SOLVING THE SATISFIABILITY PROBLEM USING MESSAGE-PASSING TECHNIQUES

S. J. Pumphrey, May 2001 (Part III Physics Project Report)

A Bayesian message-passing technique, the sum-product algorithm, is described and applied to the satisfiability problem, specifically 3-SAT. The results of this algorithm on randomly generated 3-SAT problems and standard SAT test sets are discussed and compared with conventional techniques. The algorithm is found to perform poorly in regions of the parameter space in which it is conventionally 'hard' to find solutions. Several adaptations to the basic sum-product algorithm are thus examined in an attempt to improve its effectiveness in these regions. Finally, the successes and failures of the algorithm are discussed, and conclusions drawn as to the direction of future research in this field.

1. INTRODUCTION

The propositional satisfiability (or SAT) problem is one of the oldest and most researched problems in computer science and computational physics. It is one of the major problems in machine vision, and has applications in many fields of artificial intelligence, including natural language parsing, task scheduling, 3D object semantics, and logical reasoning[12]. A technique to rapidly solve SAT problems would revolutionise the field of computational science. However, such an algorithm is thought impossible and the best achievable is likely to be a method which finds a solution *most* of the time.

The objective in a SAT problem is to find an assignment to a set of Boolean[1] variables such that a logic statement about those variables is 'satisfied', i.e. *true*. At first sight this can seem a fairly abstract problem, so consider the following example[13]:

> *You are chief of protocol for the embassy ball. The crown prince instructs you to either invite Peru or to exclude Qatar. The queen asks you to invite either Qatar or Romania or both. The king, in a spiteful mood, wants to snub Peru or Romania or both. Is there a guest list that will satisfy the whims of the entire royal family?*

In this simple SAT problem, we can identify the three binary variables as whether or not Peru, Qatar, and Romania are invited. The question can then be more concisely stated: we want to find an assignment to *P*, *Q*, and *R* which satisfies the logic formula,

$$(P \vee \neg Q) \wedge (Q \vee R) \wedge (\neg P \vee \neg R), \tag{1}$$

where the symbols $\wedge$, $\vee$, and $\neg$ correspond to the logical operations AND, OR, and NOT, respectively.

In this case, the problem is so small that it can be quickly solved by brute force methods: we can just try all the possible assignments until one satisfies the logic. For instance, let us start by attempting to invite all three countries. This satisfies the crown prince's request that we invite Peru, and the queen's desire to invite both Romania and Qatar, but we find that we can't fulfil either of the king's wishes. If we instead consider inviting Peru and Qatar only, we find that this is a solution to the problem; the logic is *satisfiable*. Note that a second solution exists, in which only Romania is invited, and, for instance, if there were a further restriction that we couldn't invite both Peru and Qatar, then this would become the unique solution.

This is a fairly contrived example, but it demonstrates one method for solving the problem: try every possible assignment to the binary variables until the logic is satisfied. As the number of binary variables increases, however, the number of possible assignments increases exponentially and so even at fairly small numbers of variables, the problem quickly becomes intractable.

[1] A Boolean, or binary, variable, *X*, is one which takes the value $x \in \{true, false\}$.

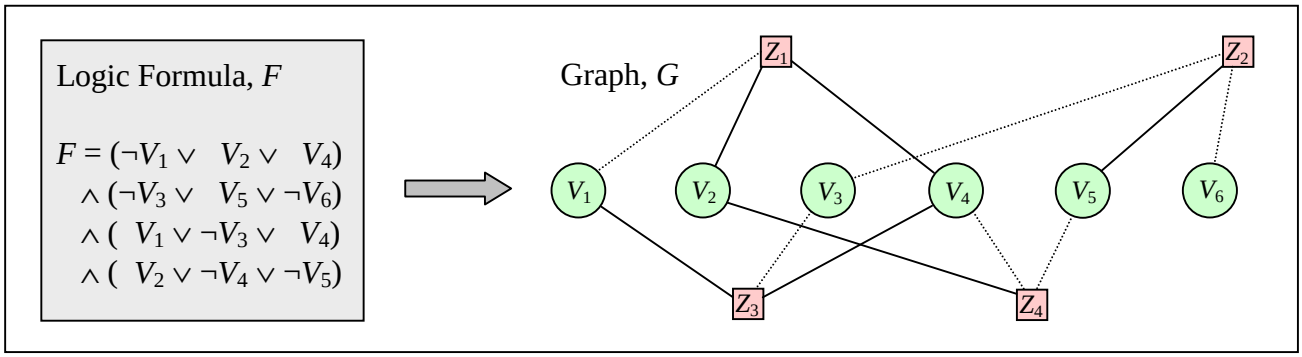

Fig. 1. Example of the type of graph used in this investigation, and its formation from a 3-SAT problem. Six *propositions*, $\{V_1, \ldots, V_6\}$, (circles) are shown, connected to four *checks*, $\{Z_1, \ldots, Z_4\}$, (squares) by arcs which are either 'normal' (solid) or 'negated' (dotted). The value of a *check* is calculated from an assignment to $\{V_i\}$ by performing the logical OR of all the connected propositions, negating those connected by 'negated' arcs. In a satisfying assignment all *checks* will be *true*.

In order to compare methods for solving such problems, the concept of *computational complexity* is important. The usual way of specifying this is to state how the time taken to solve a problem scales with some measure of the 'size' of that problem. For instance, the number of steps to find a solution to a SAT problem, using the brute force method above, scales as 2^n and hence it is termed 'exponential complexity'.

Two classes of computational complexity are of interest, called 'P' and 'NP', where 'P' $\subseteq$ 'NP', and for which it is believed 'P' $\neq$ 'NP'. Problems in which the complexity is polynomial (e.g. n, n^2, or even n^{500}) are members of class 'P', and are treated here as 'easily' solvable. The SAT problem, however, along with many other 'hard' problems, is in the class 'NP', and, more specifically, it has been shown that it is *NP-complete*[4]. This is one of the most important types of computational problems, since any one NP-complete problem can be converted to any other NP-complete problem in polynomial time. Hence, a polynomial time solution to the SAT problem would provide a polynomial time solution to a host of other problems, including the travelling salesman problem[16], automated computer chip design[7], and the general decoding problem for error correcting codes[18].

This investigation focuses on the application of *Bayesian inference* to the problem of finding the *most probable* assignment to the variables, *given* that the logic is satisfied. More specifically, the technique examined is to consider the Boolean variables as vertices on a graph, and the logic clauses as a network of edges and '*checks*' connecting those vertices (see *Fig. 1*). The assignment that most likely satisfies the logic statements can then be determined by sending 'messages' along the edges about the probabilities of the variables being *true* or *false* given that the *checks* are *true*. An analysis of this technique, and how well it performs in comparison with more conventional algorithms, forms the basis of this study, but we will first describe the SAT problem in more detail and review some standard approaches to its solution.

2. PROPOSITIONAL SATISFIABILITY

2.1 Terminology and Definitions

A logic *function*, or *formula*, is a general combination of Boolean variables (referred to as *propositions*) and logical operators. The same function can often be rewritten in several different forms, but here we will always use *conjunctive normal form* (CNF). A function in conjunctive normal form is composed of one or more *clauses*, joined by the AND ($\wedge$) operator. Each *clause* consists of one or more *literals*, joined by the OR ($\vee$) operator, where each *literal* is either a single proposition or its negation (X or $\neg X$). For example, the embassy ball problem, eq. (1), is in conjunctive normal form, where the three clauses are the wishes of the three royals.

Conjunctive normal form is useful in the context of satisfiability since the clauses can be interpreted as a set of separate logic statements which must be individually satisfied by the assignment. This approach then simplifies the calculation of whether the formula is satisfied by an assignment, as each statement is *true* if any literal in that statement is *true*.

A standard subclass of the propositional satisfiability problem is the *k*-SAT problem, in which each logic clause contains exactly *k* literals. For $k = 2$ the problem has a polynomial-time solution, but for $k > 2$ the problem is NP-complete, and hence, without loss of generality, we focus exclusively on 3-SAT.

In this work, we define N to be the number of propositions and M to be the number of clauses. We can then define, for a 3-SAT problem:

- $V = \{V_1, \ldots, V_N\}$ to be the set of N propositions.
- $L = \{V_1, \neg V_1, \ldots, V_N, \neg V_N\}$ to be the set of possible literals for V.
- $C = \{C_1, \ldots, C_M\}$ to be the set of clauses, where $C_i = \{l_1, l_2, l_3 \in L\}$ is a set of three literals.
- $Z = \{Z_1, \ldots, Z_M\}$ to be the set of '*checks*', i.e. the values of $\{C_i\}$ given an assignment to $\{V_i\}$.

The aim of the SAT problem is thus to find an assignment to the $\{V_i\}$ such that the $\{Z_i\}$ are all *true*.

A central theme of this investigation is the use of *inference* to help solve such logic problems. This is a technique by which it is possible to calculate the probability of parameters, θ, given data, D, and it is performed using Bayes' theorem:

$$P(\theta \mid D) = \frac{P(D \mid \theta)P(\theta)}{P(D)}, \tag{2}$$

where $P(\theta \mid \mathrm{D})$ is known as the '*posterior probability*', $P(D \mid \theta)$ is the '*likelihood*', $P(\theta)$ is the '*prior probability*', and $P(D)$ is a normalising constant.

In terms of the SAT problem, we are trying to infer the probabilities that the propositions, $\{V_i\}$, are *true*, given that the logic is satisfied, i.e. $P(V_i = true \mid Z_j = true\ \forall j)$. Hence, we define the prior probabilities, $\{g_i\}$, as our prior knowledge of the probabilities that propositions $\{V_i\}$ are *true*. Given a random SAT formula, we know nothing beforehand, in general, about any satisfying assignments to V, and so we set each g_i to $g = 0.5$. This is known as using a *non-informative* prior.

2.2 Conventional SAT Solvers

Algorithms to solve the SAT problem can be described as either *complete* or *incomplete*. Complete solvers are guaranteed to return a result as they explore the entire problem space[2], though as a result they all suffer from a worst-case exponential computational complexity. Incomplete solvers, on the other hand, may not find a result as they employ, in general, heuristics and randomness to guess likely assignments, and then explore the local problem space around those assignments. On certain types of problem the incomplete algorithms are good in that they find solutions very quickly. In large, hard problems, however, it is quite possible that these solvers 'miss' the area of the problem space containing the solution and yield no result.

Complete solvers do not, generally, attempt to attack a SAT problem by trying all possible assignments. Instead, the most common approach, based upon the technique of '*backtracking*', is called the *Davis-Putnam-Logemann-Loveland* (DPLL) procedure[5][17]. It works by assigning a value to each of the propositions in turn, and, with each assignment, 'removing' that proposition from the logic. This is achieved by, for each of the clauses the proposition appears in, removing the clause if it is satisfied by the assignment, or else removing the *false* literal from the clause. If, at any point, an assignment results in an empty clause, i.e. all literals in it are *false*, then it 'backtracks' to

[2] We use the term 'problem space' here to refer to the state space of all possible assignments.

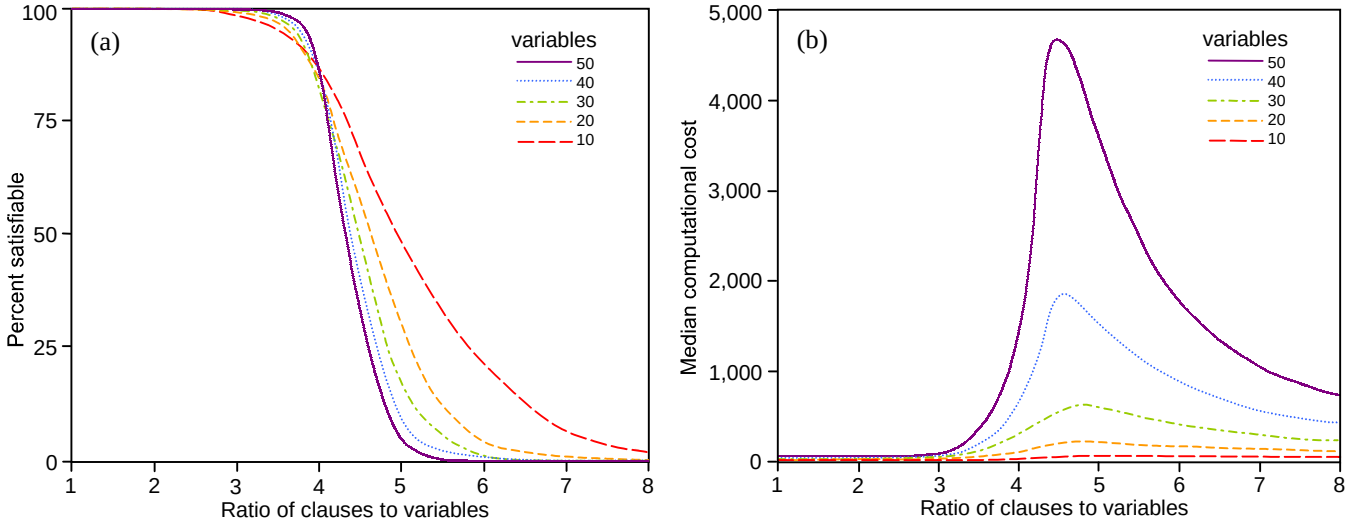

Fig. 2. (a) Graph of percent satisfiable vs *r* for random 3-SAT. (b) Graph of median 'cost' of finding solutions vs *r* for random 3-SAT. (Adapted from [13]).

the last decision it made and tries a different one. This will scan the entire problem space, and it is more efficient than simply testing each possible assignment, as it can prune whole branches off the search tree without having to explore to the leaves.

In contrast, most incomplete solvers implement a *stochastic local search* (SLS), in which a random initial assignment is adjusted iteratively so as to satisfy as many of the clauses as possible[14]. Some implement sophisticated heuristics as to which propositions to adjust in each iteration, but the overall effect is to explore the local state space around the initial assignment, and they therefore do not explore the whole search tree.

A great deal of research has gone into methods for solving the SAT problem, and as a result many complete and incomplete algorithms exist. In order to compare the performance of different algorithms (and to encourage more work on new algorithms), the 'DIMACS challenge' [8][15] test sets were introduced. These consist of a number of hard satisfiability problems, which we will use later in this investigation as a benchmark for the message-passing SAT algorithm.

2.3 SAT Problem Characteristics

In the past decade, much work has gone into attempting to characterise the behaviour of the SAT problem as a function of the parameters *N* and *M*. The difficulty has been that it appears to be very sensitive to how the clauses are chosen, and, for a given method, sometimes the resulting problem is very easy and sometimes it is very hard[1]. For anything but the smallest *N* and *M*, it is not possible to try every possible combination, and so it is necessary to resort to statistical techniques.

The paradigm is to use 'random 3-SAT' formulas, in which the clauses are generated by choosing three propositions at random from *V*, and either negating a proposition or not with probability 0.5. If hundreds of such formulae are then generated and tested, the average number with solutions is found to be a characteristic function of *N* and *M*. It is observed that the behaviour depends on the ratio $r = M / N$, and that at a particular value of *r*, $r \approx 4.3$, the problem seems to undergo a *phase transition* from always finding a solution to never finding one (see *Fig. 2*(*a*))[3][10]. For small *r*, the problem is *under-constrained* and, as there are many solutions, they are easy to find. For large *r*, the problem is *over-constrained* and it is easy to show that a solution doesn't exist. For *r* around the phase transition, however, finding a result becomes very hard indeed, and a graph of the 'cost' of finding a result shows a peak (*Fig. 2*(*b*)). The discovery of this phase-transitional behaviour has resulted in parallels being drawn between the behaviour of a SAT problem and several physical systems, including '*spin glasses*'[2], and it is hoped that these comparisons will lead to novel approaches to both solving SAT and understanding the physical systems in the future.

3. MESSAGE PASSING

We here outline a graph-based treatment of the SAT problem, in which the logic and propositions are arranged into a '*belief network*' (also known as a *Bayesian network*), and the probabilities are iteratively updated by message-passing along the edges. The specific algorithm under study is the '*sum-product algorithm*', which was first used by Gallager in 1963 on the decoding problem for error correcting codes[9]. The algorithm as applied to the SAT problem is sketched below, with emphasis primarily on practical aspects (see Appendix for some further mathematical details).

3.1 Definitions

In this study, we consider the undirected bipartite[3] graph $G(\{V, Z\}, \{E_a, E_b\})$ with vertices $\{V, Z\}$ and edges $\{E_a, E_b\}$, where V and Z are as defined earlier[4], and E_a and E_b are two distinct sets of edges, joining the V and Z vertices, and determined by the clauses, C. If proposition V_i appears un-negated in clause C_j, then the arc (V_i, Z_j) will be in E_a, else, if it appears negated in C_j, the arc will be in E_b. Thus, we identify arcs in E_a as 'normal' and arcs in E_b as 'negated'.

We will need to be able to enumerate the arcs at each vertex, and, for each arc on a given vertex, we need to be able to specify the index of that arc at the connected vertex. For example, consider *Fig. 3*(*a*) in which vertex V_1 has four arcs, and vertex Z_1 has three. The 'first' arc at V_1 connects to Z_1, but it is the 'second' arc at Z_1 which connects to V_1. In order to simplify the discussion below, we introduce the following notation:

- let $c(i, j)$ be the index of the *check* connected to *node* V_i by the jth arc from V_i,
- let $d(i, j)$ be the index of the *node* connected to *check* Z_i by the jth arc from Z_i,
- let $e(i, j)$ be the index of the arc **from** $Z_{c(i, j)}$ **to** V_i, and
- let $f(i, j)$ be the index of the arc **from** $V_{d(i, j)}$ **to** Z_i.

Using this notation, the example above can be rewritten as: $c(1, 1) = 1$ and $e(1, 1) = 2$.

Finally, we define the *belief network* on which the sum-product algorithm is to be performed as a set of conditional probability *vectors* over G, where each vertex is associated with a vector, and each element of that vector is related to an arc connected to that vertex. Mathematically, we can represent this in the form $B(G, \{\boldsymbol{Q}, \boldsymbol{R}^{(x)}\})$, where $\boldsymbol{Q} = \{\underline{Q}_i\}$ are conditional probability vectors for the *nodes*, $\{V_i\}$, and $\boldsymbol{R}^{(x)} = \{\underline{R}_i^{(x)}\}$, for $x \in \{true, false\}$, are conditional probability vectors for the *checks*, $\{Z_i\}$ (see *Fig. 3*(*b*)). In this work, we define the exact meaning of $\boldsymbol{Q}$ and $\boldsymbol{R}$ as:

$$Q_{ij} = P(V_i = true \mid \text{all information from connected arcs other than } j), \text{ and} \quad (3)$$

$$R_{ij}^{(x)} = P(Z_i = true \mid V_{d(i, j)} = x, \text{ all information from connected arcs other than } j). \quad (4)$$

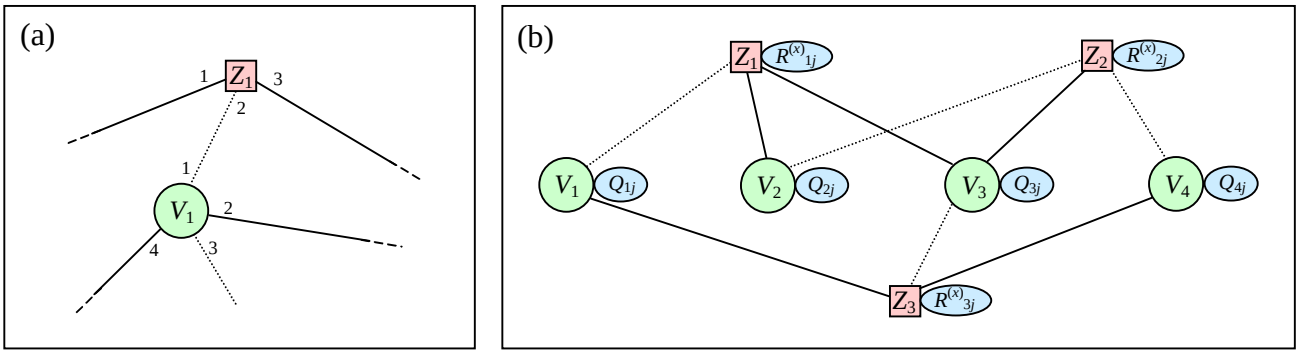

Fig. 3. (a) Example of enumerating the arcs. (b) Example of a *belief network* (ovals represent the probability vectors).

[3] A *bipartite* graph is a network with two types of vertices, and with arcs that only connect vertices of different types.
[4] We use, below, the term '*node*' to refer exclusively to members of V, and the term '*check*' to refer to members of Z.

3.2 The Sum-Product Algorithm

As the values of the Z_i depend on the V_i, and vice versa, in order to calculate the posterior probability it is necessary to update the values of $\boldsymbol{Q}$ and $\boldsymbol{R}^{(x)}$ iteratively. The sum-product algorithm achieves this as follows.

At the start of the calculation, the Q_{ij} are initialised to the *prior probability distribution*, g_i. There are then two steps to perform each iteration; the first updates $\boldsymbol{R}^{(x)}$ given $\boldsymbol{Q}$, and the second calculates $\boldsymbol{Q}$ given the new $\boldsymbol{R}^{(x)}$. The calculations are somewhat complicated by the existence of 'negated' arcs, which correspond to negating the connected proposition, but are otherwise similar in character to those described in [18].

Step 1: This follows from the definition of $\boldsymbol{R}^{(x)}$ in eq. (4): for each arc on a *check*, we want to calculate the probability that the *check* is *true*, given that the *node* on that arc has value $x \in \{true, false\}$. For example, for $x = true$, then if the arc is 'normal', this probability is 1, otherwise it is 1 – P(not satisfied by the other arcs). The probability, P_{ik}, that *check* Z_i is not satisfied by the kth attached *node*, $V_{d(i,\,k)}$, depends on the (conditional) probability of that *node* being *true*, $Q_{d(i,\,k)f(i,\,k)}$, and whether or not the arc is 'negated':

$$P_{ik} = \begin{cases} Q_{d(i,k)\,f(i,k)} & \text{'negated' arc } k \\ 1 - Q_{d(i,k)\,f(i,k)} & \text{'normal' arc } k \end{cases} \tag{5}$$

After evaluating the probabilities P_{ik}, $R_{ij}^{(x)}$ can be calculated as described above, i.e.:

$$R_{ij}^{(true)} = \begin{cases} 1 - \prod_{k \neq j} P_{ik} & \text{'negated' arc } j \\ 1 & \text{'normal' arc } j \end{cases} \qquad R_{ij}^{(false)} = \begin{cases} 1 - \prod_{k \neq j} P_{ik} & \text{'normal' arc } j \\ 1 & \text{'negated' arc } j \end{cases} \tag{6}$$

Step 2: The updating of the Q_{ij} is the step incorporating Bayes' theorem; the probability of V_i being *true* given that the *checks* are all *true* is proportional to the probabilities of the *checks* being *true* given V_i *true*, which are the probabilities $R_{c(i,k)e(i,k)}^{(true)}$. We thus calculate the product of those probabilities (excluding arc j), multiply by the prior probability of being *true*, g_i, and normalise:

$$Q_{ij} = \frac{g_i \prod_{k \neq j} R_{c(i,k)e(i,k)}^{(true)}}{(1 - g_i) \prod_{k \neq j} R_{c(i,k)e(i,k)}^{(false)} + g_i \prod_{k \neq j} R_{c(i,k)e(i,k)}^{(true)}} \tag{7}$$

The final, and most important, part of the algorithm is the method for calculating the posterior probability of each proposition being *true* given that each clause must also be *true*. When applied to a graph with no cycles (a *polytree*), and once converged, the following calculation yields the exact posterior probabilities:

$$p_i = P(V_i = true \mid Z_j = true \ \forall j) = \frac{g_i \prod_{k} R_{c(i,k)e(i,k)}^{(true)}}{(1 - g_i) \prod_{k} R_{c(i,k)e(i,k)}^{(false)} + g_i \prod_{k} R_{c(i,k)e(i,k)}^{(true)}} \tag{8}$$

Note that this time the product is taken over all k; this can be done with little extra effort when calculating the Q_{ij} as in eq. (7).

When applied before convergence of the algorithm, or on graphs with cycles, eq. (8) yields the 'pseudoposterior probability', which can be used to detect convergence. Although not the exact posterior probability for graphs with cycles, it has been found by MacKay in [18] to work well for determining an assignment (in which the exact probabilities do not feature).

4. IMPLEMENTATION AND RESULTS

Initially, the aim was to code the sum-product algorithm for efficiency, and thus streamlined C code was written which performed the basic algorithm. During preliminary testing of this program, however, it became apparent that there were many parameters and configurations to explore and so the important factor became the adaptability of the program. To this end, it was rewritten in C++, using classes to separate distinct areas of the program and make changes easier and faster to implement.

4.1 Algorithm Testing

In order to compare the performance of various adaptations to the sum-product algorithm, the following prescription for generating and testing SAT formulae was used:

- It was decided that the programs should be tested on random CNF formulae which had at least one solution. In order to generate these, a solution, $\underline{v}_{secret}$, was first randomly generated in which each proposition was *true* or *false* with even probability. Next, clauses were generated by selecting three propositions at random and negating them with probability 0.5 each. Finally, the clauses were then checked to see if they were satisfied by $\underline{v}_{secret}$, and if not they were discarded. Once the clauses had been created, $\underline{v}_{secret}$ was discarded too.
- The iterative step was performed up to 100 or $\sqrt{M}$ times, whichever was larger.
- Assignments were generated from the pseudoposterior probabilities, p_i, after each iteration by:

$$V_i = \begin{cases} true & p_i \geq 0.5 \\ false & p_i < 0.5 \end{cases} \tag{9}$$

- The assignment was tested to see if it satisfied the formula. If it did, then the calculation was stopped, and *success* was reported.
- A comparison between the $\{p_i\}$ this iteration and those of the previous iteration was made. If no change (to two decimal places) was detected for 4 iterations, the calculation was stopped and *failure* reported, as it was assumed that the algorithm had reached a stable state which was not a solution to the problem.
- If the iterations were completed and yet no solution found, *failure* was reported.
- To explore the parameter space, results were gathered for a wide range of M (spread exponentially between 2 and 1000000), and for N = 10, 20, 50, 100, and 1000. For each N and M, 100 results were obtained, so as to give an outline of the statistical properties.

4.2 Presentation of Results

As the algorithm design and testing processes for this investigation were closely linked, the method and results are presented together for each variation of the basic algorithm. Six variants are described, and to help minimise confusion, they will each be given an identifier of the form: `SAT<name>`, where `<name>` is a label related to the method used.

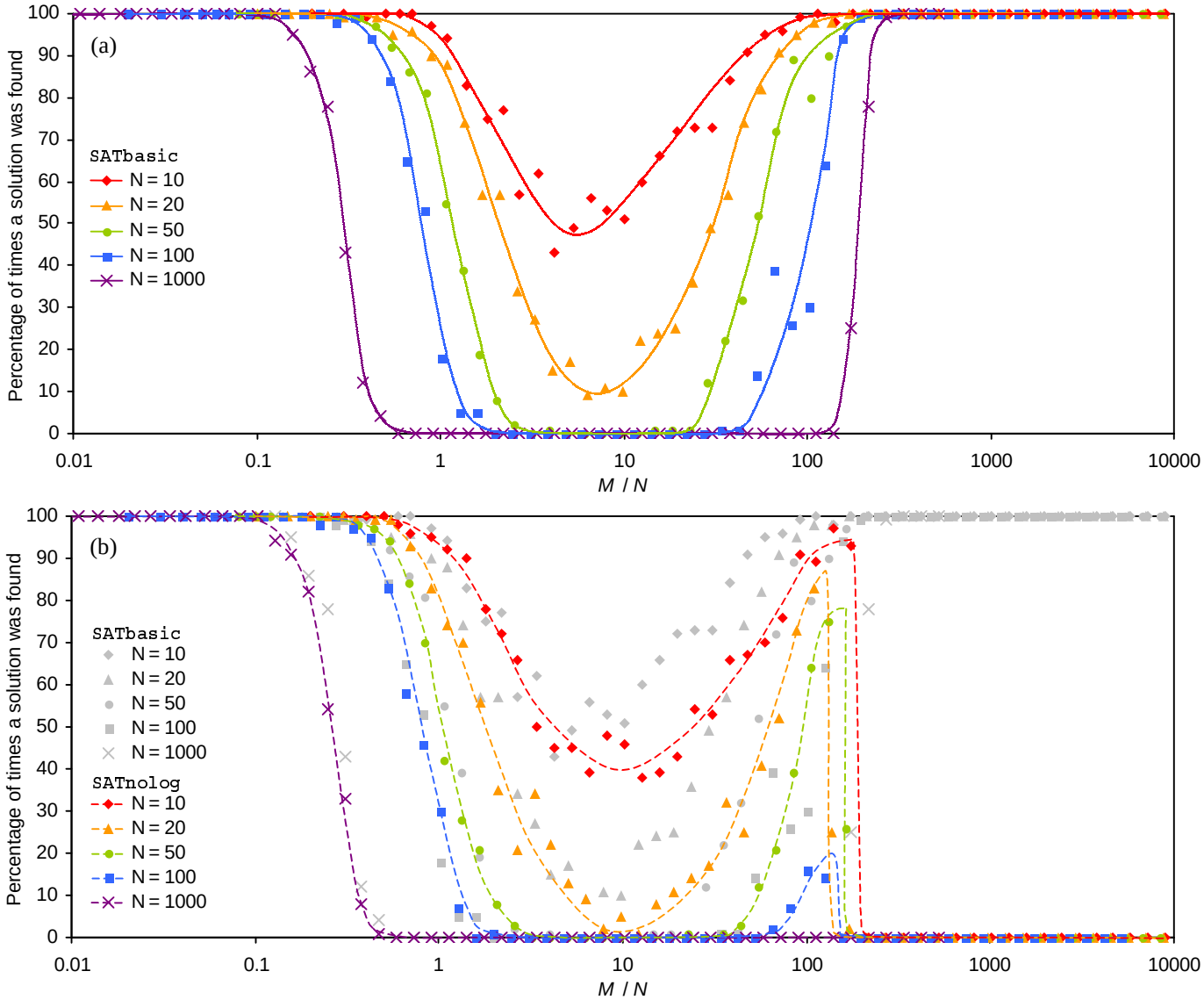

Fig. 4. (a) Graph of percentage of times a solution was found against M / N for `SATbasic`. (b) Graph of percentage of times a solution was found against M / N for `SATnolog`, with the points for `SATbasic` plotted as a comparison. (*NB. The lines are just to guide the eyes and help distinguish trends*.)

4.3 Basic Sum-Product Algorithm

Initial tests of the basic algorithm disclosed several difficulties linked to floating point errors[5], including a critical problem occurring when the number of connections to each *node* was too large: the products of $R_{ij}^{(true)}$ and $R_{ij}^{(false)}$ could both become zero, hence making eq. (7) undefined. This problem was overcome in two ways: first, attempts were made to stop the Q_{ij} becoming too close to 0 or 1 by limiting them to between 10^{-8} and $1 - 10^{-8}$, and second, instead of using probabilities, the algorithm was adapted to use the logarithm of the probabilities. This allowed the products of $R_{ij}^{(true)}$ and $R_{ij}^{(false)}$ to both be very small, but still retain accuracy when normalised. Thus, this method gives the best possible accuracy for the probabilities, and will be referred to as `SATbasic`, whilst the version not using logarithms will be called `SATnolog`. The results for `SATbasic` are shown in *Fig. 4*(*a*), and a comparison of the two methods shown in *Fig. 4*(*b*). It was observed that although using logarithms yielded a slower algorithm, it allowed a greater range of M / N to be explored and was 'truer' to the original sum-product algorithm. It was therefore decided to use `SATbasic` as the base for further investigations.

[5] A *floating-point error* is a by-product of the limited precision with which real numbers are stored on a computer.

4.4 Improving the Algorithm

The characterisation of `SATbasic`, *Fig. 4*(*a*), revealed a failure to find the solution around the 'hard' regions of the problem space ($M / N \approx 4$), and a number of approaches were considered in order to improve its performance in this area. Investigating in detail the behaviour of `SATbasic` on 'hard' problems, it was noticed that the sum-product algorithm exhibits two main types of behaviour:

- **Unstable:** the pseudoposterior probabilities change greatly from one iteration to the next. In some cases, they end up taking only values very close to 0 or 1, at which point the pattern might sometimes repeat a cycle of moves or otherwise evolve in a seemingly chaotic fashion.
- **Weak decisions:** the pseudoposterior probabilities are close to 0.5, and the system is effectively 'unsure' as to what assignment to make.

To attempt to deal with these behaviours, two different algorithms were developed. The first addresses the problem of weak decisions by choosing a few propositions which have pseudoposterior probabilities furthest from 0.5, and 'fixing' them to the closer of 0 or 1 by setting their prior probabilities. The iterations are then allowed to continue, and the hope is that by making a decision, we have resolved whatever conflicts were causing the sum-product algorithm to be uncertain. Care has to be taken in making the choice of which ones to fix, however, since the system may be in the 'unstable' regime, and probabilities furthest from 0.5 may be in a state of flux. Therefore, this 'fixing' algorithm, which we shall refer to as `SATf`, ranks the propositions by the following (arbitrary) measure:

$$(5 - s_i) + (0.5 - f_i/200) + |0.5 - p_i|, \tag{10}$$

where $s_i \in \{0, \ldots, 5\}$is a measure of the stability of the proposition, 0 being stable, and 5 being unstable, and f_i is the number of flips of assignment (out of 100) this proposition has undergone. The 3% of propositions with the lowest value of this measure are then 'fixed'. `SATf` then continues this running and fixing until two thirds of the propositions have been fixed before returning *failure*.

The second attempt to improve the behaviour of the sum-product algorithm involved changing the prior probability for all propositions, g, from 0.5, to see if the weak decisions and instability were reduced. This adaptation, referred to as `SATg`, attempts to solve each SAT formula with a range of g between 0.2 and 0.8. A comparison of the performance of `SATg`, `SATf`, and `SATbasic` is shown in *Fig. 5*, below. It is clear from these results that both `SATf` and `SATg` performed better than `SATbasic` in most cases, and that of those two, `SATg` was generally the superior. In order to shed some light on why this might be, we next investigated the prior probabilities in more detail.

4.5 Prior Probabilities

The choice of prior probabilities, g_i, was discussed in section 2.1, and it was decided that for random CNF clauses it was best, mathematically, to choose the non-informative prior; $g_i = g = 0.5$. Now consider what we know if we are creating the clauses according to the prescription in section 4.1. If a particular proposition, X, in $\underline{v}_{secret}$ were *true*, then if a clause is created with the literal X in, it would always be accepted. A clause with $\neg X$ in, however, would only be accepted if one or both of the other two literals is *true*, which occurs with probability 0.75. Hence, for M / N large (many clauses concerning each proposition), one might expect more clauses to contain X than $\neg X$. The converse is, of course, true if X is *false* in $\underline{v}_{secret}$.

To investigate this problem, `SATbasic` was adapted to use this information to 'cheat' by counting the numbers of V_i and $\neg V_i$ in the logic, and setting its prior probabilities as follows:

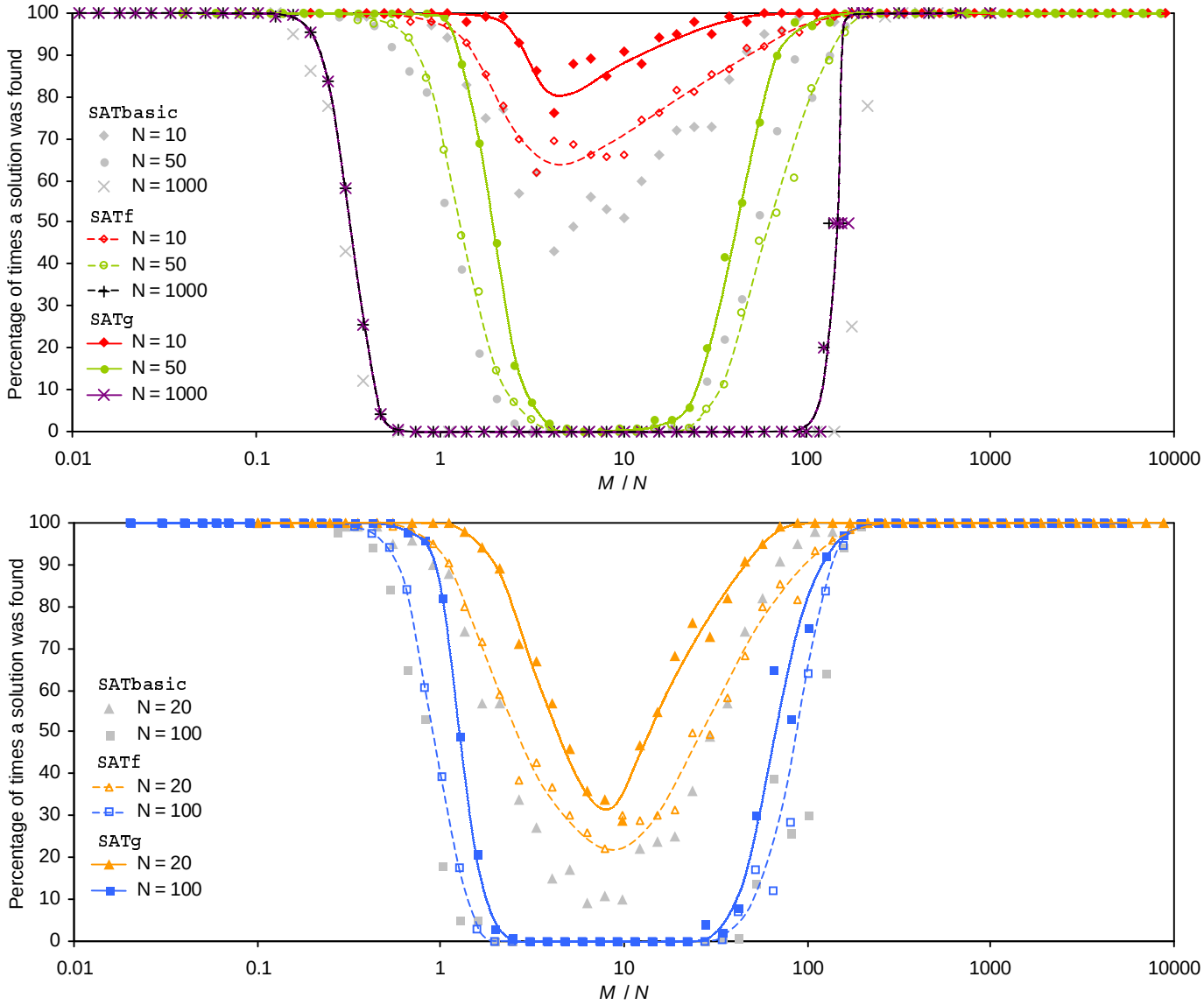

Fig. 5. Graphs of percentage of times a solution was found against M / N for `SATbasic`, `SATf`, and `SATg`. For clarity, the results have been plotted on two separate graphs, and it should be clear that `SATg` generally outperforms the other two algorithms. (*NB. The lines are just to guide the eyes and help distinguish trends.*)

$$g_i = \begin{cases} 0.8 & (\#\text{ of } V_i) > (\#\text{ of } (\neg V_i) + 1) \\ 0.2 & (\#\text{ of } V_i) < (\#\text{ of } (\neg V_i) - 1) \\ 0.5 & \textit{otherwise} \end{cases} \tag{11}$$

The results of this variant, `SATcheat`, are shown in *Fig. 6*(*a*) below, and are somewhat surprising; there seems to be very little difference between the results of `SATcheat` and `SATbasic`. Investigating further, `SATbasic` was used to analyse the effect of varying g for two different N and M (see *Fig. 6*(*b*)). The resulting distribution of finding a solution against g is clearly uniform over most of the range, which is perhaps strange, as physically a low or high g corresponds to expecting a lower or higher number of *true*-valued propositions in the satisfying assignment, but the algorithm seems to not be sensitive to changes in this variable. This goes some way towards explaining the success of `SATg`. By varying g, we effectively run the sum-product algorithm a number of times (with a slightly different start condition each time) and thus it performs better than `SATbasic`, which only runs the algorithm once. It would appear that the more times it is run, with varying start conditions, the more likely it is to find a solution. This is demonstrated in *Fig. 6*(*c*), which shows

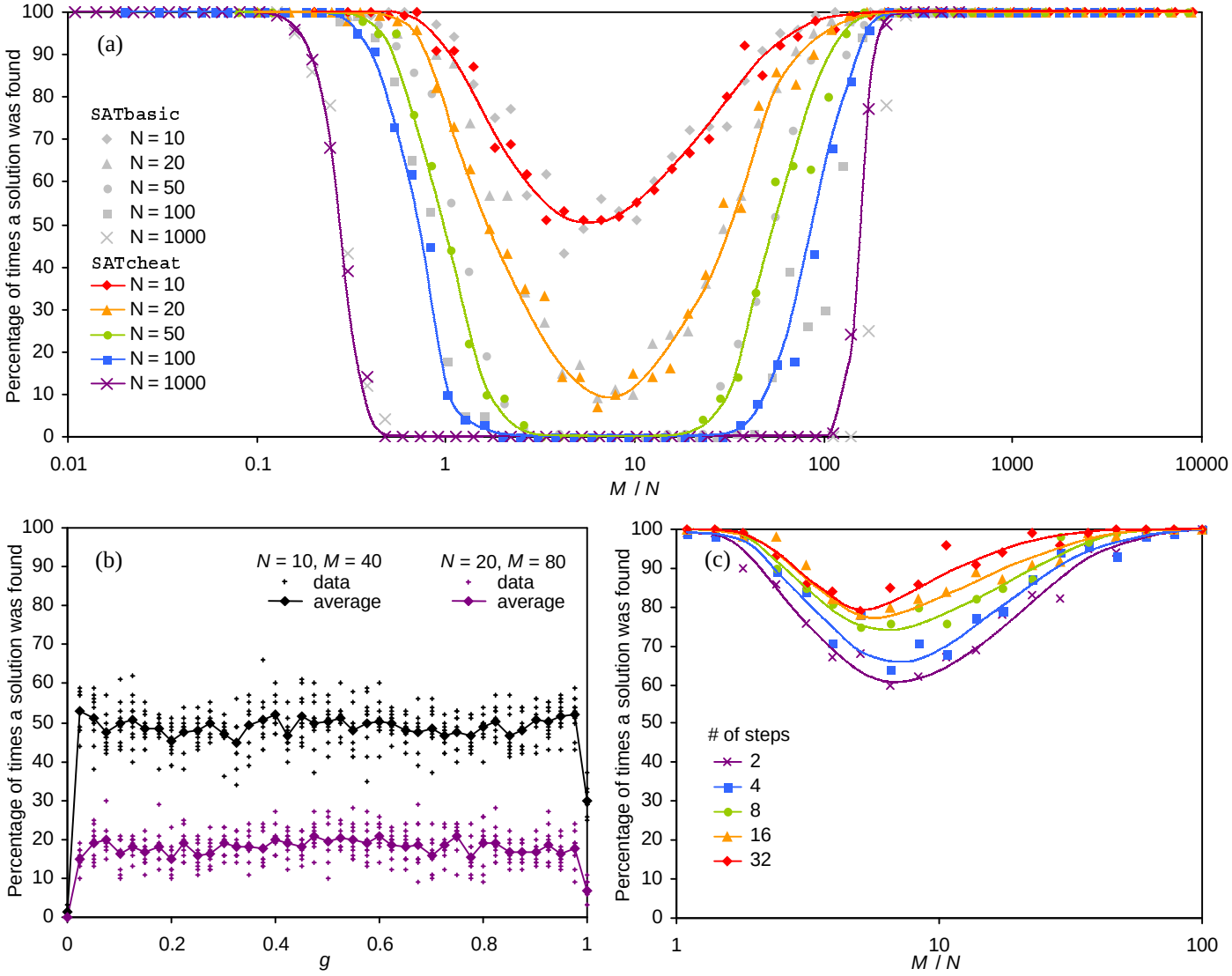

Fig. 6. (a) Graph of percentage of times a solution was found against M / N for `SATcheat`. (b) Graph of percentage of times a solution was found against g for `SATbasic` for two values of N and M (10 sets of data each). (c) Graph of percentage of times a solution was found against M / N for `SATg`, for N = 10, and varying the number of steps in g used. (*NB. The lines are just to guide the eyes and help distinguish trends.*)

the results for `SATg` of varying the number of steps in g used to go between 0.2 and 0.8. We observe that as the number of steps increases, the percentage of times a solution was found increases.

4.6 Benchmarks and Comparisons

We can see from *Fig. 5* that, of the algorithms discussed so far, `SATg` performs best in most cases. It was therefore chosen as the algorithm to test against standard SAT benchmarks. In order to put the results in perspective, the test sets were also attempted by the state-of-the-art incomplete solver, `walksat`[21]. The benchmarks used were the following selection of test sets, available from internet satisfiability sites:

- **DIMACS**: two sets of DIMACS[8] problems were tested: **AIM**, artificially generated random 3-SAT instances (only the satisfiable set), and **LRAN**, large random 3-SAT instances.
- **uf250**: 100 uniform CNF 3-SAT instances with N = 250, M = 1065, from SATLIB[20].
- **n50cnf**: 50 uniform CNF 3-SAT instances with N = 50, M = 215, from [11].

The results are slightly disappointing; `SATg` solved none of the **AIM**, **LRAN**, or **uf250** sets, and only 10% of the **n50cnf** set. In contrast, `walksat` solved 54% of **AIM**, 2 out of 3 in **LRAN**, and 99% and 98% of **uf250** and **n50cnf** respectively. Note, however, that all these sets are in the 'hard' region, $M / N \approx 4.3$, and that they were created by generating random 3-SAT and selecting the instances found hardest by some complete solvers.

Despite the poor performance of `SATg` on the hardest problems, we have shown above that the message-passing approach works very effectively on the high and low regions of M / N, and it is thought that it should be possible to create enhanced adaptations of the algorithm with improved performance in the hard region.

4.7 Backtracking

To determine if this was indeed the case, the final part of this investigation involved creating hybrid schemes, i.e. using the sum-product solver as a heuristic in some standard complete algorithms. This is a large field, however, and it was only possible to focus on one technique in the time available: the DPLL algorithm described in section 2.2. The sum-product algorithm was used at each step to determine which of the propositions was the 'best' to fix, and which value to fix it to. In order to compare the resulting program, `SATb`, on a level footing with the previous methods, however, it was necessary to first 'remove' the completeness of the algorithm (and so reduce it to polynomial time) by limiting the number of backtracks to 50. This allowed the algorithm to make only 50 incorrect decisions before returning *failure*, and hence the resulting performance is a good indication of the effectiveness of the sum-product algorithm as a heuristic. The results are summarised in *Fig. 7* below, and it is clear that `SATb` outperforms the other sum-product-based algorithms tested above. We suggest that the performance of `SATb` might be improved by using the sum-product algorithm to assign values to more than one proposition at a time. A further enhancement would be to make use of the information gained each time an inconsistency arises, perhaps by introducing some extra *checks* into the belief network indicating that a successful assignment is likely to be different to the failed assignment.

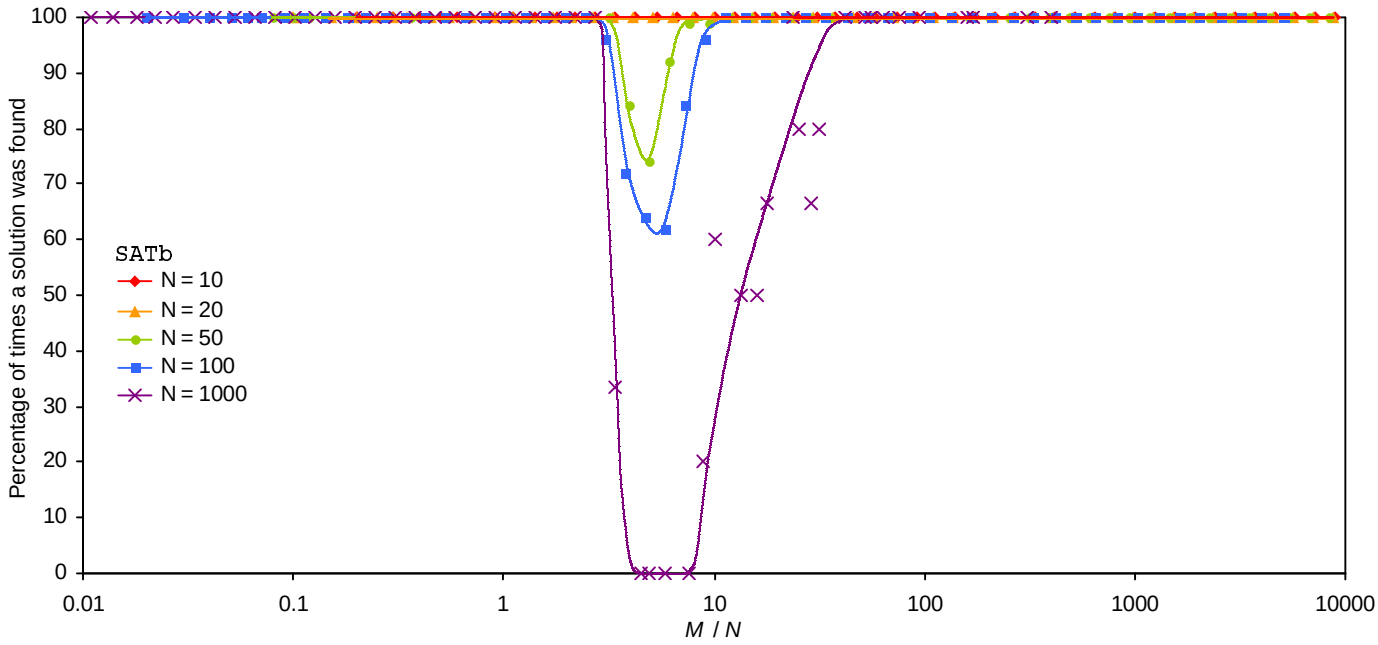

Fig. 7. Graph of percentage of times a solution was found against M / N for `SATb`, to the same scale as *Figs. 4* & *5*. For $N = 1000$ and $M / N > 4$, it was necessary to reduce the number of runs per data point from 100 to only a few to speed up data collection, and hence those data points have a greater scatter. The range of M / N for which this algorithm is useful, however, is clearly much larger than previous algorithms. (*NB. The lines are just to guide the eyes and help distinguish trends.*)

5. DISCUSSION

Graph-based techniques for solving logic problems, though well established and of significant use, are often overlooked by researchers. The original Davis-Putnam algorithm[6] for solving SAT problems was in fact graph-based, but was then developed into a purely logic-based backtracking algorithm (DPLL). Recent work on developing the original graph-based methods has met with some success, and afforded some insight into the behaviour of SAT problems, though the research is still at an early stage[19].

We saw above that the graph-based algorithm in this investigation, though good for under- or over-constrained instances, performs poorly in regions of the parameter space where it is conventionally hard to find solutions; *Fig. 4*(*a*) shows `SATbasic` failing to find a solution for a range of M / N, centred on $M / N \approx 5$, which grows as N is increased. Some of the adaptations to the basic algorithm, specifically `SATg` and `SATb`, perform much better in these regions, but it would appear that these methods still fail to solve the hardest SAT problems. Despite these problems, we believe that such Bayesian techniques can be very valuable in that, if the pseudoposterior probabilities converge to a stable solution, they convey information about how every proposition is affected by every other proposition. Such a holistic approach to the problem has been lacking from the field for many years, and could provide a powerful tool for analysing SAT problems effectively.

More research is undoubtedly needed into methods to help the algorithm converge on a stable solution though, and some possibilities include:

- **Post-processing:** A brief study of 'post-processing' the output of the sum-product algorithm was undertaken for this discussion, in which the pseudoposterior probabilities were used as a heuristic in a conventional backtracking search. It was found to work well for small N, but was sometimes very slow for larger N, indicating that the heuristic was not particularly effective. As a further investigation, it would be interesting to consider the application of stochastic local search methods to the assignment generated by `SATbasic`. This would show whether the algorithm is getting 'close' in problem space to a satisfying assignment.
- **Problem structure:** A fundamental difficulty with the algorithm is that it was designed for polytrees, whereas a uniform SAT problem at the phase transition produces a dense graph with many cycles. This density of connections is likely to be the main cause of failure of the technique, and it may be worth considering if any other message-passing algorithms are more suited to the types of graphs generated by such problems. Another alternative would be to investigate the different structures of networks generated from *non-uniform* SAT problems. In particular, some real-world SAT problems have graphs with fewer cycles and the sum-product algorithm may work particularly well on problems of these forms.
- **Update schedule:** It is possible that the algorithm suffers from problems due to all the messages being passed synchronously. A brief investigation of asynchronous message-passing was carried out and no significant difference was observed, but the results were inconclusive.
- **Backtracking with inference:** As mentioned in section 4.7, it seems likely that the performance of `SATb` or similar would be greatly improved by inferring information when inconsistencies or failures arise.

Even if it is found to be ineffective in finding satisfying assignments, the message-passing technique studied here is potentially still useful in that it gives a new perspective on the problem, and it should be possible to analyse why the algorithm fails and thus gain insight into what makes a 'hard' problem hard.

The above investigation was very time-limited, and it was not possible to perform more than a small fraction of the possible experiments on the algorithm. We have discussed some possible

directions for future research, but it is worth also considering some extensions to the work carried out. In particular:

- **Unbiased CNF:** It would be useful to gather a set of results using an unbiased logic-generation algorithm in which a satisfied clause is discarded a quarter of the time, though it is expected that this would not have a very great effect on the forms of the graphs.
- **Number of iterations:** A series of experiments determining the numbers of iterations required for convergence would be worthwhile. The choice of 100 iterations was almost arbitrary in this study, and it may be that this was too many or too few and that by changing the number of iterations it could be possible to improve the algorithm's performance.
- **Tweaking parameters:** `SATf` and `SATb` have tuneable parameters in the measure used to rank the *nodes*, eq. (10). Experiments to adjust this equation could result in improved performance for both of these algorithms.

6. CONCLUSIONS

In this investigation, we have successfully explored the behaviour of not only the basic sum-product algorithm but also several adaptations, and demonstrated that they solve the SAT problem well for a wide range of M / N. We have shown that such algorithms on their own are not sufficient to tackle hard SAT instances ($M / N \approx 4$), but, in the process, this study has revealed several possible areas for future research which may lead to either significant improvements in performance or an insight into the nature of hard SAT problems.

REFERENCES

[1] D. Achlioptas, C. Gomes, H. Kautz, and B. Selman, "Generating satisfiable problem instances," *Proc. AAAI-2000*, 2000.

[2] G. Birola, R. Monasson, and M. Weight, "A variational description of the ground state structure in random satisfiability problems," *Eur. Phys. J. B.*, vol. 14, pp. 551-568, 2000.

[3] P. Cheeseman, B. Kanefsky, and W.M. Taylor, "Where the *really* hard problems are," *Proc. IJCAI-91*, vol. 1, pp. 331-337, 1991.

[4] S.A. Cook, "The complexity of theorem-proving procedures", *Conference Record of the Third Annual ACM Symposium on the Theory of Computing*, pp. 151-158, 1971.

[5] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Comm. ACM*, vol. 5, no. 7, pp. 394-397, 1962.

[6] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201-215, 1960.

[7] S. Devadas, "Optimal layout via Boolean satisfiability," *Proc. Int. Conf. on Computer-Aided Design* (*ICCAD*), pp. 294-297, Nov. 1989.

[8] DIMACS challenge: http://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/

[9] R.G. Gallager, "Low density parity check codes," no. 21 in Research Monograph Series. Cambridge, MA: MIT Press, 1963.

[10] I.P. Gent and T. Walsh, "The SAT phase transition," Research paper 679, Department of AI, University of Edinburgh, 1994.

[11] J. Gottlieb: http://www.in.tu-clausthal.de/~gottlieb/benchmarks/3sat/

[12] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah, "Algorithms for the satisfiability (SAT) problem: a survey." Preliminary version, 1996.

[13] B. Hayes, “Can’t get no satisfaction”, *Am. Sci.*, vol. 85, no. 2, pp. 108-112, Mar. 1997.
[14] H.H. Hoos and T. Stützle, “Towards a characterisation of the behaviour of stochastic local search algorithms for SAT,” *AI*, vol. 112, pp. 213-232, 1999.
[15] D. Johnson, and M. Trick, eds. “DIMACS series in discrete maths and theoretical computer science,” *AMS*, vol. 26, 1996. See also, [8].
[16] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, eds. “The travelling salesman problem.” John Wiley & Sons, New York, 1985.
[17] P. Liberatore, “On the complexity of choosing the branching literal in DPLL,” *AI*, vol. 116, pp. 315-326, 2000.
[18] D.J.C. MacKay, “Good error-correcting codes based on very sparse matrices,” *IEEE Trans. Inform. Theory*, vol. 45, no. 2, pp. 399-431, Mar. 1999.
[19] I. Rish, “Efficient reasoning in graphical models,” Ph.D. thesis, University of California, Irvine, 1999.
[20] SATLIB: http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/
[21] B. Selman, H. Kautz, and B. Cohen, “Local search strategies for satisfiability testing,” *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation challenge*, Oct. 1993, published as [15].

APPENDIX

Here we formalise some aspects of the discussion in section 3 of the main text. As described in §3.1, we consider the undirected bipartite graph $G(\{V, Z\}, \{E_a, E_b\})$ with vertices $\{V, Z\}$ and edges $\{E_a, E_b\}$, where $V = \{V_1, \ldots, V_N\}$ is a set of N Boolean variables, $Z = \{Z_1, \ldots, Z_M\}$ is a set of M Boolean variables, and $E_a, E_b \subseteq V \times Z$ are two distinct sets of edges joining the V and Z vertices. The set V represents the N propositions, and the set Z represents the Boolean value of the M clauses. The sets of edges are related to the logic formula for the specific SAT instance, in that:

$$E_a = \{(V_i, Z_j) \mid V_i \in C_j\}, \tag{12}$$
$$\text{and } E_b = \{(V_i, Z_j) \mid (\neg V_i) \in C_j\}. \tag{13}$$

If an arc $(V_i, Z_j) \in (E_a \cup E_b)$, we say that V_i and Z_j are *connected*. We can now define:

$$a(i) = \{ j \mid (V_j, Z_i) \in E_a\} \tag{14}$$
$$\text{and } b(i) = \{ j \mid (V_j, Z_i) \in E_b\} \tag{15}$$

as the sets of indices of all V_j connected to vertex Z_i by arcs in E_a and E_b respectively. These can be used to express mathematically how the values of Z_i are calculated from an assignment to V:

$$Z_i = \bigvee_{k \in a(i)} (V_k) \quad \vee \quad \bigvee_{k \in b(i)} (\neg V_k), \tag{16}$$

where the 'big-$\vee$' notation takes the logical OR over its arguments. Furthermore, we can now precisely define the conditions for 'negated' and 'normal' arcs in eqs. (5) and (6) as:

$$\text{'negated' arc } k \;\Leftrightarrow\; d(i, k) \in b(i) \tag{17}$$
$$\text{'normal' arc } k \;\Leftrightarrow\; d(i, k) \in a(i) \tag{18}$$

Finally, it is worth briefly mentioning an efficient way of performing the products in eq. (7); the *forward-backward* algorithm. Let $\underline{F}^{(x)}$ and $\underline{B}^{(x)}$ be four vectors of length L, where $x \in \{true, false\}$ and L is the number of arcs on *node* V_i. The elements of the vectors are given by:

$$F_j^{(x)} = \prod_{k=1}^{j} R^{(x)}_{c(i,k)e(i,k)} \tag{19}$$

$$B_j^{(x)} = \prod_{k=L-(j-1)}^{L} R^{(x)}_{c(i,k)e(i,k)} \tag{20}$$

We need only to calculate these elements once, and then we can efficiently calculate:

$$\prod_{k \neq j} R^{(x)}_{c(i,k)e(i,k)} = \begin{cases} B^{(x)}_{L-1} & j = 1 \\ F^{(x)}_{L-1} & j = L \\ F^{(x)}_{j-1} B^{(x)}_{L-j} & \textit{otherwise} \end{cases} \tag{21}$$