Testing Neural Net Algorithms on General Compressible Data

Arun K. Jagota †, Kenneth W. Regan ‡
† Department of Computer Sciences, University of North Texas, email: jagota@cs.unt.edu
Denton, TX, 76203, USA
‡ Department of Computer Science, State University of New York at Buffalo
Buffalo, NY, USA

***Abstract*— This paper advocates that a "realistic" evaluation of a algorithm should include tests not only on "random" data, but also on *compressible* data. We evaluate a suite of nine neural-net heuristics for the MAX CLIQUE problem on both uniformly-random and compressible data, using a time-bounded analogue q of the *Solomonoff-Levin universal distribution* m for the latter. The Maximum Clique problem is NP-hard to solve approximately in the worst-case; however it is easy to approximate within a factor of 2 on "random" graphs. Three of the neural algorithms, while nearly competitive with the other six on random data, are observed to work poorly under q, for graphs with 100 and 400 vertices. This gives some indication that the compressible distribution q discriminates the performance of various algorithms better than does random data. However, the results obtained here are preliminary and not definite. We suspect that a theoretical result of Li and Vitanyi, namely that the distribution m asymptotically draws out worst-case behavior in an algorithm, shows up as a practical phenomenon, but the results so far do not strongly support this.**

1 Introduction

Most algorithm testing is done either on "random" instances of the problem at hand, or on a relatively narrow range of instances tied to a particular application. Our first purpose is to point out that "random" instances are not necessarily *realistic* instances. Our second purpose is to shed more light on the possibility of doing scientific testing in ways that retain both realism and generality over many applications. This is an abstract of our still-preliminary efforts to show that neural-net heuristics behave markedly differently on "random" data versus data drawn from distributions that attempt to reflect how instances arise in practice. A full current version of our work is available on the Internet at `ftp://ETC`.

Our point of departure is the common-sense position that real-world instances are *not* "random." The usual meaning of "random" is (1) that the instance is drawn from the uniform distribution on the space of all possible instances. For example, a "random" n-vertex undirected simple graph is obtained by flipping a coin for each pair (i, j) $(1 \leq i < j \leq n)$ to see whether edge (i, j) is present in the graph. (Note that the underlying sample space will have many isomorphic copies of the same graph.) Information theory provides a second perspective: (2) a random instance is one that is *incompressible*; i.e., one whose standard encoding to a solving algorithm cannot be replaced by an appreciably shorter string encoding. These two meanings mostly align for our purposes, since (by a standard counting argument) the vast majority of strings cannot be appreciably compressed. Thus a "random" string from (1) will, with high probability, be one of the strings in (2). For example, while some special n-vertex graphs such as the n-cycle can be compressed to $O(\log n)$ bits under fairly general "smart encoding" schemes, the standard encoding length of $N = (n^2 - n)/2$ bits cannot be improved for the vast majority of graphs.

Note that (2) does not pre-suppose a distribution or an instance space. Instead, it leads to one, called the *Solomonoff-Levin distribution* $\mathbf{m}$. The basic idea of $\mathbf{m}$ is that if a graph (or any string) g_1 is compressible to K bits while g_2 cannot be compressed below N bits, then g_1 is 2^{N-K} times as likely to occur as g_2. The two main scientific ideas behind $\mathbf{m}$, which have recently been expounded at length in articles and a book by Li and Vitanyi [?, ?, 11, ?], are:

1. Compressible inputs occur much more frequently than "random" ones.
2. In the absence of any specific knowledge about the distribution of inputs to a computational problem, one should assume that they come from $\mathbf{m}$.

The latter statement is justified by a theorem that for every computable distribution $\mathbf{d}$ on strings, there is a constant C depending only on $\mathbf{d}$ such that for all strings x, $\mathbf{m}(x) \geq \mathbf{d}(x)/C$. That is, $\mathbf{m}$ makes all strings at least as likely, within the constant factor C, as $\mathbf{d}$ did. The former expresses that real-world data sets usually have a much shorter specification than the number of data points entered or generated. For instance, graphs having specific regularities tend to present themselves to be solved more frequently than do "scattershot" graphs. For both reasons, $\mathbf{m}$ is also called the *universal prior distribution*.

The major rub is that $\mathbf{m}$ itself is not a computable distribution; indeed, there is no computable way to sample according to $\mathbf{m}$, or even to estimate $\mathbf{m}(x)$ within a constant factor—all attempts will undershoot $\mathbf{m}(x)$ on many x. However, we *can* attempt to sample in ways intended to preserve many of the important properties of $\mathbf{m}$. Whereas $\mathbf{m}$ is based on a universal programming system for all computation, we define a distribution $\mathbf{q}$ based on a programming system that is universal only for those computations that can be done in time $n \cdot (\log n)^{O(1)}$, which is called *quasi-linear* time. Li and Vitanyi themselves suggested

polynomial-time sampling, but times above n^2 are not really efficient in practice. Quasi-linear time includes many important computational primitives, including sorting and FFT, that are not known to be computable in strictly-linear time. Our **q** assigns higher weight to size-N structures that can be (compressed and) decompressed in time that is quasi-linear as a function of N (not of K). Scientifically, the relevance of **q** amounts to asserting that real-world instances have regularities that can be efficiently perceived and produced. In our actual sampling, we made some further compromises described and justified below.

The main spur to our work was the following theoretical result of Li and Vitanyi [11]:

> **Theorem.** The average-case running time under **m** of any terminating algorithm is within a constant factor of its *worst*-case running time over all inputs of a given size.

With essentially the same proof, we show in our full report that the same statement can be made about optimization performance: any algorithm with instances drawn according to **m** fares nearly as poorly as it does on the worst instances. The proof idea is mainly that "the worst instance of size n" is a ($\log(n)$+const)-sized description of itself, and so gets high weight under **m**—hurting the **m**-average more than "the best instance" helps it. Miltersen [?] shows that, subject to something like P $\neq$ NP, no polynomial-time computable distribution can be "malign" to the same degree as **m**. Hence **q** itself is not so malign; note also that the above description does not give a $qlin(n)$ time way to decompress itself. Despite this and the way the proof of the Li-Vitanyi result borders on a triviality, we suspect that the above is the iceberg's tip of a phenomenon that compressible instances as they arise in the real world really tend to be harder than "random ones." The particular phenomenon we test is:

> **Test Question.** Is the performance of heuristic algorithms for problems markedly poorer under **q** than under uniform distribution?

The particular computational problem we study is the MAX CLIQUE problem: given an undirected graph G, compute $\omega(G)$, which stands for the maximum size of a clique in G—and furthermore, find a clique of that size. This is a well-known NP-hard problem. We study it for several reasons:

1. Many important computational problems can be efficiently transformed into cases of the MAX CLIQUE problem.
2. Clique problems are well-suited to neural nets. There is a wide variety of heuristics that can be tested.
3. A great deal is known about the theory of cliques in graphs, especially under uniform distributions. This theory acts as a scientific control on our experiments.

For a practically-minded treatment of the first point see [?], while the second is borne out by the algorithms described below. We discuss the third point now. The main theoretical results about cliques in random graphs are:

> **Theorems.** With very high probability, an n-vertex graph selected under uniform distribution will have maxim*um* clique size about $2\log_2 n$, and will also have no maxim*al* cliques of size less than $\log_2 n$ at all.

Indeed, according to *Matula's Theorem* [?], $\omega(G)$ clusters into two adjacent integer values near $2\log n$. Since any halfwitted algorithm can be expected at least to output a maxim*al* clique, the second clause says that all algorithms come within a factor of 2 of optimum on random graphs. Karp [10] suggested that all polynomial-time algorithms eventually fall down to that factor of 2 on random graphs as n gets large. Practical tests on random graphs up to several thousand vertices do much better than a factor of 2 and often come within 10%; see [?, 12] and our own results under **u** below. Jerrum and Sinclair [?] give strong theoretical as well as empirical evidence that any fall-down for a particular heuristic based on the *Metropolis process* does not happen for graphs of less than 80,000-to-a-million vertices. Similar results are known for the distributions $\mathbf{u}_p(n)$ under which edges are independently present with probability p, $0 < p < 1$.

Now we argue that in "real-world" clique instances G, the critical size of $\omega(G)$ that one needs to know about is more like n/C or $n^{1/2}$, *not* so low as $\log n$ or $2\log n$. The standard transformation from an m-clause, n-variable instance ϕ of 3SAT to an equivalent instance of MAX CLIQUE produces a graph G_ϕ of $3m+2n$ vertices that has a clique of size $n+m$ iff ϕ is satisfiable. Common transformations from other combinatorial optimization problems also have target clique sizes in the n^ϵ-to-$\Omega(n)$ range. This higher range also figures into the celebrated non-approximability results of Feige et al. [?] and Arora et al. [1] and later refinements. These results produce graphs G such that in the "yes" case, G has a clique of size n^e, while in the "no" case, G has no clique of size n^{e-d}, where $e > d > 0$ are fixed constants. It is now known that d can be larger than $e/3$, with the conclusion that (unless NP = P) $\omega(G)$ cannot be approximated within a factor of $\omega(G)^{1/3}$, let alone a factor of 2, in the worst case [?]. The graphs in all these results have only $O(n) = O(N^{1/2})$ non-edges ($6m + n$ non-edges in the first) and hence are compressible to $O(N^{1/2})$ size. These facts led us to do our actual testing on the "slice" of **q** afforded by graphs decompressed from seeds of sizes in some interval around $N^{1/2}$.

Thus the MAX CLIQUE problem gives a sharp distinction between "random" and "real-world" instances. This distinction applies also to *pseudo*random instances, insofar as the pseudorandom generators (PSRGs)

that produce them are intended to replicate all properties of uniform distribution that are subject to feasible statistical tests. Note that graphs G produced by a PSRG are compressible, since they are described by the relatively short seed string s used and the number i such that the ith iteration of the PSRG on s produced G. However, such PSRG graphs may still be only a tiny fraction of those produced by a *universal* decompressor, such as the programming system on which **q** is based. This distinction seems borne out by our statistics below; the **q**-graphs had much larger clique sizes than the "random" graphs produced by the standard UNIX PSRG. Note also that we are not claiming that all compressible instances are hard, but only that "general compressible data," where "general" refers to a universal distribution such as **m** or **q**, is harder on average than random data.

Still, this distinction does not say anything about the behavior of heuristic algorithms on these respective kinds of instances. However, the above theory gives us a good control benchmark for a positive answer to our above **Test Question**.

> **Benchmark.** We consider a heuristic algorithm A to perform "markedly poorer" under **q** than under **u** if its average performance ratio $\omega(G)/A(G)$ under **q** falls below the factor-of-2 guarantee under **u**.

In test cases where one cannot compute $\omega(G)$ exactly, and one is testing a suite of algorithms of varying strengths, one can use the stronger algorithms A' to control the weaker ones, by using $A'(G)/A(G)$ instead as the term for G in the average computed for **q**. This is a "conservative" pragmatic choice and cannot produce a "false positive" result. Using the exact clique solving algorithm of D. Johnson and others at AT&T Bell Labs [**?**], we were able to compute $\omega(G)$ for all the 100-vertex graphs we generated, but on 17 of the fifty 400-vertex graphs (all with larger clique sizes), this algorithm did not halt within a day's computing time (per graph). ⟨**Stub:** *State your actual experience, and add the cite of Johnson.*⟩ Note also that by Matula's Theorem above, we were able to simply use $2\log_2 n$ in place of $\omega(G)$ for all the graphs G generated under **u**, with a clear conscience.

Our *results* for the suite of nine neural algorithms that we tested are summarized in Tables 1 and 2 below. Tables of results on the individual **q**-graphs are available in the full report. There were three positive answers, but the other six heuristics performed just as well under **q** as under **u**. Hence our best conclusion about whether the Li-Vitanyi principle sits atop a real practical phenomenon is "maybe: needs more testing on larger graphs."

Our results do support the conclusion that **q** discriminated the quality of the heuristics better than **u** did. Here we remark that our approach is not limited to performance on optimization problems—it is equally applicable to *learning* problems, which are the main focus of much current mainstream neural network research. Our general position is:

> If one has tested an algorithm on random data and it performs well, one should also test it on compressible data. One may find that the good results on random data do not transfer over to compressible data. This may give a better indication of how the algorithm will perform on real data than the results on random data alone.

Section 2 describes the neural algorithms in full detail; this section has independent interest. Section 3 gives more detail of the methodology of our experiments. Section 4 discusses the results and conclusions further.

2 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].

2.1 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.

2.2 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:

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

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 (1) 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 (1) 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, 13], may be described as a generalization of CHD in which the sigmoidal gain λ is varied during the evolution of (1). 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 (1) 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.

3 Experimental Methodology

(**Stub:** *A brief description of "The* $\mathbf{q}(x)$ *Sampler" should be moved here. We need not give the actual Gurevich-Shelah operations—a reference to our full report and the actual software being available by ftp is enough. It's enough to say: there are 8 operations plus parameters; they are easy to program; they have general uses. And some down-sides: at our seed sizes, the effects were dominated by the two ops that expand strings; the four ops that alter bits in strings had only limited (though perhaps sometimes significant) play. Thus our "slice of* $\mathbf{q}$*" was rather grainy.*)

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 [12] 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 (1) 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.

4 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($\emptyset$,1) is a randomized version of SD($\emptyset$) 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($\emptyset$,n) and SSD(V,n) are multiple restart versions of SSD($\emptyset$,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] B. Bollobás and P. Erdős. Cliques in random graphs. *Proc. Camb. Phil. Soc.*, 80:419–427, 1976.

[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*, page To Appear. AMS, January 1995. Proceedings of the Second DIMACS Challenge: Cliques, Coloring, and Satisfiability.

[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. Average case complexity under the universal distribution equals worst-case complexity. *Information Processing Letters*, 42:145–149, May 1992.

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

[13] 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.

Table 1: Average performance on p-random graphs. This table is excerpted from Table I in [8].

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 2: Average performance ratio SSD$(V, n)(G)/A(G)$ on all graphs. The numbers for the $\mathbf{q}$ graphs are obtained by averaging over the performance ratios, not by taking the performance ratio of the average.

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