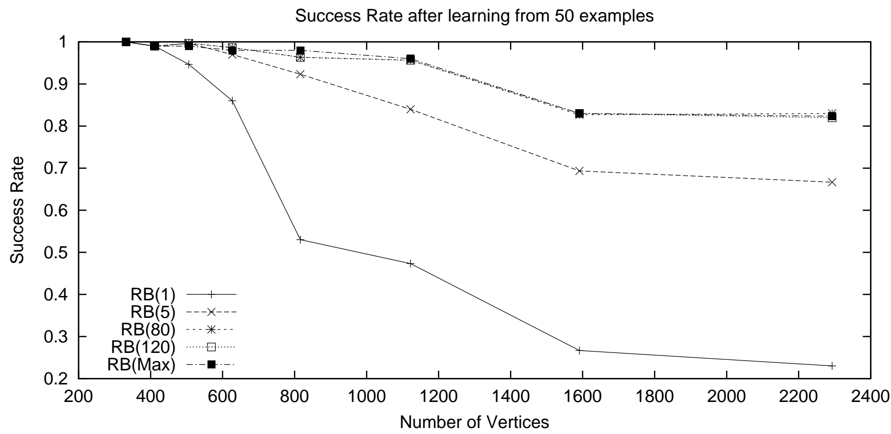

Figure 3: Success Rate for target=expected maximum, 50 examples

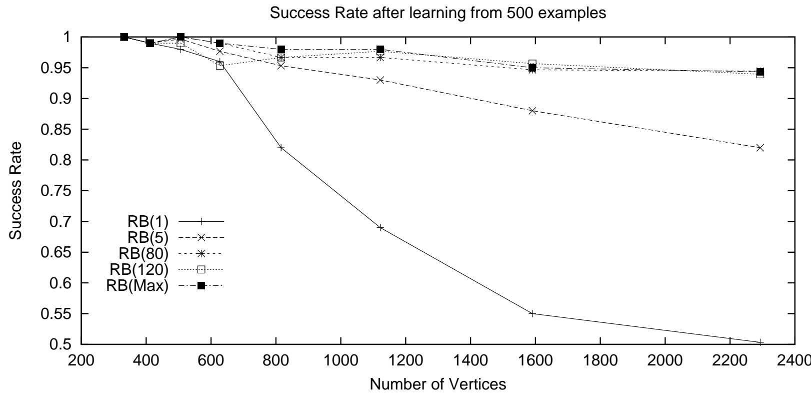

Figure 4: Success Rates for target=expected maximum, 500 examples

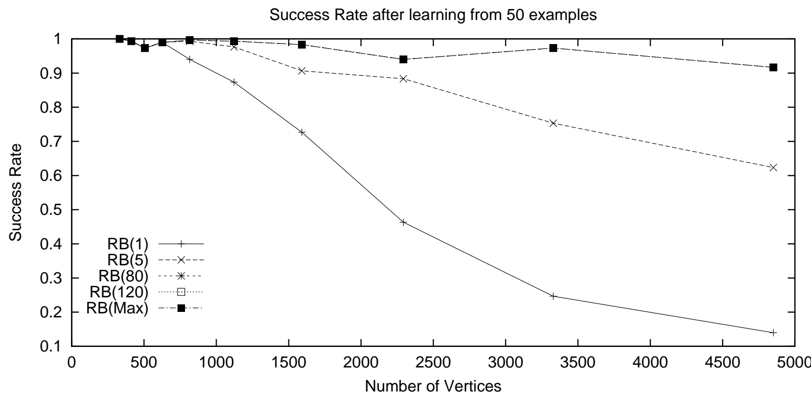

Figure 5: Success Rates for target=expected -1, 50 examples

6 Conclusions.

Our experiments show that, as a method of tailoring the algorithm's search to a class of inputs, database learning is an effective and flexible tool. The strategy is very general, the process of learning backtracking restrictions can be adapted for a variety of different[7] problems, and application of this strategy does not depend on specific classes of inputs. In general, an algorithm "trained" on one class of inputs cannot be expected to perform as well on other classes; additional training may be needed. For the database learning, the switch from class to class does not require developing a separate program or even a re-compilation of the same program: the search performed by the program is determined by the database developed during learning and stored outside of the program's code; the database is read in by the

[7] Such adaptations were developed for the vertex coloring and max-sat problems; of course, the adaptation of the scheme does not automatically create an efficient program.

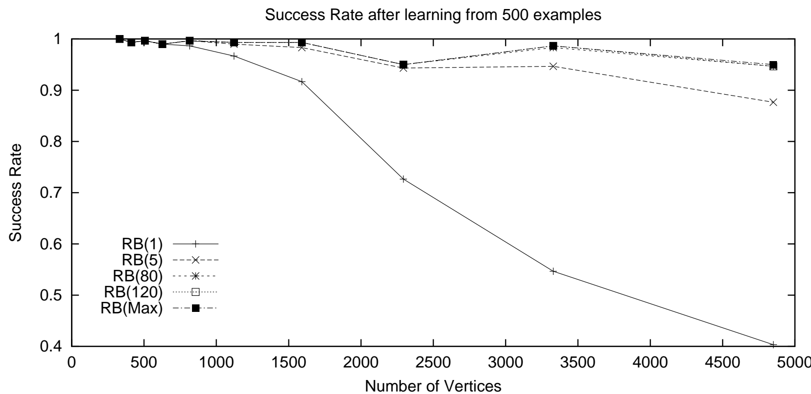

Figure 6: Success Rates for target=expected -1, 500 examples

program from a file together with the input.

The strategy itself is easily amenable. The type of extrapolation of the initial database employed for our test-problem can easily be changed by using different implementations of Stage 4. If the total scanning of the search area is prohibitively long, it can be replaced with multiple random probes of the area. This may lower the success rate, so to preserve efficiency and increase the success rate, a more sophisticated learning should be used for which the distribution of the solutions in the search area is also taken into consideration.

Because of the generic nature of database learning, it can be used as the first step in algorithm design, when no efficient theoretical algorithm is available. As a tool in experimental algorithm design, database learning is most useful when a problem must be solved repeatedly for different inputs of the same type. The initial database can be continuously updated with every new run; this process would potentially improve the accuracy and efficiency of the algorithm.

The experiments with the maximum independent set problem suggest that the method is effective in the case of the problems with a "short" objective function. The failure of `RB` to outperform `RS` on graphs of density < 0.4 points, in our view, to the deficiency of the procedure for extrapolating D.INITIAL to D.FINAL. Our interpretation of the results of the comparison is that, for the case of graphs of smaller density (< 0.4), the search area described by D.INITIAL is "porous," filled non-uniformly with solutions. The remedy to the situation may come from the following improvements of the method: (a) including in learning the information about the frequency of the distribution of the solutions in the search area, and then using this as a guide to scanning the area by the algorithm; (b) developing more sophis-

ticated reordering of the branches of the backtracking tree which hopefully "compress" the search area into a smaller one, in which the solutions are distributed evenly; and (c) developing such methods of self-learning that could be combined with database learning.

References

[1] R. Battiti, Reactive search: Toward self-tuning heuristics. V. J. Rayward-Smith, editor, *Modern Heuristic Search Methods,* chapter 4, pages 61–83. John Wiley and Sons Ltd, 1996.

[2] R. Battiti and M. Protasi, Reactive local search for the maximum clique problem, *Technical Report TR-95-052, ICSI, 1947 Center St.- Suite 600 - Berkeley, California, September 1995.*

[3] J. Berry and M. Goldberg, Path optimization and Near-Greedy Analysis for Graph Partitioning: An Empirical Study, *Proceedings of the 1995 Symposium on Discrete Algorithms.*

[4] B. Bollobás and P. Erdös, Cliques in Random Graphs, *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol 80, (1976) pp. 419–427.

[5] B. Bollobás and A. Thomason, Random Graphs of Small Order, *Annals of Discrete Mathematics* **28** (1985), pp. 47–97.

[6] M. Goldberg, R. Rivenburgh, Constructing Cliques Using Restricted Backtracking, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge,* David S. Johnson and Michael A. Trick (eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1995.

[7] David S. Johnson and Michael A. Trick (eds.) *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge,* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1995.

[8] D. Matula, The Largest Clique Size in a Random Graph, Southern Methodist University, Tech. Report, CS 7608 (April 1976).

[9] P.M. Pardalos and Jue Xue, The Maximum Clique Problem, Department of Industrial and System Engineering, University of Florida, Gainesville Fl, 1992.

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