Performance of Neural Network Algorithms for Maximum Clique On Highly Compressible Graphs

Arun K. Jagota *
Department of Computer Sciences
University of North Texas
Denton, TX, 76203, USA

Kenneth W. Regan †
Dept. of Computer Science
SUNY-Buffalo, USA

Abstract

The problem of finding the size of the largest clique in an undirected graph is NP-hard, even to well-approximate, in the worst case. Simple algorithms, including some we study here, work quite well however, on graphs sampled from $\mathbf{u}(n)$, the uniform distribution on n-vertex graphs. It is felt by many, however, that $\mathbf{u}(n)$ does not accurately reflect the nature of instances that come up in practice. It is argued that when the actual distribution of instances is unknown, it is more appropriate to suppose that instances come from the Solomonof-Levin or *universal* distribution $\mathbf{m}(x)$ instead, which assigns higher weight to instances with shorter descriptions (i.e., to those that are structured or compressible). We extend a theorem of Li and Vitanyi to show that the average-case performance of any approximation algorithm on random instances drawn from $\mathbf{m}(x)$ has the same asymptotic order as its worst-case performance. Because $\mathbf{m}(x)$ is neither computable nor samplable, we employ a realistic analogue $\mathbf{q}(x)$ which lends itself to efficient empirical testing. We experimentally evaluate how well certain neural network algorithms for

*The first author was involved in this research while at the Dept of Computer Science, SUNY-Buffalo
†The second author was supported in part by NSF Grant CCR-9011248.

Maximum Clique perform on graphs drawn from $\mathbf{q}(x)$, as compared to those drawn from $\mathbf{u}(n)$. The experimental results are as follows. All nine algorithms we evaluated performed roughly equally-well on $\mathbf{u}(n)$, where as three of them—the simplest ones—performed markedly poorer than the other six on $\mathbf{q}(x)$. Our results suggest that $\mathbf{q}(x)$, while postulated as a more realistic distribution to test the performance of algorithms than $\mathbf{u}(n)$, might also discriminate their performance better. Our $\mathbf{q}(x)$ sampler can be used to generate compressible instances of any discrete problem.

1 Introduction

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 [2, 15]) and are applicable to MAX-CLIQUE.

Recently it has been shown that even calculating $\omega(G)$ to within a given constant factor is NP-hard [1]. In fact, the main result of [1] gives a fixed $\epsilon > 0$ such that if there is a polynomial-time algorithm A which gives $\omega(G)/A(G) \leq n^{\epsilon}$ for all G of sufficient size n, then NP = P. Nevertheless, there are simple heuristics which come to within a factor of 2 of optimal for "most" graphs, in the sense of the uniform distribution $\mathbf{u}(n)$. This distribution is defined on n-vertex undirected graphs, by letting each edge (i, j), $(1 \leq i < j \leq n)$, exist independently at random with probability 1/2. When the probability is p instead of 1/2, we denote the distribution as $\mathbf{u}_p(n)$ instead. For example, Theorem 8 of R. Karp [10] implies that for any $\epsilon > 0$ and sufficiently large n, the heuristic we call SD-$\emptyset$ gives an expected performance ratio $E[\omega(G)/\text{SD-}\emptyset(G) \;:\; \mathbf{u}_p(n)]$ which is less than $2 + \epsilon$.

However, it is felt in many quarters that the uniform distribution does not accurately reflect the nature of instances which come up in practice. M. Li and P. Vitanyi [11] argue that when the actual distribution of instances is unknown, it is most appropriate to suppose that they come from the non-computable Solomonoff-Levin or *universal* distribution $\mathbf{m}(x)$. One reason is that every computable distribution is majorized by a constant multiple of $\mathbf{m}(x)$. Another reason related to *Occam's Razor* is that the objects occurring most often in nature or in practice have short descriptions. 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 of Py is appreciably less than the bit length $|x|$ of x. The shorter descriptions x has, the more weight is given to $\mathbf{m}(x)$. By contrast, uniform distributions favor instance strings x which are incompressible; i.e. whose shortest descriptions are essentially "PRINT x." Li and Vitanyi [12] show that with respect to $\mathbf{m}(x)$, the average-case running time of any algorithm whatsoever has the same order as its worst-case running time. In Section 2 we obtain the same result with performance ratios by approximation algorithms in place of running time.

The main purpose of our work is to test whether certain algorithms perform noticeably worse under distributions weighted toward compressible inputs. We also wish to ascertain which algorithms perform poorly on such distributions and which perform well. Section 3 describes our efficient approximation $\mathbf{q}(x)$ to $\mathbf{m}(x)$. Section 4 describes the algorithms we tested, ranging from simple to sophisticated neural network heuristics. All these algorithms were tested extensively on random graphs in [8], and several of them on a variety of graphs with different kinds of structures in [9], in which paper extensive comparisons were made with other algorithms. Section 5 describes the experimental methodology, which includes a description of the test graphs, a discussion of the criterion for evaluating performance, and a description of the parameter values of certain parametrized algorithms. In Section 6 we present the experimental results and their analysis.

The main contributions of this paper are:

- Theorem 1, extending Li and Vitanyi's result to approximation performance.
- A method for testing algorithms under an efficient approximation to the universal distribution. To our knowledge, this is the first concerted effort at testing under such non-uniform distributions.

- Experimental results (Section 6) which show that, while all nine heuristics performed roughly equally-well under $\mathbf{u}(n)$, three of the simpler ones performed markedly poorer than the other six under $\mathbf{q}(x)$. For example, SD-$\emptyset$, the simplest heuristic, retrieved on average a clique size within a factor of 1.30 of the one retrieved by the best heuristic, on graphs drawn from $\mathbf{u}(n)$ (see Table 3). By contrast, the size of the clique retrieved by SD-$\emptyset$ was on average poorer by a factor of 3 than the one retrieved by the best heuristic, on graphs drawn from $\mathbf{q}(x)$ (see Table 2).

- Experimental results (Section 6) which reveal some interesting characteristics of certain individual algorithms. For example, a simple variant of SD-$\emptyset$, only a very little more sophisticated than SD-$\emptyset$, ended up performing much better on $\mathbf{q}(x)$. One of the continuous neural network methods, MFA, which has been claimed in the neural network literature to work well in practice, did indeed work very well on $\mathbf{q}(x)$. The other continuous method, CHD, was discernably poorer than MFA but significantly better than SD-$\emptyset$ and the other two simplest algortihms, on $\mathbf{q}(x)$.

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 [11] 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," so that all other programs begin with '1', and every string $x \in \{0,1\}^n$ has the description $0x$ of length $n+1$.

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 [12], we show:

Theorem 1 *For any algorithm A there is a constant $c > 0$ 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,

$$w(H_n) \geq \sum_{y \ : \ |y| \leq n+1-p} 2^{-(p+|y|)} = \frac{n+1-p}{2^p}. \tag{3}$$

Hence

$$E[\omega(G)/A(G) \;:\; \mathbf{n}] \geq wc(A,n)\cdot \mathbf{n}(H_n) = wc(A,n)\frac{w(H_n)}{W(n)} \geq \frac{wc(A,n)(n+1-p)}{2^p n} \tag{4}$$

$= \Theta[wc(A,n)]$. □

Remark. The time taken by P to compute G_n is exponential in n. P. Miltersen [13] 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 $E[\omega(G)/A(G) \;:\; \mathbf{u}]$.

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?*. A systematic answer to this question is beyond the scope of this paper. Instead we address a weaker version of this question:are instances with short descriptions hard *for certain algorithms?* The rest of this paper presents a means of testing this question, a description of the algorithms we tested, and our experimental results that give an answer to this question within the scope of our investigations.

3 The $\mathbf{q}(x)$ Sampler

We use a functional programming notation $\mathcal{L}$ due to Y. Gurevich and S. Shelah [4] 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 $x \in \{0,1\}^n$:

$$\begin{aligned} w_{\mathbf{q}}(x) &:= \sum 2^{-|z|} \ : \ |z| \leq |x|+1, U(z,|x|) = x, \\ W_{\mathbf{q}}(n) &:= \sum w_{\mathbf{q}}(x) \ : \ x \in \{0,1\}^n, \\ \mathbf{q}(x) &:= w_{\mathbf{q}}(x)/W_{\mathbf{q}}(n). \end{aligned} \tag{5}$$

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 the uniform distribution. Our choice of quadratic compression was partly motivated by the hard-to-approximate graphs in [3, 1]. 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 [3]), two of which are joined by an edge iff the oracle answers in the respective transcripts do not contradict each other. It is an open question whether the "O(1)" can be reduced to nearly-linear; even if so, the bit-size of $G_{M,x}$ is still bounded below by $\frac{1}{2}m^2$ (with judicious padding just in case M rejects x).

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.The seeds were recorded to make the experiments repeatable. See the Appendix for implementation details.

The following description of the eight basic string functions from [4] 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.

$R0_{u,y}(x):$ 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.

There are two constructors: functional composition, and "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 [4] 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 $\mathbf{m}$. However, it should be pointed out that whereas the parameter strings $u, v, \ldots$ are fixed in individual $\mathcal{L}$-programs, the definition of $\mathbf{q}$ effectively quantifies over them. This allows for quadratic and greater expansions. 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 the output.

The main practical reason for using $\mathcal{L}$ is its simplicity and ease of implementability. 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 The Neural Network Algorithms

All neural network algorithms evaluated in this paper are based on the Hopfield model [6, 7], and are described in detail in [8]. Here we describe them only briefly, without explaining their neural network implementation in much detail. It is worth noting that all these algorithms arise as manifestations of essentially a single meta-algorithm: one that minimizes the usual *energy function* in the Hopfield model [6, 7].

4.1 Discrete Algorithms

Steepest Descent. Steepest Descent (SD) is a discrete serial-update neural network heuristic that minimizes energy in greedy fashion. In each time step, the unit to switch decreases energy by the maximum amount. We use the notation $SD(V_0)$ to denote that Steepest Descent starts initially from some subset $V_0 \subseteq V$ of vertices. SD iteratively transforms V_0 into a maximal clique C, terminating efficiently within $2n$ iterations [8]. Let V_i denote the vertex-set in iteration i and assume that it is not a maximal clique. SD emulates the following heuristic in iteration i:

If V_i is not a clique then

a vertex with minimum degree in the induced subgraph $G[V_i]$ is removed from V_i

else if V_i is a clique then

a vertex in $V \setminus V_i$ adjacent to every vertex in V_i is added to V_i.

Ties are broken lexicographically.

ρ-annealing. ρ-annealing is another discrete serial-update neural network heuristic, which works by carrying out annealing while minimizing energy. More precisely, a certain parameter of the network, called ρ, is varied while the network minimizes energy by steepest descent. This is analogous to varying the temperature T in simulated annealing. We omit the precise description of ρ-annealing here, for which the reader is refered to [8]. An intuitive description is as follows.

1. Start with small ρ and with the initial state $V_0 := V$.

2. Run SD(V_0) with this value of ρ to transform V_0 into U.

3. Increase ρ, set $V_0 := U$, and go to step 2.

The algorithm is terminated when ρ becomes sufficiently large. It turns out that when ρ is small, the set U retrieved in step 2 is not required to be a clique; however as ρ is increased, certain constraints get ever tighter, ultimately forcing U to be a clique. In other words, like simulated annealing, this algorithm starts with loose constraints—allowing an unbiased exploration of the search space—and progressively tightens them until the final solution U forms a clique. A precise characterization of the behavior of this algorithm is in [8].

Stochastic Steepest Descent. Stochastic Steepest Descent (SSD) is a randomized variant of SD. The deterministic moves of SD are replaced by energy-minimizing moves that favor the steepest direction, but probabilistically. More precisely, The unit to switch is picked with probability proportional to the amount of energy its switch would decrease. (The probability is zero if the switch would keep the energy same or increase it.)

The algorithm is motivated by the desire to randomize the choice of unit to switch, which allows one to use repeated runs of the algorithm to boost the size of the clique found, while not totally relinquishing the greedy heuristic emulated by SD, which often works well (see Tables 1 and 2, and [8]).

Let $SSD(V_0, i)$ denote i runs of SSD on a given graph, with V_0 as the initial state (vertex set) in each run. (Note that the initial state is the same in each run.) The largest clique found in a run is the output of the algorithm. One run of SSD terminates within $2n$ unit-switches (iterations) [8], which keeps one run as efficient as SD.

4.2 Continuous Algorithms

The description of the continuous algorithms assumes familiarity with the continuous Hopfield model [7].

Continuous Hopfield Dynamics. This algorithm, called the continuous Hopfield dynamics (CHD) [7, 5], is described by a system of n coupled nonlinear differential equations, presented here in discretized form:

$$S(t+1) := S(t) + \gamma(-S(t) + \bar{g}_\lambda(WS(t) + I)) \tag{6}$$

Here $S_i \in [0, 1]$ is the state of the i^{th} neuron, I_i the external bias of the i^{th} neuron, W the $n \times n$ symmetric weight matrix, $g_\lambda(x) = \frac{1}{1 + e^{-\lambda x}}$ a sigmoid with gain λ, $\bar{g}(\bar{x})$ notational shorthand for $(g(x_i))$, and γ the Euler step size. The continuous-time version of (6) minimizes an energy function during its evolution [7], into which the MAX-CLIQUE problem can be encoded [8]. With sufficiently large λ and sufficiently small γ, if (6) is started from any initial state $S(0) \in [0,1]^n$ and iterated sufficiently-many times, it provably terminates at a fixed point S from which a maximal clique of the encoded graph can be recovered [8].

For a discussion of the significance of CHD from the point of view of neural implementation and optimization applications, see [7, 5, 8]. CHD is especially interesting because it may be viewed as the essential special case of the algorithm presented next—a continuous optimization method developed only recently, but one that is already beginning to make its mark on optimization as it occurs in practice.

Mean Field Annealing. The second continuous heuristic, called *Mean Field Annealing* (MFA) [2, 15], may be described as a generalization of CHD in which the sigmoidal gain λ is varied during the evolution of (6). This is done by employing an annealing schedule, a sequence $\{\lambda_i, \mu_i\}$ of k elements, where λ_i is the value of the sigmoidal gain and μ_i the number of times (6) is to be iterated with the sigmoidal gain set at λ_i. Usually λ_i is a monotonically increasing function of i. The detailed algorithm is as follows.

$S := S(0)$

for $i := 1$ to k do

  for $j := 1$ to μ_i do

    $S := S + \gamma(-S + \bar{g}_{\lambda_i}(WS + I)$

With sufficiently small γ and sufficiently large μ_i, S converges to a fixed point at each value of i [7, 5]. Additionally, with sufficiently large k, and with λ_i growing sufficiently slowly with i, MFA is known to deterministically approximate simulated annealing during its evolution [2, 5], while being more efficient.

5 Experimental Methodology

All experiments on the neural network algorithms and their evaluation on $\mathbf{u}(n)$ and $\mathbf{q}(x)$ were performed on a SUN SparcStation I.

Details of the Test Graphs. Experiments were performed on 100-vertex graphs and on 400-vertex graphs. The bitstrings of the 100-vertex graphs had length 4950, and those of the 400-vertex graphs had length 79,800. For $n = 100$ and $n = 400$, three sets of fifty n-vertex graphs 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 recorded to make the experiments repeatable.

The third set was generated using seed strings of lengths 65..85 for the fifty 100-vertex graphs and eleven of the 400-vertex graphs, and 270..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 greater than $79,800$. 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 in any significant way. It took about 12 hours 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, and so ended with many repetitions of u. We do not have an intuitive idea of the extent to which this yielded repeated patterns in the graph. 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. The MFA heuristic was by far the slowest of the lot.

Sample Sizes. Each set of test graphs contained fifty graphs. It is reasonable to ask if this sample size is adequate. For graphs drawn from $\mathbf{u}_p(n)$, several arguments lead to the conclusion that a sample size of fifty graphs is more than adequate for our purposes. First, the expected size of the maximum clique in a graph drawn from $\mathbf{u}_p(n)$ has a sharp threshold [14] and the range of sizes of maximal cliques in such graphs is also quite narrow. Thus, any maximal-clique finding algorithm, for example most of the ones in the current paper, is guaranteed to find a clique in a narrow range. This argument is buffeted by experimental results reported in [8], which give the distribution of clique sizes found in fifty graphs drawn from $\mathbf{u}_p(400)$, $p = 0.5, 0.9$, which turns out to have a very small variance.

For graphs drawn from $\mathbf{q}(x)$, however, it was not clear a priori what an adequate sample size should be. We decided to start with a sample size of fifty. On this sample size, the results reported in Tables 1 and 2 (see Section 6 for their presentation and analysis) displayed certain trends so clearly and consistently that we felt confident that our observations were sound and would remain basically unchanged on larger sample sizes.

Evaluating Performance. The main hurdle in analyzing the results is that there is no easy way of calculating the size of the largest clique in a graph. We could have used some exponential-time algorithm to find the exact answer, but this would have been quite time-consuming on the 2700 runs. Therefore, instead of comparing the *absolute* performance of these algorithms on $\mathbf{u}(n)$ versus those on $\mathbf{q}(x)$, we decided to compare their *relative* performances, in particular how well or poorly certain algorithms performed relative to others, on $\mathbf{u}(n)$ versus $\mathbf{q}(x)$.

Parameter Settings of the Continuous Algorithms. The continuous algorithms—CHD and MFA—use certain free parameters whose values needed to be set. The values that we used are described below to make the experiments independently repeatable. One needs to refer to [8] in order to understand some of the parameters.

CHD was operated at $\rho = -10n$, $\lambda = 1$, $\gamma = 0.1$, $I_i = |\rho|/4$ for all i, and with the number of iterations of (6) fixed in advance to n. The initial state to CHD was set to $S(0) := (0.5 + \delta)^n$, where δ was a random value in $[-0.05, 0.05]$. The settings are the same as in [8], and are motivated there.

MFA was operated with the same settings for ρ, λ, γ, I, and the initial state $S(0)$ as was CHD, and with the following geometric annealing schedule:

$$T_i = a_{i-1}T_{i-1}; T_1 = \frac{2}{6}n|\rho|$$

where $a_i = 0.9$ for $i \leq 4$ and $a_i = 0.5$ for $i > 4$. Here $T_i = 1/\lambda_i$. The settings are essentially the same as in [8], and are motivated there. Experimental results in [8] also reveal that CHD and MFA continue to work well on the graphs tested in [8], on parameter settings in a reasonably large neighborhood of those described above.

6 Experimental Results

Tables 1 and 2 give the size of the clique retrieved by each of the nine algorithms, on fifty 100-vertex and fifty 400-vertex graphs sampled from $\mathbf{q}(x)$ respectively.

SD($\emptyset$) is the steepest descent algorithm whose initial state is the empty set. It emulates the naive heuristic:

> Start from the empty set and extend it, by adding, in each step, one suitable vertex selected lexicographically, until it forms a maximal clique.

SSD(∅,1) is a randomized version of SD(∅) in which the vertex to be added is selected randomly, from the feasible choices, instead of in lexicographic fashion. SD(V) is the steepest descent algorithm whose initial state is the entire vertex set V. It turns out that SD(V) emulates the following algorithm [8]:

```
S := V
while S is not a clique do
    Pick a vertex v ∈ S with minimum degree in S
    Delete v from S
endwhile
while S is not a maximal clique do
    Pick the lexicographically first vertex v ∉ S adjacent to every vertex in S
    Add v to S
endwhile
```

SD($V, 1$) is a randomized version of SD(V) in which the vertex to be deleted in an iteration of the first loop is picked with probability proportional to $S - \text{degree}_S(v)$ (the smaller the degree, the higher the probability), and the vertex to be added in an iteration of the second loop is picked at random from the feasible choices. SSD(∅,n) and SSD(V,n) are multiple restart versions of SSD(∅,1) and SSD(V,1) respectively—the largest clique found in the n runs is output.

From Tables 1 and 2, the following observations can be made:

- SD($\emptyset$) works the poorest. SSD($\emptyset$,1) and SSD(V,1) work moderately better but remain significantly poorer than the best algorithms. Thus, randomization alone helps but not as much as one might expect.

- SD(V) works much better than SD($\emptyset$) and nearly as well as the best algorithms. Thus replacing the initial state of SD($\emptyset$) by V, which makes the SD algorithm greedier, improves the performance much more dramatically than by randomizing SD($\emptyset$) alone. Randomizing SD(V), to get SSD(V,1), in fact worsens the performance significantly.

- The ρ-annealing algorithm consistently works just slightly better than SD(V).

- The multiple restart algorithms—SSD($\emptyset$,n) and SSD(V,n)—work the best, with SSD(V,n) working just very slightly better. This shows that the real benefit of randomization is that it allows multiple restarts, which boosts performance immensely.

- The continuous algorithm CHD works moderately poorer than the continuous algorithm MFA, which was anticipated, but also works discernably poorer than the discrete algorithm SD(V), which was not anticipated.

These observations hold for both Tables 1 and 2—if anything, the effects are more pronounced in Table 2 than in Table 1.

From these results we may cluster the algorithms into four groups, using the size of the clique found as the measure. In order of decreasing performance, the clusters are:

1. SSD(V,n), SSD($\emptyset$,n), ρ-annealing, MFA, and SD(V).

2. CHD.

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

4. SD($\emptyset$).

Table 3 gives the clique sizes found by these nine algorithms on random graphs, i.e. graphs drawn from $\mathbf{u}_p(n)$. The relative rankings of the algorithms in Tables 1, 2, and 3 are mostly the same, though there is one notable exception, explained later in this paragraph. The performance differentials between these algorithms are however far wider on graphs drawn from $\mathbf{q}(x)$ than on graphs drawn from $\mathbf{u}_p(n)$. On graphs drawn from $\mathbf{u}_{1/2}(n)$, the clique sizes obtained by all nine algorithms are in a small range. SD($\emptyset$) remains the poorest working algorithm, but only marginally so. On graphs drawn from $\mathbf{u}_{0.9}(n)$, the range of clique sizes gets larger and so do the performance differentials, although most of the relative rankings remain unchanged. The one notable exception is the MFA algorithm which performs significantly better than all the others. Although MFA worked very slightly poorer than the multiple restart algorithms on graphs drawn from $\mathbf{q}(x)$ it works significantly better on graphs drawn from $\mathbf{u}_{0.9}(n)$. Another interesting feature of the graphs drawn from $\mathbf{u}_{1/2}(n)$ versus those drawn from $\mathbf{q}(x)$ revealed by the algorithms is that, whereas both types of graphs have roughly the same density, the algorithms, on average, retrieve much larger cliques on graphs drawn from $\mathbf{q}(x)$ than on graphs drawn from $\mathbf{u}_{1/2}(n)$. Certainly this is not surprising as it is reasonable to expect a correlation between the compressibility of graphs drawn from $\mathbf{q}(x)$ and the fact that they contain large cliques.

The *performance ratio* of algorithm A relative to algorithm B on graph G is defined as the clique size found by B divided by the clique size found by A. Table 4 gives the performance ratio of each algorithm relative to $SSD(V,n)$, averaged over graphs drawn from $\mathbf{q}(x)$ and over graphs drawn from $\mathbf{u}_p(n)$. $SSD(V,n)$ is chosen as the reference algorithm because it works best on graphs drawn from $\mathbf{q}(x)$. The results reported in Table 4 are drawn from the earlier tables. The results viewed in this fashion tend to support our earlier observations in more dramatic fashion. For example, SD($\emptyset$) has a poor performance ratio, 2.42, on graphs drawn from $\mathbf{q}(100)$, which worsens markedly, to 7.56, on graphs drawn from $\mathbf{q}(400)$. By contrast, SD($\emptyset$) has much better performance ratios on graphs drawn from $\mathbf{u}_p(100)$, $p = 0.5, 0.9$, which remain unchanged on graphs drawn from $\mathbf{u}_p(400)$, $p = 0.5, 0.9$.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

[8] A. Jagota. Approximating maximum clique in a Hopfield-style network. *IEEE Transactions on Neural Networks*, 6(3):724–735, 1995.

[9] A. Jagota, L. Sanchis, and R. Ganesan. Approximating maximum clique using neural network and related heuristics. In D.S. Johnson and M. Trick, editors, *DIMACS Series: Second DIMACS Challenge*, to appear. AMS, January 1995.

[10] 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.

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

[12] 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.

[13] 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.

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

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

Appendix: A Practical Guide to the $\mathbf{q}(x)$ Sampler

In this appendix we describe the $\mathbf{q}(x)$ sampler at a level appropriate for users who might consider evaluating their algorithms on test data sampled from $\mathbf{q}(x)$. At the end of this section, we also give instructions on how to fetch the software.

A macro-level description of the sampling procedure is as follows:

Algorithm $\mathbf{q}(x)$-`Sampler`

1. Generate a random string s in $\{0,1\}^m$.

2. Decode s to give a pair (P, x), where P is a program in the formalism explained in Section 3 and x is a binary string.

3. Run the program P on the binary string x and output the binary string y.

The string s is then a description of the string y.

In practice, the $\mathbf{q}(x)$-`Sampler` is called repeatedly and only those strings y whose lengths exceed that of s by a certain threshold are kept, all others are thrown away. This ensures that the remaining strings are compressible by a certain minimum amount.

The Implementation of Step 2

The algorithm to decode a string s into a pair (P, x) was designed to ensure that almost every program that could be the target of some string s' of length m had nonzero probability to be the target of a random string s. The decoding procedure is sketched below.

1. Interpret the first $\frac{\lfloor \log m \rfloor}{2}$ bits of s as storing the length $|x|$ of x.

2. Interpret the next $|x|$ bits as storing x.

3. Let r denote the number of bits remaining at this point. Interpret the first $\frac{\lfloor \log r \rfloor}{2}$ bits of these remaining r bits as storing the number of non-iterated operators in the program P.

4. Let o denote the number of operators found above. Interpret the next $\frac{\lfloor \log o \rfloor}{2}$ bits as storing the number of iterated operators. These are obtained by placing balanced parentheses around subsequences of non-iterated operators.

5. Interpret the next set of bits to indicate where the parentheses should be placed.

6. Interpret the next set of $o \times 3$ bits as the actual codes of the non-iterated operators. There are 8 such operators, so 3 bits suffice to identify individuals amongst them.

7. Interpret the next set of bits to give the length of every parameter to every operator.

8. Interpret the remaining bits as the actual parameters for all the operators, using the knowledge of the syntax of every operator and the knowledge of the length of every parameter computed in the previous step.

An Example

This example was constructed from an actual run using the software discussed later. The following command was invoked at the `Unix` command line.

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

The program `rsb` generates a random string of length 30 using 10 as the seed. The program `nlt_dec` decodes this string, taking the string length as a command-line argument, and outputs the string x, followed by the program P. The operators in P are identified by name and followed by their parameter strings. The operators in P are sequenced under the assumption that P is to be executed from left to right, with the first operator taking x as the input and producing an output string, which becomes an input to the second operator and so on. The program `nlt` interprets P, and executes it on the string x.

The actual strings and program produced in this example were:

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

The output y in this instance turned out to be shorter than s.

How to Fetch the $\mathbf{q}(x)$ Sampler

The $\mathbf{q}(x)$ sampler may be retrieved by anonymous ftp as follows:

```
ftp ftp.cs.buffalo.edu
Name: anonymous
> cd users/jagota
> get nlt.README
> quit
```

The file `nlt.README` provides further instructions.

Other Important Usage Details

The $\mathbf{q}(x)$ sampler can be used to generate compressible instances of any discrete problem, whether on binary strings, or on structured objects such as graphs. One needs to be keep in mind the following issues however.

1. The good news is that the generation of compressible strings is quite efficient. In our case, one of every six seed strings turned out to generate a highly compressible string. To ensure this efficiency, however, the seed strings need to be longer than a certain minimum length. We found that seed strings of length less than 70 almost never generated compressible strings y, perhaps because the programs P that result from such strings are too short and trivial to generate compressible strings.

2. The bad news is that our implementation of `nlt` is quite inefficient because the code that performs certain internal searches that need to be performed in order to execute certain operators in P is not optimized.