Performance of MAX-CLIQUE Approximation Heuristics Under Description-Length Weighted Distributions

Arun K. Jagota
SUNY/Buffalo

Kenneth W. Regan[1]
SUNY/Buffalo

[1]The second author was supported in part by NSF Grant CCR-9011248.

Current address for both authors: Computer Science Department, 226 Bell Hall, UB North Campus, Buffalo, NY 14260-2000. Email: *jagota@cs.buffalo.edu, regan@cs.buffalo.edu.*

Abstract

We study the average performance of several neural-net heuristics applied to the problem of finding the size of the largest clique in an undirected graph. This function is NP-hard even to approximate within a constant factor in the worst case [ALM+92], but the heuristics we study are known to do quite well on average for instances drawn from the uniform distribution on graphs of size n. We extend a theorem of M. Li and P. Vitanyi [LV92] to show that for instances drawn from the *universal distribution* $\mathbf{m}(x)$, the average-case performance of any approximation algorithm has the same order as its worst-case performance. The universal distribution is not computable or samplable. However, we give a realistic analogue $\mathbf{q}(x)$ which lends itself to efficient empirical testing. Our results so far are: out of nine heuristics we tested, three did markedly worse under $\mathbf{q}(x)$ than under uniform distribution, but six others revealed little change.

Keywords Neural network, Hopfield network, heuristics, testing, clique problem, NP-completeness, approximation, Kolmogorov complexity, probability distributions.

1. Problem Description

The MAX-CLIQUE problem is to compute the size $\omega(G)$ of the largest clique (i.e., complete subgraph) in a given graph G, and further to find a clique of that size. It has long been known that $\omega(G)$ is NP-hard to compute exactly, so interest has centered on approximating $\omega(G)$ closely enough to suit the many applications which can be formulated in terms of MAX-CLIQUE. These include constraint-satisfaction, object-recognition, and other real-world problems. Many approximation heuristics have resulted from the study of neural networks for solving hard combinatorial optimization problems (see [BMM+89, PS89]) and are applicable to MAX-CLIQUE.

Even calculating $\omega(G)$ to within a given constant factor is NP-hard [ALM+92]. However, any heuristic A which is sensible enough always to return a maxim*al* clique comes to within a factor of 2 of optimum for "most" graphs, in the following sense: For $N \geq 2$ and $0 < p < 1$, let $\mathcal{G}_p(N)$ stand for the model of random labeled N-vertex undirected graphs in which each edge (i, j), $1 \leq 1 < j \leq N$, exists independently with probability p. B. Bollobás and P. Erdös [BE76] show that for any p and $\epsilon > 0$, the probability Prob[G has no maximal cliques of size $< (1/2)\omega(G) - \epsilon$] approaches 1 as $N \longrightarrow \infty$. The convergence is fast enough to imply $E[\omega(G)/A(G) : \mathcal{G}_p(N)] < 2 + \epsilon]$ for any sensible A and sufficiently large N. Essentially the same result was obtained by R. Karp [Kar76], who considered the "simple greedy" heuristic

$$V := \emptyset; \text{ for } i := 1 \text{ to } N, \text{ if } V \cup \{ v_i \} \text{ is a clique then } V := V \cup \{ v_i \}; \text{ return}(V),$$

which we call SD($\emptyset$). Going by sharp estimates of $\omega(G)$ for most G under $\mathcal{G}_p(N)$ (see [Mat70, BE76, Pal85]), the heuristics A we tested did much better than the factor of 2, and this is in line with other experimental work on clique and the related case of graph coloring under $\mathcal{G}_p(N)$ that we have seen [BT85, CHd87, PW89]. Despite theoretical indications [Jer90] that some of these heuristics tail off to a performance ratio of 2 for very large values of N (above 100,000), the nature of graphs drawn from $\mathcal{G}_p(N)$ still stands in marked contrast to the main result of [ALM+92], which actually gives a fixed $\epsilon > 0$ such that achieving $\omega(G)/A(G) \leq N^{\epsilon}$ for all G is NP-hard.

Several authors (see M. Li and P. Vitanyi [LV90]) have contended that the distributions $\mathcal{G}_p(N)$, or the corresponding distributions $\mathbf{unif}_n^p$ on $\{ 0, 1 \}^n$ with $\Pr(1) = p$, $n = \binom{N}{2}$, do not represent instances which tend to come up in practice. Practical instances x often arise from specifications which are much shorter than the bit-length of the data object x. A *description* of x is a program P and an argument y such that $P(y) = x$. The description is *short* if the total bit length $P(y)$ is less than $|(x)|$. The *Solomonoff-Levin distribution* $\mathbf{m}(x)$, as described in [LV90], puts more weight on inputs which have short descriptions. By contrast, $\mathbf{unif}_n^p$ favors inputs x which are incompressible; i.e., whose shortest descriptions are essentially "PRINT(x)." Li and Vitanyi concur that when the actual distribution of instances for a problem is unknown, it is most appropriate to postulate that they come from $\mathbf{m}$. In [LV92] they showed that with respect to $\mathbf{m}$, the average-case running time of any algorithm whatsoever has the same order as its worst-case running time. Our Theorem 1 is a similar result with performance ratios by approximation algorithms in place of running time. However, there is no computable, let alone efficient, way to draw samples according to the distribution $\mathbf{m}$, and we feel the proofs in [LV92] and the next section do not have practical import in themselves.

The main purpose of this work is to test whether the phenomenon suggested by Theorem 1 and the result in [LV92] can be observed under practical conditions. Namely, we ask

Question. Do algorithms perform notably worse under distributions weighted toward insatnces with short descriptions?

We also wish to ascertain which algorithms perform poorly on such distributions and which algorithms well. Section 3 describes our experimental methodology, and defines our efficient approximation $\mathbf{q}(x)$ to $\mathbf{m}(x)$. Section 4 describes the algorithms we tested, ranging from simple to more sophisticated neural-net heuristics. In Section 5 we present the data obtained thus far, on samples of fifty 100-vertex and fifty 400-vertex compressible graphs generated under a restriction of $\mathbf{q}$ to quadratically compressible graphs. We compare them with similar tests on p-random graphs with $p = 0.5$ and $p = 0.9$.

Summary of Contributions

- Theorem 1, extending Li and Vitanyi's result to approximation performance.
- A general method for testing algorithms under efficient approximations to the universal distribution. Our particular formulation uses a string algebra due to Y. Gurevich and S. Shelah [GS89]. To our knowledge, this is the first concerted effort at testing under such non-uniform distributions.
- Our experimental data (Tables I,III, and IV) show that $\mathrm{SD}(\emptyset)$, $\mathrm{SSD}(\emptyset, 1)$, and $\mathrm{SSD}(V, 1)$ fare notably worse under our restriction of $\mathbf{q}$ than under $\mathbf{unif}_n^p$. For $N = 400$ under $\mathbf{q}$ their average performance ratio over fifty graphs was not even within a factor of 2 of the best heuristic (Table IV). However, the six other heuristics show little relative change under $\mathbf{unif}_n^p$ versus under $\mathbf{q}$ (Table IV). Hence these data do not give broad support for the phenomenon raised above.
- The good performance under $\mathbf{q}$ of deterministic greedy (i.e., steepest) descent from the full vertex set V to a clique, called $\mathrm{SD}(V)$, seems surprising to us. $\mathrm{SD}(V)$ did nearly as well as mean-field annealing (MFA) [BMM+89] and a variant called *ρ-annealing* introduced in [Jag92], and better than continuous Hopfield dynamics (CHD) [Hop84]. As expected, it outperformed one run of stochastic steepest descent ($\mathrm{SSD}(V, 1)$), but it did only very slightly worse than $N := |V|$ runs of the latter ($\mathrm{SSD}(V, N)$). For details on the heuristics and their performance, see Section 5.

2. Theoretical Work

Fix a reasonable means of encoding programs P by binary strings so that any string $z \in \{0,1\}^*$ can be uniquely decoded as $z = Py$, where y is the input to P. One way is to follow the prescript in [LV90] that no program be a prefix of any other; many familiar programming languages enforce this via "END" statements. Somewhat arbitrarily we take '0' to code the one-line program "PRINT"; then all other programs begin with '1', and every string $x \in \{0,1\}^n$ has the description $0x$ of length $n + 1$. Let $\mathcal{G}_n$ stand for the collection of labeled N-vertex undirected graphs ($n = \binom{N}{2}$), which is in 1-1 correspondence with $\{0,1\}^n$.

Li and Vitanyi's work is framed in terms of probability distributions defined on all of $\{0,1\}^*$, as appropriate for inductive inference on variable-size data sets. Here we are concerned with members of $\mathcal{G}_n$, and so work in terms of *ensembles* $\mathbf{d}_n$ of probability distributions, each $\mathbf{d}_n$ defined on $\{0,1\}^n$. This involves a mathematical difference which we compensate for by giving n "for free" to programs P computing descriptions of strings x in $\{0,1\}^n$. For all n and $x \in \{0,1\}^n$ we define:

$$\begin{aligned} w(x) &:= \sum 2^{-(|P|+|y|)} \ : \ P(y,n) = x, |P| + |y| \leq n+1, \\ W(n) &:= \sum w(x) \ : \ x \in \{0,1\}^n, \\ \mathbf{n}(x) &:= w(x)/W(n). \end{aligned} \tag{1}$$

By straight counting, $W(n) \leq \sum_{P,y:|P|+|y|\leq n+1} 2^{-(|P|+|y|)} \leq n+2$. The ensemble of probability distributions $\mathbf{n}(x)$ is similar to $\mathbf{m}(x)$, and is likewise not computable as a function of x. For any algorithm A which computes a total function from graphs to natural numbers, define $wc(A,n)$ to be the maximum of $\omega(G)/A(G)$ over $G \in \mathcal{G}_n$. Adapting the proof of the result for running time in [LV92], we show:

Theorem 1. *For any algorithm A there is a constant $c \geq 1$ such that for all n,*

$$E[\omega(G)/A(G) \ : \ \mathbf{n}] \geq \frac{1}{c} wc(A,n). \tag{2}$$

Proof. Write a fixed program P which on any input (y,n) ignores y and finds by exhaustive search the lexicographically first graph $G \in \mathcal{G}_n$ for which $\omega(G)/A(G) = wc(A,n)$. Call this graph H_n, and let $p := |P|$. Since P ignores y,

$$w(H_n) \geq \sum_{y \ : \ |y| \leq n+1-p} 2^{-(p+|y|)} = \frac{n+1-p}{2^p}.$$

Hence

$$E[\omega(G)/A(G) \ : \ \mathbf{n}] \geq wc(A,n) \cdot \mathbf{n}(H_n) =$$

$$= wc(A,n) \frac{w(H_n)}{W(n)} \geq \frac{wc(A,n)(n+1-p)}{2^p n} = \Theta[wc(A,n)]. \quad \square$$

Remark. The time taken by P to compute H_n is exponential in n. P. Miltersen [Mil91] gives evidence that the property "average case = Θ[worst case]" does not hold for any polynomial-time computable distribution. It is an interesting theoretical question to give conditions under which for polynomial-time computable $\mathbf{d}$, $E[\omega(G)/A(G) \ : \ \mathbf{d}]$ is worse than $E[\omega(G)/A(G) \ : \ \mathbf{unif}]$.

We interpret Theorem 1 as saying that the hardest instances have short descriptions. We are chiefly interested in the converse:

Are instances with short descriptions *hard?*

Insofar as Theorem 1 hinges on just one graph H_n under a noncomputable distribution, its 'yes' answer may seem irrelevant, but we suspect it to be a harbinger that this is indeed so under realistic conditions. The rest of this paper presents a means of testing the question, a description of the algorithms we tested, and some data which give mixed answers.

3. Methodology

We use a functional programming notation $\mathcal{L}$ due to Y. Gurevich and S. Shelah [GS89] which captures all programs which run in *nearly linear time*, viz. time bounded by $c \cdot n(\log n)^k$ for some fixed $c, k > 0$. We wrote in the C programming language a program D which decodes any binary string z into an $\mathcal{L}$-program P and an argument y. Details of $\mathcal{L}$ and D are given below. We also wrote in C a program U which takes P, y, and the target graph size n as arguments, and simulates P on input y. If $x := P(y)$ does not have length n or greater, the output is discarded. Else we define $U(z, n) = U(P, y, n)$ to be the graph G_x constructed by taking the first n bits of x.

With reference to our specific decoder D and simulator U, we define, for all n and $x \in \{0,1\}^n$:

$$\begin{aligned} w_{\mathbf{q}}(x) &:= \sum 2^{-|z|} \ : \ |z| \leq |x| + 1, U(z, |x|) = x, \\ W_{\mathbf{q}}(n) &:= \sum w_{\mathbf{q}}(x) \ : \ x \in \{0,1\}^n, \\ \mathbf{q}(x) &:= w_{\mathbf{q}}(x)/W_{\mathbf{q}}(n). \end{aligned} \tag{3}$$

The distribution $\mathbf{q}$ is computable and samplable by generating strings z of length $\leq n + 1$ uniformly at random. Except in relatively rare cases, the sampling and decoding of each z takes nearly linear time as a function of n.

In the series of tests reported in this paper we did not do this, but rather restricted our sampling to strings z of length approximately $\sqrt{n}$. Limitations of time and hardware led us to avoid working with seed strings of length close to n, as sampling according to $\mathbf{q}$ would require. We felt that if the phenomenon raised above were true, we could detect it more readily by limiting the sample to graphs known to have relatively high weight under $\mathbf{q}$, and comparing that to samples drawn from uniform distribution. Hence we worked with this restriction of $\mathbf{q}$. Our choice of quadratic compression was partly motivated by the hard-to-approximate graphs in [FGL+91, ALM+92]. These graphs are described by a fixed oracle Turing machine M which represents a successful interactive proof system for an NP-complete language, and an instance x of some length m. The graph $G_{M,x}$ has $m^{O(1)}$ vertices which represent accepting *transcripts* of the protocol (see [FGL+91]), two of which are joined by an edge iff the oracle answers in the respective transcripts do not contradict each other. The description of $G_{M,x}$ has length $m + O(1)$, while the bit-length of $G_{M,x}$ is at least m^2.

The decoding strategy we used was to regard z as $l \cdot y \cdot P'$, where l is a self-delimiting description of the length of the argument y and P' includes the bits for P as well its self-delimiting description (number of functions in P; number of bits for each of their parameters). The advantage of this decoding strategy is that y and P scale well with length of z.See Appendix I for implementation details.

The following description of the eight basic string functions from [GS89] assumes that the shown occurrences of substrings meeting the 'if' conditions are leftmost in x, and for $R2$, $R3$, and E, that the "parameter strings" $u, v, \ldots$ all have the same length.

Basic Functions:

$R0_{u,y}(x)$: If $x = ur$ then yr, else x.
$R1_{u,y}(x)$: If $x = tur$ then tyr, else x.
$R2_{u,v,y,z}(x)$: If $x = sutvr$ then $sytzr$, else x.
$R3_{u,v,w,y,z}(x)$: If $x = sut_1vt_2wr$ with $|t_1| = |t_2|$ then sut_1yt_2zr, else x.
$E_{u,v}(x)$: Simultaneously replace every 0 in x with u and every 1 with v.
$C_{u,v}(x)$: If $x = E_{u,v}(y)$ for some y then y, else x.
$A_u(x)$: Add a tail of $|x| \log |x|$-many copies of u to x.
$D_u(x)$: Delete the maximal tail of u's in x.

Constructors:

1. Composition of functions.
2. *Iterated replacements* of the form $(\vec{R})^*(x)$, where $\vec{R}$ is a composition of any number of $R0 \ldots R3$ functions, and $\vec{R}$ is applied $|x|$-many times.

The main theorem of [GS89] states that every function computed by a random access machine in nearly linear time (NLT) is computed by some program in $\mathcal{L}$, and vice-versa. Thus $\mathcal{L}$ is universal for NLT computation. This justifies regarding $\mathbf{q}$ as an efficiently computable analog of $\mathbf{m}$. It does not follow that our decoder runs in nearly-linear time, since the definition of $\mathbf{q}$ quantifies over $\mathcal{L}$-programs, and indeed we obtain quadratic and greater expansions in the length of the output x compared to that of the seed z. However, except for cases where an occurrence of $C_{u,v}(\cdot)$ or $D_u(\cdot)$ causes a large contraction of an expanded string, the time remains nearly linear in the length of x.

The main practical reason for using $\mathcal{L}$ is its simplicity and ease of implementing. Also, while the expansion operations E and A always apply, the contraction operations C and D most often have no effect. Hence $\mathcal{L}$ has a bias toward expansion which is not unnatural, and which reduces the sampling time. Indeed, we were surprised to find that no fewer than one out of every six randomly chosen seeds expanded out to a large enough graph.

4. Experiments and Analysis

The experiments were performed on a SUN SparcStation I. Two sizes of graphs were considered: 100 vertices and 400 vertices. The bitstrings of the 100-vertex graphs had length 4950, and those of the 400-vertex graphs, 79,800. For each size, three sets of fifty graphs each were generated. One set was drawn from the uniform distribution with $p = 0.5$, and one with $p = 0.9$. All seed strings were generated using the standard UNIX pseudorandom number generator, and were recorded so that the experiments could be repeated.

The third set was generated using seed strings of lengths $65 \ldots 85$ for the fifty 100-vertex graphs and eleven of the 400-vertex graphs, and $270 \ldots 285$ for thirty-nine of the 400-vertex graphs. When we compiled the 100-vertex set we found that eleven seeds expanded to strings of length $> 79,800$. Rather than truncate them to length 4950, we decided to discard them from the 100-vertex sample but include them into the 400-vertex sample. For hardware reasons we also set a limit of 700,000 on the number of bits produced at any stage of the decoding,

and discarded those seeds which broke it from the 400-vertex sample. We believe that these practical decisions did not bias our results. It took a morning and afternoon of computing time to assemble this set.

The long strings of 0s and 1s were truncated to length 4950 or 79,800 and formed into an adjacency matrix for the graph in the order (1,2), (1,3), (2,3), (1,4),... of edges. Over three-fourths of these strings were generated by $\mathcal{L}$-programs whose final instruction was "Add a tail of $|x|$-many copies of u to x," where u was fairly long, so that most of the final bit-string comprised repetitions of u. We do not know what systematic effects this may have had on cliques in the resulting graphs, and on our results. The average density of the fifty 100-vertex graphs was about 0.47, with a large variance. Four of these graphs were nearly empty, while there was one occurrence of the complete graph on 100 vertices.

Nine heuristics were tested on each of the six sets, giving 2700 runs in all. For each 400-vertex graph, it took about two hours to run all nine. This time was dominated by the MFA heuristic.

The main hurdle in analyzing the results is that there is no easy way of computing the clique number of a given graph. The test graphs were too large to run any algorithm we know that always gives exact answers. For graphs G drawn from $\mathcal{G}_p(N)$, $\omega(G)$ can be estimated quite well by the quantity $c_{N,p}$:= the real number k such that $e_{N,p}(k) = 1$, where $e_{N,p}(k) := \binom{N}{k} p^{(k^2-k)/2}$ gives the expected number of cliques of size k in G. It was shown by D. Matula [Mat70] (see also [Pal85])[1] that for any p, the probability that a graph G drawn from $\mathcal{G}_p(N)$ does *not* have $|\omega(G) - c_p(N)| < 1$ approaches 0 as $N \to \infty$. The following table compares $c_p(N)$ to the average clique sizes and performance ratios returned by the two heuristics which fared the worst and the two which fared the best in our tests on p-random graphs. Details on the heuristics are in the next section, and the figures are accurate to two decimal places.

N	p	$c_p(N)$	SD($\emptyset$)	SSD($\emptyset, 1$)	SSD(V, N)	MFA	Perf. ratios
100	0.5	9.84	6.34	6.48	8.60	8.50	1.14–1.55
100	0.9	34.86	23.86	23.40	28.76	30.02	1.16–1.49
400	0.5	13.20	8.30	8.44	11.04	10.36	1.20–1.59
400	0.9	55.22	36.12	35.84	43.20	49.94	1.11–1.54

Unfortunately, no such estimate (of largest clique) exists for graphs drawn from $\mathbf{q}(x)$. To gauge how close to optimal a given run came, the best we could do was compare it with the largest clique size returned on that graph. This is one reason we ran as many heuristics as we did. Since the heuristic SSD(V, N) (described below) came out optimal or close to optimal in every case, we have calculated performance ratios relative to SSD(V, N) in Table IV. For consistency, Table IV also gives the performance ratios relative to SSD(V, N) for the p-random graphs.

[1] Matula's theorem is usually stated in terms of the estimate $d_p(N) := 2\log_b(N) - 2\log_b\log_b(N) + 1 + 2\log_b(e/2)$ for $c_p(N)$, where $b := 1/p$. For $p = 0.5$, $d_p(N)$ is quite close to $c_p(N)$ even for small N, but for $p = 0.9$ and the values of N we consider it is a serious underestimate: $d_{.9}(100) = 22.53$ not 34.86, $d_{.9}(400) = 43.86$ not 55.22.

5. The Neural Network Heuristics

Hopfield-clique Network. The Hopfield-clique Network (HcN) [Jag92] is a binary weights special case of the symmetric-weights zero-diagonal Hopfield Network (HN) [Hop82, Hop84]. HcN imposes the following restriction on the weights of HN: for $i \neq j$, $w_{ij} \in \{\rho, 1\}$ where $\rho < 0$. In the discrete version, unit states are: $S_i \in \{0, 1\}$. For $\rho < -N$ and $0 < w_0 < 1$, the stable states of discrete HcN are exactly the maximal cliques of its underlying graph G_N [Jag92]. This is the basis for applying discrete HcN to MAX-CLIQUE.

The concept of an *energy function*, which is central to Hopfield-type networks, makes possible a uniform description of several MAX CLIQUE heuristics, including some which have been studied without reference to Hopfield nets. The heuristics minimize the energy function during operation, thereby guaranteeing termination to a stable state (a maximal clique in the case of HcN).

Steepest Descent Heuristic. Steepest descent is a discrete heuristic (on discrete HcN), based on serial updates. At time t, the unit whose switch would reduce the energy maximally is switched. With $\rho < -2N$, SD(V_0) (V_0 is initial state) works as follows, in terms of G_N. Let V_i denote the vertex set in iteration i. In iteration i, if V_i is not a clique, then a vertex with minimum degree in $G[V_i]$ is removed from it; else a vertex in $V \setminus V_i$ adjacent to every vertex in V_i is added to it. Ties are broken lexicographically. The above algorithm converges in $\Theta(N)$ iterations. With $\rho < -2N$, SD(V) and SD($\emptyset$) emulate two well known greedy large-clique finding algorithms (MAX in [Gri83] and the "naive" one whose average-case performance Karp [Kar76] studied, respectively).

Rho-annealing. ρ-annealing [Jag92] is a discrete heuristic specific to (discrete) HcN. The idea is to vary ρ while performing SD. This is analogous to varying temperature T in simulated annealing; hence its name.

1. $\rho \leftarrow \rho_0 < 0 \quad [\rho_0 = -0.125]; \quad V' \leftarrow V;$
2. loop $\quad V' \leftarrow \mathrm{SD}(V', \rho);$ if $\rho < -N$ then exit; Decrease $\rho \quad [\rho \leftarrow 2\rho];$ endloop

The above description involves parameters; the square brackets contain the values we used in our experiments. Our graph characterization of SD when $\rho < -2N$ is inapplicable when $\rho \geq -N$. In this case, convergence to a stable state is guaranteed by the Hopfield convergence theorem [Hop82].

Stochastic Steepest Descent. Stochastic Steepest Descent (SSD) is a natural stochastic variant of SD, We have not seen it defined or tested in the context of MAX-CLIQUE. The deterministic moves of SD are replaced by gradient descent stochastic moves that favor the steepest direction. The motivation is, with initial state V, to improve upon the already good performance of SD(V) (see Table I; earlier discussion) without a significant risk of worsening it. The unit to be updated is selected via a probability distribution P that has zero probability of "up-hill" moves and favors large (steepest-descent like) decreases in energy. Specifically, let $C(t) \equiv \{i | \Delta E_i(t) < 0\}$ and ΔE_i the amount of energy the switch of i would decrease. Then

$$P[S_i \text{ is switched at time } t] = 0 \text{ if } \Delta E_i(t) \geq 0; \; = \frac{\Delta E_i}{\sum_{j \in C(t)} \Delta E_j} \text{ otherwise.}$$

P ensures that exactly one unit is switched. The probability of a switch is proportional to the amount of energy it decreases. We have a result that SSD converges in $\leq 2N$ unit switches (because it is gradient-descent). Non gradient-descent heuristics like simulated annealing are *much* slower. Let SSD(V_0, i) denote i runs of SSD on a given graph, with V_0 as input. The best clique found is chosen. We have empirically evaluated SSD$(\emptyset, 1)$, SSD$(V, 1)$, SSD$(\emptyset, N)$, and SSD(V, N).

Continuous Hopfield Dynamics. In this version of HcN, units are continuous: $S_i \in [0, 1]$. The first continuous heuristic is the continuous energy-descent dynamics (CHD) of Hopfield [Hop84] (see also [HKP91]) which we employ on it. It is described by a system of N coupled nonlinear differential equations (see [HKP91], page 54). Here we show its system of discretized difference equations (i.e. its Euler solution): $S_{n+1} := S_n + \gamma(-S_n + \bar{g}_\lambda(WS_n))$, where $S \in [0, 1]^N$ is the network state vector; $\bar{g}(\bar{x})$ is notational shorthand for $(g(x_i))$; $g_\lambda(x) := \frac{1}{1 + e^{-\lambda x}}$ is a sigmoid with λ its gain; and γ is the step size. The original system of equations minimizes an energy function [Hop84] thus always converging. As λ is increased, the stable states approach those of discrete HN [Hop84], hence maximal cliques in HcN. This is the basis for applying continuous HcN to MAX-CLIQUE. CHD is a significant dynamics for other reasons [Hop84, HKP91] and also a special case of MFA (see below). In the experiments in this paper, CHD was operated at $\rho = -10N$, $\lambda = 1$, $\gamma = 0.1$, and # iterations $= N$. The initial state was $S_0 = (0.5 + \delta)^N$ where δ is random noise in $[-0.05, 0.05]$. These are the same settings as in [Jag92] and not justified here. The only different setting is $w_0 := |\rho|/4$. In [Jag92], $w_0 = 0$ led to some problems.

Mean Field Annealing. The second continuous heuristic is Mean Field Annealing (MFA) [BMM+89, PS89]. MFA is continuous, deterministic, and performs gradient-descent (as opposed to simulated annealing (SA) [KJV83] which is discrete, stochastic, and does not perform gradient-descent). MFA is an efficient approximation to SA; careful experimental comparisons justify this remark [BMM+89]. MFA is also a generalization of CHD in which the parameter $T = \frac{1}{\lambda}$ is annealed (varied). For a better understanding of MFA and of its connections with simulated annealing (more generally statistical mechanics), mean field theory, MFA, and CHD see [HKP91, BMM+89, Jag92]. The algorithm is:

$T \leftarrow T_0$; Initialize S_0
repeat
    repeat $S_{n+1} := S_n + \gamma(-S_n + \bar{g}_{1/T}(W\, S_n))$
    until fixed point S_f is reached;
Decrease T; $S_0 \leftarrow S_f$
until T is sufficiently small.

In our experiments in this paper, the settings of ρ, λ, γ, and the initial state (S_0 at T_0) are the ones described for CHD above. The annealing schedule was geometric: $\boxed{T_n = a_{n-1} T_{n-1}; T_0 = \frac{2}{6} N |\rho|}$ where $a_n = 0.9; n \leq 3, a_n = 0.5; n > 3$. The settings are essentially the same as in [Jag92]. The only different setting is again $w_0 = \frac{|\rho|}{4}$. In [Jag92], $w_0 = 0$ led to some problems. Justifying the settings is infeasible here; our empirical results partly validate them.

6. Experimental Results

Table I and II report the detailed performance (size of clique returned) of our nine heuristics on 50 100-vertex and 50 400-vertex graphs respectively. The graphs are sampled from our approximation to the universal distribution. Each row corresponds to one graph. Comparisons of the column averages of Tables I and II (last row of each table) show the essential invariance of the relative performance of the algorithms. It also shows the heuristic pairs whose performance differences become exaggerated as the order N of the graphs increases.

Pairwise comparisons of heuristics.

- Comparison of SD($\emptyset$) and SD(V) columns shows the effect of the initial state: SD(V) is always better and usually by upwards of a factor of two.
- Comparison of CHD and MFA columns shows the improvement due to annealing: MFA is on most occasions better; sometimes same.
- Comparison of SSD($V, 1$) and SSD(V, N) columns shows the improvement due to N runs: SSD(V, N) is always much better.
- Comparison of SD(V) and ρ-annealing columns shows identical behavior mostly, but sometimes ρ-annealing does much better.
- Comparison of SD($\emptyset$) and SSD($\emptyset, 1$) columns shows the improvement due to randomization: SSD($\emptyset, 1$) is always moderately better.

Statistical Analysis of Variance. We further analyzed the performance of our heuristics (Tables I and II) by the program ANOVA (analysis of variance) using 'sas'—a statistical analysis package. Analysis was modeled as a randomized complete-block design: the graphs are the blocks and the heuristics are the treatments. A linear model with block-treatment interaction was used. That is:

$$X_{ij} = R_i + T_j + R_i \times T_j + \text{error}$$

where X_{ij} is the clique size returned by heuristic j on graph i. R_i is the contribution to X_{ij} because graph i was used; T_j is the contribution to X_{ij} because heuristic j was used. The results are:

1. On the data of Table I (fifty 100-vertex graphs), ANOVA found statistically significant differences. It clustered heuristics into four groups:
   A. SSD(V, N), SSD($\emptyset, N$), ρ-A, MFA, SD(V),
   B. CHD,
   C. SSD($V, 1$), SSD($\emptyset, 1$), and
   D. SD($\emptyset$).

2. On the data of Table II (fifty 400-vertex graphs), ANOVA found statistically significant differences. It clustered heuristics into three groups:

   A. SSD(V,N), SSD$(\emptyset,N)$, ρ-A, MFA, SD(V), CHD,

   B. SSD$(V,1)$, SSD$(\emptyset,1)$, and

   C. SD$(\emptyset)$.

In all of the clusterings, clusters are ordered (A,B,C,D) from best to worst and the within-cluster ordering is also from best to worst.

Table III reports the average performance of the nine heuristics on p-random graphs. Comparisons with Tables I and II show that SD$(\emptyset)$ performs relatively much better on p-random graphs than on graphs drawn from $\mathbf{q}(x)$. Also, on 0.9-random graphs (100- or 400-vertex) MFA is clearly the best heuristic—better than SSD(V,N) (On graphs drawn from $\mathbf{q}(x)$ both are tied for first place). The relative rankings of the heuristics are roughly the same in Tables I, II, and III; the differences are exaggerated in Tables I and II (compared to Table III). Comparisons of 0.5-random graphs versus graphs sampled from $\mathbf{q}(x)$ also show that the average size of the retrieved clique is significantly larger for the latter (even though the edge density is roughly the same). This is expected because the latter have more structure (are highly compressible).

Table IV shows the average performance ratio of each heuristic A relative to SSD(V,N). That is, the entries give SSD$(V,N)(G)/A(G)$ for each A other than SSD(V,N). The average is computed over the 50 graphs. The average performance ratio of SD$(\emptyset)$, SSD$(\emptyset,1)$, and SSD$(V,1)$ is significantly poorer on graphs drawn from $\mathbf{q}(x)$ than on p-random graphs. This difference gets exaggerated as the order of the graph increases. The other six heuristics exhibit remarkably little change in their performance ratios on p-random graphs versus graphs from $\mathbf{q}(x)$.

Table V shows the distribution of clique sizes for the nine heuristics on p-random graphs. The entries in each row give the number of graphs out of 50 on which the given heuristic A returned a clique of the size given in the column header. The following conclusions may be drawn:

- The variance is quite small in all cases. The distribution in each row is centered about the mean and resembles a binomial distribution. It is known [BE76] that the distribution of $e_{N,p}(k)$ (i.e., the expected number of cliques of size k in a graph drawn from $\mathcal{G}_p(N)$) is approximately binomial. Thus the averages given in Table III are good estimators of the true means.

- The variance increases markedly as p is increased; i.e., as the graph becomes denser.

- The variance of SSD$(\emptyset,N)$ and SSD(V,N) is significantly lower than the rest. This is expected because they employ N stochastic runs each.

- Comparisons of the pairs of rows of SD(V) and ρ-annealing provides further evidence that the two heuristics exhibit very similar performance.

References

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *The Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, page to appear, 1992.

[BE76] B. Bollobás and P. Erdös. Cliques in random graphs. *Proc. Camb. Phil. Soc.*, 80:419–427, 1976.

[BMM+89] G. Bilbro, R. Mann, T.K. Miller, W.E. Snyder, D.E. Van den Bout, and M. White. Optimization by mean field annealing. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1, pages 91–98, San Mateo, 1989. (Denver 1988), Morgan Kaufmann.

[BT85] B. Bollobás and A. Thomason. Random graphs of small order. *Annals of Discrete Math*, 28:47–97, 1985.

[CHd87] M. Chams, A. Hertz, and D. deWerra. Some experiments with simulated annealing for coloring graphs. *European J. Oper. Research*, 32:260–266, 1987.

[FGL+91] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Approximating clique is almost np-complete. In *The Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 2–12, 1991.

[Gri83] J.R. Griggs. Lower bounds on the independence number in terms of the degrees. *Journal of Combinatorial Theory, Series B*, 34:22–39, 1983.

[GS89] Y. Gurevich and S. Shelah. Nearly linear time. In *Proceedings, Logic at Botik*, Lecture Notes in Computer Science No. 363, pages 108–118. Springer-Verlag, 1989.

[HKP91] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, 1991.

[Hop82] J.J. Hopfield. Neural networks and physical systems with emergent collective c omputational abilities. *Proceedings of the National Academy of Sciences, USA*, 79, 1982.

[Hop84] J.J. Hopfield. Neurons with graded responses have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences, USA*, 81, 1984.

[Jag92] A. Jagota. Efficiently approximating max-clique in a hopfield-style networ k. In *International Joint Conference on Neural Networks*, volume 1, New York, 1992. Baltimore, June, IEEE.

[Jer90] M. Jerrum. The elusiveness of large cliques in a random graph. Technical report, Department of Computer Science, University of Edinburgh, August 1990.

[Kar76] R.M. Karp. The probabilistic analysis of some combinatorial search algorit hms. In J.F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 1–19. Academic Press, New York, 1976.

[KJV83] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220, 1983.

[LV90] M. Li and P.M.B. Vitanyi. Kolmogorov complexity and its applications. In J. van-Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 187–254. Elsevier and MIT Press, Amsterdam/New York, 1990.

[LV92] M. Li and P.M.B. Vitanyi. Average case complexity under the universal distribution equals worst-case complexity. *Information Processing Letters*, 42:145–149, May 1992.

[Mat70] D. Matula. On the complete subgraph of a random graph. In *Combinatory Mathematics and its Applications*, pages 356–369. Chapel Hill, NC, 1970.

[Mil91] P. Miltersen. The complexity of malign ensembles. In *The Proceedings of the 6th Annual IEEE Conference on Structure in Complexity Theory*, pages 164–171, 1991.

[Pal85] E.M. Palmer. *Graphical evolution.* Wiley, New York, 1985. Matula's theorem on page 76.

[PS89] C. Peterson and B. Söderberg. A new method for mapping optimization problems onto neural netw orks. *International Journal of Neural Systems*, 1:3–22, 1989.

[PW89] D. Petford and D.J.A. Welsh. A randomised 3-colouring algorithm. *Discrete Mathematics*, 74:253–261, 1989.

Table I — 100-vertex graphs sampled from $\mathbf{q}(x)$

Graph	SD($\emptyset$)	SD(V)	ρ-A	SSD($\emptyset, 1$)	SSD($V, 1$)	SSD($\emptyset, N$)	SSD(V, N)	CHD	MFA
G1	5	8	8	7	6	9	11	10	11
G2	10	29	29	29	29	29	29	29	28
G3	8	21	21	11	14	21	21	16	21
G4	4	9	9	4	7	8	9	4	8
G5	9	8	8	5	10	10	10	7	10
G6	17	40	40	36	30	40	40	40	40
G7	7	12	13	7	6	11	13	9	13
G8	5	6	6	5	5	6	6	6	5
G9	4	3	3	4	2	4	4	4	3
G10	5	10	10	8	5	10	10	4	8
G11	7	22	22	11	16	21	22	16	22
G12	3	27	27	6	8	27	26	27	27
G13	8	13	13	9	9	14	14	7	13
G14	5	7	7	4	4	7	8	4	7
G15	19	26	26	21	17	27	26	22	27
G16	4	3	3	4	2	4	4	4	3
G17	6	9	9	7	8	10	11	8	10
G18	15	31	31	4	23	31	31	31	31
G19	6	9	8	7	5	8	9	7	8
G20	15	31	31	31	31	31	31	31	30
G21	4	5	5	4	3	5	5	4	4
G22	9	26	26	13	16	26	26	21	26
G23	5	8	9	4	8	11	10	9	10
G24	5	9	9	8	7	10	10	8	9
G25	5	4	4	4	4	4	5	4	5
G26	19	42	42	32	40	42	42	36	41
G27	100	100	100	100	100	100	100	100	100
G28	5	7	9	6	6	8	8	7	7
G29	16	21	21	11	16	23	23	13	21
G30	5	9	9	11	8	12	12	9	10
G31	3	3	3	2	3	4	3	3	2
G32	15	39	39	18	21	39	39	38	38
G33	60	60	60	38	57	65	60	60	60
G34	8	8	8	6	4	13	13	5	7
G35	5	35	35	25	35	35	35	35	35
G36	8	10	11	8	10	11	11	7	11
G37	3	2	2	2	2	3	3	2	2
G38	8	13	13	7	9	13	13	7	13
G39	4	17	17	10	6	16	17	17	17
G40	8	29	29	27	11	27	29	29	29
G41	5	11	11	8	4	13	12	8	11
G42	4	5	5	5	3	5	5	5	4
G43	7	21	21	8	11	20	20	18	21
G44	2	26	26	26	24	26	26	26	25
G45	12	37	37	21	32	37	37	37	36
G46	3	6	6	2	3	5	6	5	6
G47	6	12	13	7	10	13	14	9	14
G48	4	6	6	4	6	7	7	5	6
G49	4	11	11	4	9	10	11	8	11
G50	6	10	10	7	8	12	13	10	13
Avgs.	10.20	18.20	18.42	12.96	14.26	18.76	19.00	16.62	18.38

Table II — 400-vertex graphs sampled from $\mathbf{q}(x)$.

Graph	SD($\emptyset$)	SD(V)	ρ-A	SSD($\emptyset, 1$)	SSD($V, 1$)	SSD($\emptyset, N$)	SSD(V, N)	CHD	MFA
G1	6	16	16	9	6	12	15	16	15
G2	12	90	90	16	12	90	90	90	90
G3	12	36	39	10	27	34	40	39	41
G4	16	166	166	130	144	163	166	166	166
G5	20	130	130	100	40	130	130	130	130
G6	10	31	33	20	22	27	34	26	29
G7	8	26	26	19	17	27	26	23	27
G8	4	43	43	11	21	43	43	41	43
G9	4	23	23	9	5	22	23	22	23
G10	8	25	30	9	13	29	30	20	23
G11	11	74	74	28	35	72	74	74	74
G12	3	68	68	47	23	68	68	67	65
G13	8	35	35	17	6	32	34	34	34
G14	10	25	25	14	11	25	25	15	24
G15	11	93	93	85	65	93	93	63	62
G16	21	51	51	35	55	65	65	65	64
G17	47	115	115	64	115	114	115	83	114
G18	41	141	141	92	100	139	138	122	141
G19	4	11	11	5	10	13	14	10	14
G20	7	27	27	11	11	27	22	16	28
G21	12	80	82	24	39	82	82	61	80
G22	15	40	58	39	52	76	76	76	57
G23	6	59	59	15	29	53	59	56	59
G24	2	132	132	75	131	132	132	132	132
G25	4	35	35	7	7	34	35	34	33
G26	5	14	20	20	12	25	24	21	27
G27	6	18	18	9	8	14	15	8	16
G28	135	135	135	6	97	135	135	135	135
G29	60	71	71	28	67	72	72	72	72
G30	5	14	14	9	12	14	14	11	12
G31	192	226	226	222	223	225	226	205	225
G32	3	125	125	111	105	125	125	124	123
G33	11	13	13	12	10	13	13	13	12
G34	9	16	17	10	9	16	16	16	17
G35	8	25	24	11	27	25	27	23	25
G36	6	27	27	22	10	29	27	12	26
G37	25	56	56	56	28	56	56	20	57
G38	4	22	22	5	10	21	21	20	22
G39	166	193	188	197	177	200	203	174	194
G40	9	28	28	51	10	51	51	28	27
G41	11	92	93	62	78	93	93	80	93
G42	17	56	42	30	20	55	56	40	43
G43	13	17	29	17	19	33	30	17	29
G44	19	57	57	43	41	59	58	46	58
G45	6	52	62	17	30	62	62	58	62
G46	8	50	70	40	32	69	69	43	69
G47	4	7	7	7	3	7	7	7	6
G48	5	58	58	6	12	58	58	58	58
G49	2	59	59	42	29	59	59	59	58
G50	14	58	76	56	54	74	75	58	77
Avgs.	20.90	61.22	62.78	39.6	42.38	63.84	64.42	56.58	62.22

Table III — Average performance on p-random graphs

N	p	SD($\emptyset$)	SD(V)	ρ-A	SSD($\emptyset, 1$)	SSD($V, 1$)	SSD($\emptyset, N$)	SSD(V, N)	CHD	MFA
100	0.5	6.34	7.98	8.06	6.48	6.42	8.36	8.60	7.44	8.50
100	0.9	23.86	28.16	28.34	23.40	24.82	27.60	28.76	27.92	30.02
400	0.5	8.30	9.88	10.34	8.44	8.24	10.80	11.04	9.16	10.36
400	0.9	36.12	43.80	44.58	35.84	36.82	41.86	43.20	43.24	49.94

Table IV — Average performance ratio SSD$(V, N)(G)/A(G)$ on all graphs

Source	SD($\emptyset$)	SD(V)	ρ-A	SSD($\emptyset, 1$)	SSD($V, 1$)	SSD($\emptyset, 1$)	SSD(V, N)	CHD	MFA
$\mathbf{q}(100)$	2.42	1.10	1.09	1.79	1.60	1.03	—	1.31	1.10
$\mathcal{G}_{.5}(100)$	1.35	1.08	1.07	1.37	1.32	1.02	—	1.15	1.01
$\mathcal{G}_{.9}(100)$	1.20	1.02	1.01	1.23	1.15	1.04	—	1.03	0.96
$\mathbf{q}(400)$	7.56	1.09	1.03	2.64	2.19	1.02	—	1.24	1.05
$\mathcal{G}_{.5}(400)$	1.33	1.12	1.07	1.31	1.33	1.02	—	1.21	1.02
$\mathcal{G}_{.9}(400)$	1.19	0.99	0.97	1.21	1.17	1.03	—	1.00	0.87

Table V — Distributions of clique sizes found for p-random graphs

$N = 100$, $p = 0.5$

Heuristic	5	6	7	8	9	10
SD($\emptyset$)	8	23	13	6		
SD(V)		1	13	24	10	2
ρ-A			11	27	10	2
SSD($\emptyset, 1$)	4	24	16	6		
SSD($V, 1$)	7	13	24	5		
SSD($\emptyset, N$)				32	18	
SSD(V, N)				21	28	1
CHD		6	21	18	5	
MFA			2	24	21	3

$N = 100$, $p = 0.9$

Heuristic	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
SD($\emptyset$)	2		3	5	10	13	10	2	2	3						
SD(V)							3	6	11	8	11	8	1	1		1
ρ-A							2	5	14	5	10	9	3	1		1
SSD($\emptyset, 1$)	1	2	3	12	11	8	3	7	2	1						
SSD($V, 1$)		2	1	5	6	8	9	7	6	4	1	1				
SSD($\emptyset, N$)							1	12	12	11	12		1	1		
SSD(V, N)								4	4	15	12	11	1	2	1	
CHD				1		1	2	8	6	15	11	1	2	1	1	1
MFA									1	7	12	12	10	6		2

$N = 400$, $p = 0.5$

Heuristic	6	7	8	9	10	11	12
SD($\emptyset$)		10	22	11	7		
SD(V)			2	18	16	12	2
ρ-A			1	5	24	16	4
SSD($\emptyset, 1$)	1	2	25	19	2	1	
SSD($V, 1$)		10	21	16	3		
SSD($\emptyset, N$)					10	40	
SSD(V, N)					1	46	3
CHD			9	27	11	3	
MFA				4	27	16	3

$N = 400$, $p = 0.9$

Heuristic	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53
SD($\emptyset$)			1	4	7	6	9	12	6	3	1	1												
SD(V)											1	3	7	9	16	7	3	3	1					
ρ-A											1	6	4	13	10	11	4	1						
SSD($\emptyset, 1$)			1	3	9	10	10	8	4	3	2													
SSD($V, 1$)	1		1	2	3	5	11	10	5	6	2	2	2											
SSD($\emptyset, N$)											3	14	20	13										
SSD(V, N)													9	27	10	3	1							
CHD										2	1	7	7	9	9	10	5							
MFA																	1	1	7	10	14	8	7	2

Appendix I - Tutorial on sampling from $\mathbf{q}(x)$

The procedure for sampling from the description-length weighted distribution $\mathbf{q}(x)$ is as follows:

1. A random binary string s in $\{0,1\}^m$ is generated.
2. s is decoded as a pair (P,x), where P is a Gurevich-Shelah program which when run on x produces a binary string y.

s is thus a description of the string y. Informally speaking, the above procedure is repeated and all strings y such that $|y| >> |s|$ are recorded. The obtained strings y are highly compressible and hence represent samples from a description-length weighted distribution, namely $\mathbf{q}(x)$.

The s-to-(P,x) decoder. The decoder interprets s as a "self-delimiting" description for the program P and binary string x. It has been designed with care to ensure that most Gurevich-Shelah programs are reasonably likely, given the length m of s. Here, we present only the main ideas.

1. The first $\lfloor(\log m)/2\rfloor$ bits of the m-bit string s are interpreted as storing the length l of x. The next l bits are pulled off for the argument x itself. Let r be the number of bits which remain.
2. The first $\lfloor(\log r)/2\rfloor$ bits of the remaining r bits are interpreted as storing the number of basic Gurevich-Shelah functions in P.
3. Let nc denote the number of functions found as above. The next $\lfloor(\log nc)/2\rfloor$ bits are interpreted as storing the number of iterated replacements in the program string. These iterated replacements are represented in the program string by placing parentheses around the functions involved in the replacement.
4. The next set of bits indicates where parentheses for iterated replacements are located.
5. The next set of $3nc$-many bits identify the basic functions in the program string. Three bits are used to encode each occurrence of the eight basic functions.
6. The next set of bits is decoded as a self-delimiting value of the total length of the parameter strings for each basic function.
7. The remaining bits are decoded and divided up among the parameters for each basic function according to the syntax of the operator and the length decoded in the above step.

An Example. The following example was obtained from an actual run, using the following command.

```
rsb 1 30 10 | nlt_dec 30 | nlt 100

s   =       01001011110000001001101110010 0
x,P =       00 D 011 E 10 01 R0 0 0  .
y   =       1010
```

In this case, the output y is shorter than s, and so not very useful. 'rsb' generates the random string s of length 30 with seed 10 (third command-line argument). 'nlt_dec' decodes s. It takes $|s|$ as the command-line argument. In its output, the first string is x; the rest P. In P, the operators are identified by name and followed by their parameter strings. The processing is left-to-right; the input to each operator is the output of the previous one (or x for the first operator). 'nlt' runs P on x.

Access to Code. The above code is accessible via ftp as follows:

```
ftp ftp.cs.buffalo.edu (128.205.32.9 subject-to-change)
Name : anonymous
> cd users/jagota
> get nlt.README
> quit

nlt.README provides further instructions.
```

If there are problems accessing the code, contact jagota@cs.buffalo.edu.

Tips on usage. We think that our programs may be used to generate highly compressible inputs for any application whose inputs are binary strings (vectors) or other structures (e.g. graphs in our case) that can be obtained from binary strings.

1. Though the sampling is quite efficient (one of roughly every six seed strings in our case generated a highly compressible string), the running time of 'nlt' scales poorly as the size of y increases. The current version of 'nlt' uses the direct quadratic-time method of computing iterated replacements, rather than the near-linear time tree-structured method sketched in [GS89]. In general, 'nlt' has not been optimized for search operations.

2. For this scheme to work well, the seed strings should be reasonably large (upwards of 70 bits in our case). Smaller strings produce less variability in the programs P. Couple this with the desirability of high compression (e.g squareroot in our case), the target output should preferably be upwards of 1000 bits (4950 in our case).