Technical Reports on Mathematical and Computing Sciences: TR-C173
title: Analysis of a Randomized Local Search Algorithm for LDPCC Decoding Problem
author: Osamu Watanabe and Takeshi Sawai
affiliation:
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
Meguro-ku Ookayama, Tokyo 152-8552, Japan
(`watanabe@is.titech.ac.jp`)
acknowledgments to financial supports:
Supported in part by a Grant-in-Aid for Scientific Research on Priority Areas "Statical-Mechanical Approach to Probabilistic Information Processing" 2002 and Scientific Research (C) 2001-2002 from the Ministry of Education of Japan.

Abstract: We propose an approach for analyzing the average peformance of a given (randomized) local search algorithm for a constraint satisfaction problem. Our approach consists of two approximations. Using a randomized algorithm for LDPCC decoding, we experimentally investigate the reliability of these approximations and show that they (in particular, the second one) could be used as a tool for analyzing the average performance of randomized local search algorithms.

1 Introduction

There are many problems that can be formulated as a "constraint satisfaction problem", a problem of searching for a solution that satisfies a given set of constraints.

A typical example is, of course, the following (search version of) 3SAT Problem.

<u>3SAT Problem</u> — 3-CNF SATisfaction Problem —

Input: A Boolean $3-CNF$ formula F on n Boolean variables. That is, $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause C_k is a disjunction of three literals.

Output: A *sat. assingment*, i.e., an assignment to Boolean variables that satisfies all clauses and makes $F = 1$.

A "local search" is simple yet one of the important algorithmic approaches for solving such constraint satisfaction problems. For 3SAT Problem, local search algorithms have been studied intensively since the work of, e.g., Selman, Levesque, and Mitchell [SLM92] and Gu [Gu92]. Figure 1 shows a schematic description of local search algorithms for 3SAT Problem. A concrete algorithm is obtained[1] by determining ways to choose an initial assignment at (1) and to choose one unsatisfiable clause C_k (if there are more than one) and one variable $\mathtt{X}_i$ in C_k at (2). Here we consider "randomized" local search algorithms, algorithms using randomness in such choices. For example, we can cosider the following variation of `LocalSAT` (which we will call `RandSAT`): at (1), for an intitial assignment, choose one of all 0,1-assignments uniformly at random, and at (2), choose one unsat. clause C_k and one variable $\mathtt{X}_i$ in C_k uniformly at random.

[1]We also have to fix the parameters S and t appropriately, which will be discussed later.

```
program LocalSAT(F = C_1 ∧ C_2 ∧ ··· ∧ C_m);
  repeat S times
  [ X_1, ..., X_n ← an initial assignment a = (a_1, ..., a_n);    — (1)
  [ repeat t steps
  [ [ if F = 1 then output the current assignment and halt;
  [ [ choose one unsat. clause C_k of F,
  [ [     and one variable X_i appearing in C_k;                 — (2)
  [ [ flip the value of X_i;
program end.
```

Figure 1: Local search algorithm for 3SAT

Though very simple, many researchers have shown (see, e.g., [HS00]) that this randomized local search algorithm works "quite well". More precisely, when parameters S and t are chosen appropriately, RandSAT succeeds to find a satisfying assingment (if any) *on average* under a certain probabilistic situation. Unfortunately, though, most of such positive results have been obtained through computer experiments, and existing rigorous mathematical analyses are far from verifying these computer experiments. For example, the following facts have been proved formally; but they are far from justifying the performance of RandSAT given by computer experiments.

Theorem 1. [Sch99] Set $S = (4/3)^n \approx 2^{1.33n}$ and $t = 3n$. Then RandSAT gives a sat. assignment with high probability for *all* satisfiable formula.

Theorem 2. [2] [KP92] Set $S = 1$ and $t = n$. There is some constant $d > 0$ for which the following holds: When a satisfiable formula with $m \geq dn^2$ clauses is given uniformly at random, RandSAT gives a satisfying assignment with high probability.

It may be difficult to give a rigorous mathematical analysis to the *average* performance of a randomized local search algorithm. But still there may be some "semi formal" analysis that may not be rigorous but that is still reasonable for investigating the average behavior of the algorithm. Furthermore, through such a semi formal analysis, we may be able to understand when and how the algorithm works, which also leads us to design better algorithms.

In this paper we propose such a "semi formal" approach. For a given randomized local search algorithm, our approach consists of the following two approximations.

I. Approximate an average and randomized execution of the algorithm by a simple random process.

II. Approximate this process by simple probabilistic recurrence formulas.

So far, no mathematical justification has been given to any of these two approximations. But we conjecture that the second approximation holds to a certain extent. On the other hand,

[2] Gent [Gen98] pointed out that almost trivial randomized algorithm finds a sat. assignment with high probability when a satisfiable formula with $\Omega(n \log n)$ clauses is given uniformly at random. Thus, we have a better average-case upper bound. But the theorem is still of interest as a mathematical analysis of a local search algorithm.

it seems difficult to justify the first approximation formally. In fact, this approximation may not work for some situations; we think that by investigating when and why this approximation fails to work, we may be able to understand when and why the analyzed algorithm fails to work with high probability.

For our technical discussion we will consider an example of constraint satisfaction problems — LDPCC Decoding Problem — and a randomized local search algorithm for this problem. Following our proposed approach, we will analyze the average performance of this algorithm.

2 An Example Problem: LDPCC$(3,4)$ Decoding Problem

We consider the following problem, which we will call $(3,4)$-*Low Density Parity Check Code Decoding Problem* (in short, LDPCC$(3,4)$ Decoding Problem).

LDPCC$(3,4)$ Decode — Low Density Parity Check Code Decoding —

Input: A set of the following type equations on Boolean variables $x_1, ..., x_n$.

$$\left.\begin{array}{lcl} x_{i(1,1)} + x_{i(1,2)} + x_{i(1,3)} + x_{i(1,4)} & = & c_1 \\ x_{i(2,1)} + x_{i(2,2)} + x_{i(2,3)} + x_{i(2,4)} & = & c_2 \\ \vdots & & \\ x_{i(m,1)} + x_{i(m,2)} + x_{i(m,3)} + x_{i(m,4)} & = & c_m \end{array}\right\} \quad (3)$$

Where
- $c_1 \sim c_m$ are 0 or 1 constants,
- $+$ is mod 2 addition (i.e., the ex-or operation),
- $i(j,k) \in \{1, ..., n\}$ is an index, where $i(j,1) \sim i(j,k)$ are all distinct for each j, $1 \le j \le m$, and
- each x_i appears exactly three times (which implies $m = 4n/3$).

Output: An assignment to $x_1, ..., x_n$ satisfying (3) with the smallest number of 1's.

Gallager [Gal62] introduced the notion of LDPCC and proposed a decoding algorithm, which is now called a BP decoding algorithm. (BP = (*Loopy*) *Belief Propagation* was proposed and has been studied in the AI community, as a basic algorithm for analyzing a given Basian network.) Recently several researchers (see, e.g., [Mac99]) rediscovered this code and the BP decoding algorithm. Furthermore recent detail analyses of LDPCC family (see, e.g., [Mac99, MKSV00, RSU01]) showed that some member of LDPCC family seems to have a good performance, as close as the Shannon limit. A randomized local search algorithm that we will study is a variation of randomized decoders that have been studied as a simplified model of the BP decoding algorithm[3].

2.1 Basics on Linear Codes and Our Average-Case Senario

Besides the requirements on the form of equations of (3), the above defined problem is regarded as a general decoding problem of a linear code. Thus, let us first recall briefly some basic concepts on linear codes.

[3] Notice, however, that the BP decoding algorithm is not a randomized algorithm; it is a deterministic algorithm calculating the probability — likelihood – of $\mathtt{x}_i = 1$ for each i under the condition that a given syndrome is observed.

We will call each equation of (3) a *parity check equation*, or a *check equation* in short. Let $\boldsymbol{x} = (x_1, ..., x_n)$ and $\boldsymbol{c} = (c_1, ..., c_m)$. Then a system of check equations like (3) is expressed as $H\boldsymbol{x} = \boldsymbol{c}$ by using an $m \times n$ 0,1-matrix H, which is called a *parity check matrix*.

For a parity check matrix H, a vector $\boldsymbol{a} = (a_1, ..., a_n)$ satisfying $H\boldsymbol{a} = \boldsymbol{0}$ is called a *code word* (w.r.t. H), which is supposed to be sent for communication. We consider a *binary symmetric channel* for our noise model; when a code word is transmitted, some of its bits are flipped independently at random with *noise probability* p. That is, when sending a cord word $\boldsymbol{a}$, a message $\boldsymbol{b}$ received through the channel is computed as $\boldsymbol{b} = \boldsymbol{a} + \boldsymbol{v}$, where $\boldsymbol{v}$ is a 0,1-vector whose each bit takes 1 with probability p. We call $\boldsymbol{v}$ a *noise vector*. (Here by + we mean the bit wise addition under modulo 2.)

Let $\boldsymbol{c} = (c_1, ..., c_m)$ be a vector computed from the received message $\boldsymbol{b}$ as $\boldsymbol{c} = H\boldsymbol{b}$, which is called a *syndrome*. Then from the property of the code word $\boldsymbol{a}$, and the linearity of H, it follows

$$\boldsymbol{c} \;=\; H\boldsymbol{b} \;=\; H(\boldsymbol{a} + \boldsymbol{v}) \;=\; H\boldsymbol{a} + H\boldsymbol{v} \;=\; H\boldsymbol{v}.$$

Thus, for a given $\boldsymbol{c}$ (and also a given H), solving (3) is to obtain a noise vector $\boldsymbol{v}$, from the syndrome $\boldsymbol{c}$ computed from the received message $\boldsymbol{b}$. Usually H is not regular and there are more than one solutions. On the other hand, since p is small, say $p < 0.2$, it is likely that a solution with the smallest number of 1's is the actual noise vector. This is why the problem asks for a solution with the smallest number of 1's. It should be noted, however, that from the average-case scenario we will assume (see below), the problem should not be regarded as an NP type optimization problem; rather, it should be regarded as an NP type search problem like 3SAT Problem.

A parity check matrix corresponding to the system of equations satisfying the particular form specified the above is very sparse; it has exactly four 1's in each row and exactly three 1's in each column. A parity check matrix of this type is called a $(3, 4)$-*low density parity check matrix* (an *LDPC*$(3, 4)$ *matrix* in short), and a code defined using such a sparse parity check matrix is called a $(3, 4)$-*low density parity check code* (*LDPCC*$(3, 4)$ in short).

We would like to study the "average performance" (or more generally, "probabilistic behavior") of a randomized local search algorithm for the above decoding problem. Precisely speaking, when discussing such average performance we will assume the following probabilistic situation. (The condition (b), considering only noise vectors with pn 1's, is simply for simplifying our discussion. In the following, by pn we will mean $\lfloor pn \rfloor$.)

Our Average-Case Scenario: We assume that an input instance for the LDPCC$(3, 4)$ Decoding Problem is given as follows.

(a) A parity check matrix H is chosen uniformly at random from the set of all LDPC$(3, 4)$ matrices.

(b) A noise vector $\boldsymbol{v}$ is chosen uniformly at random from the set of all n bit 0,1-vectors with pn 1's, and then a syndrome $\boldsymbol{c} = (c_1, ..., c_m)$ is computed as $\boldsymbol{c} = H\boldsymbol{v}$.

Under this scenario, we may assume that a given instance for the problem has a solution with pn 1's; that is, we consider the problem under such a promise. Furthermore, since p is small,

```
program RandDECODE(H, (c_1, ..., c_m));        % Denote (c_1, ..., c_m) by c.
  (x_1, ..., x_n) ← (0,0, ..., 0);             % Denote (x_1, ..., x_n) by x̃.
  repeat t steps
  [ if Hx̃ = c then output the current assignment and halt;
  | choose one variable x_i at random from those with the highest "penalty";
  [ flip the value of x_i;
program end.
```

Figure 2: Randomized local search algorithm for LDPCC$(3,4)$ Decoding Problem

the probability that there exists any other solution having pn or less than pn 1's is negligible. Thus, the problem of finding a solution with the smallest number of 1's is equivalent to that of finding the solution guaranteed by the promise.

2.2 Our Randomized Decoder

For solving LDPCC$(3,4)$ Decoding Problem, we consider a randomized local search algorithm given in Figure 2.

Some notions need to be defined for explaining the algorithm. Consider any problem instance, i.e., a pair of a LDPC$(3,4)$ matrix H and a syndrome vector $\boldsymbol{c} = (c_1, ..., c_m)$. We may assume that the syndrome $\boldsymbol{c}$ is generated as $\boldsymbol{c} = H\boldsymbol{v}$ from some noise vector $\boldsymbol{v} = (v_1, ..., v_n)$ with pn 1's. For any variable $\mathtt{x}_i$, a *check equation for* $\mathtt{x}_i$ is a equation of (3) with the variable $\mathtt{x}_i$. A variable $\mathtt{x}_j$ is *related to* $\mathtt{x}_i$ if it appears in one of the check equations for $\mathtt{x}_i$. Note that every variable has three check equations, and each variable has at most nine related variables. (In fact, it is quite likely that most variables has exactly nine related variables.)

Consider any point of an execution of the algorithm on H and $\boldsymbol{c}$, and let $\boldsymbol{a} = (a_1, ..., a_n)$ be a current assignment to variables $\mathtt{x}_1, ..., \mathtt{x}_n$. The *penalty* of a variable $\mathtt{x}_i$ is the number of check equations for $\mathtt{x}_i$ that are not satisfied under the current assignment. Note that penalty is an integer between 0 and 3. We say that a variable $\mathtt{x}_i$ is *wrong* (resp., *correct*) meaning that the variable is assigned wrongly (resp., correctly), that is, $a_i \neq v_i$ (resp., $a_i = v_i$). Note that a check equation is not satisfied if and only if it has odd number of wrong variables.

For example, suppose that a variable $\mathtt{x}_4$ has the following check equations, where their three syndrome bits are computed from an assignment stated as (b) to the corresponding variables.

(a)

$$\begin{aligned} \mathtt{x}_4 + \mathtt{x}_2 + \mathtt{x}_{11} + \mathtt{x}_{15} &= 0 \\ \mathtt{x}_4 + \mathtt{x}_{21} + \mathtt{x}_{23} + \mathtt{x}_{24} &= 1 \\ \mathtt{x}_4 + \mathtt{x}_{30} + \mathtt{x}_{39} + \mathtt{x}_{40} &= 0 \end{aligned}$$

(b)

$$0 + \begin{cases} 0+1+1 = 0 \\ 1+0+0 = 1 \\ 0+0+0 = 0 \end{cases}$$

Our goal is to compute the assignment given as (b). Thus, we say that a variable, e.g., $\mathtt{x}_2$ is wrong (w.r.t. the current assignment) if it's current value is 1. The penalty of $\mathtt{x}_4$ is determined by the status of variables appearing its check equations. For example, Figure 3 shows two example cases for which $\mathtt{x}_4$ has penalty 3. By using ○ (meaning "correct") and × (meaning "wrong"), we will symbolically express the status of the variable $\mathtt{x}_4$ and its related variables, which is called the *configuration* of $\mathtt{x}_4$.

$$1 + \begin{cases} 0+0+0 &= 1 \\ 1+0+0 &= 0 \\ 0+0+0 &= 1 \end{cases} \Leftrightarrow \times \begin{cases} \bigcirc & \times & \times \\ \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc \end{cases}$$

(a) The case where $\mathtt{x}_4$ itself is wrong.

$$0 + \begin{cases} 0+1+0 &= 1 \\ 0+0+0 &= 0 \\ 0+1+0 &= 1 \end{cases} \Leftrightarrow \bigcirc \begin{cases} \bigcirc & \bigcirc & \times \\ \times & \bigcirc & \bigcirc \\ \bigcirc & \times & \bigcirc \end{cases}$$

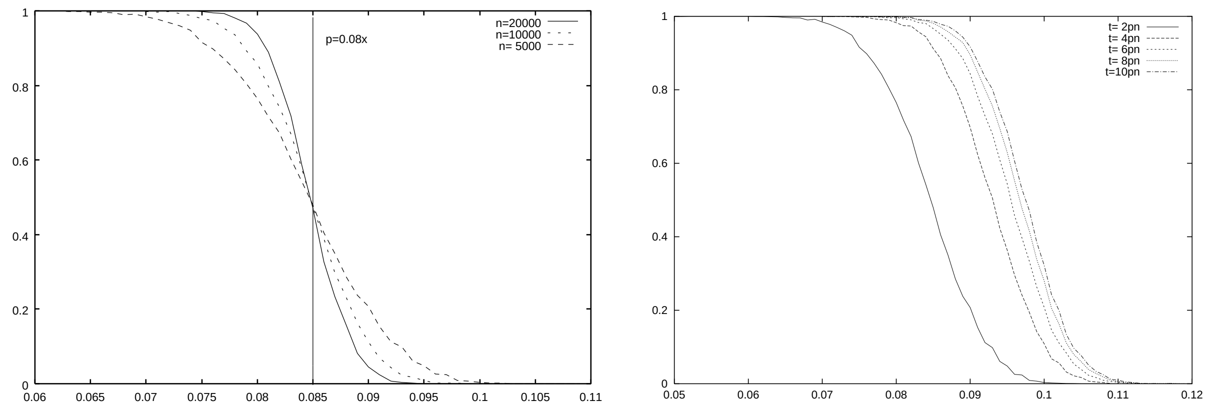
(b) The case where $\mathtt{x}_4$ is correct.

Figure 3: Two examples of an assignment for which $\mathtt{x}_4$ has penalty 3

Recall that `RandSAT` has two parameters S and t. Here, for the sake of simplicity, we set $S = 1$; that is, `RandDECODE` tries only one sequence of random search. On the other hand, while the choice of t is somewhat arbitrary, we chose $t = 2pn$ for most of our experiments. (Note that t should be at least pn because at least pn bits need to be flipped to reach the correct noise vector.)

We conducted computer experiments under our average-case scenario; these experiments show that the algorithm works quite well when p is small, say $p \leq 0.08$. Figure 4 (a) is for the relation between noise probability p and "success probability", the probability that the algorithm succeeds to find a correct noise vector when it is executed on a random parity check matrix and a random noise vector with pn 1's for various p. It shows that the algorithm succeeds with high probability when $p < 0.08$ and fails with high probability when $p > 0.09$. Figure 4 (b) indicates that the success probability is improved by using larger step bound t. (Three lines in Figure 4 (a) are for size $n = 5000$, 10000, 20000, where the range of noise probability is $p = 0.07 \sim p = 0.11$. Four lines in Figure 4 (b) are for $t = 2pn$, $4pn$, $6pn$, and $10pn$, where n is fixed to 5000. For each size and each noise probability, the algorithm is executed 20 times.)

For our analysis, we modified the algorithm `RandDECODE` slightly. We introduce the notion of "weight"; each variable $\mathtt{x}_i$ is assigned weight, which is simply $W_0^{pen_i - 1}$, where pen_i is the penalty of $\mathtt{x}_i$ and W_0 is some number determined by n. (A variable with 0 penalty is assigned 0 weight.) Then instead of choosing a flipped variable among those with the highest penalty, it is chosen from *all* variables according to their weights. With large enough W_0, e.g., $W_0 \approx n$, we can naturally expect that a variable with the highest penalty is usually flipped; hence, this modification does not change the performance of the algorithm. In fact, we checked experimen-



(a) p vs. success prob. ($t = 2pn$)

(b) p vs. success prob. ($n = 5000$)

Figure 4: Average-case performance of `RandDECODE`

tally that the difference is quite small by choosing $W_0 = n/5$ (i.e., $W_0 = 1000$ for $n = 5000$). Thus, we will analyze this modified algorithm.

3 Analysis

The state of the algorithm at each step is an assignment to variables $\mathtt{x}_1, ..., \mathtt{x}_n$; but we can also use a tuple of *variable configurations* like the one given in Figure 3 for describing this internal state. In other words, an execution of the algorithm is regarded as a process of changing variable configurations. Suppose that the configuration of variable $\mathtt{x}_4$, at some step of an execution, is the one given (a) of Figure 3, and that $\mathtt{x}_4$ is chosen for a flip variable. Then after flipping, $\mathtt{x}_4$ becomes correct, which changes the configurations of all related variables as well as the configuration of $\mathtt{x}_4$ itself. For example, the configuration of $\mathtt{x}_2$ may be changed as follows:

$$\begin{array}{c}\mathtt{x}_2\\ \bigcirc\end{array} \left\{ \begin{array}{ccc} \mathtt{x}_4 & \mathtt{x}_{11} & \mathtt{x}_{15} \\ \times & \times & \times \\ \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc \end{array} \right. \quad \overset{\text{flip } \mathtt{x}_4}{\Longrightarrow} \quad \begin{array}{c}\mathtt{x}_2\\ \bigcirc\end{array} \left\{ \begin{array}{ccc} \mathtt{x}_4 & \mathtt{x}_{11} & \mathtt{x}_{15} \\ \bigcirc & \times & \times \\ \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc \end{array} \right.$$

3.1 Approximation I

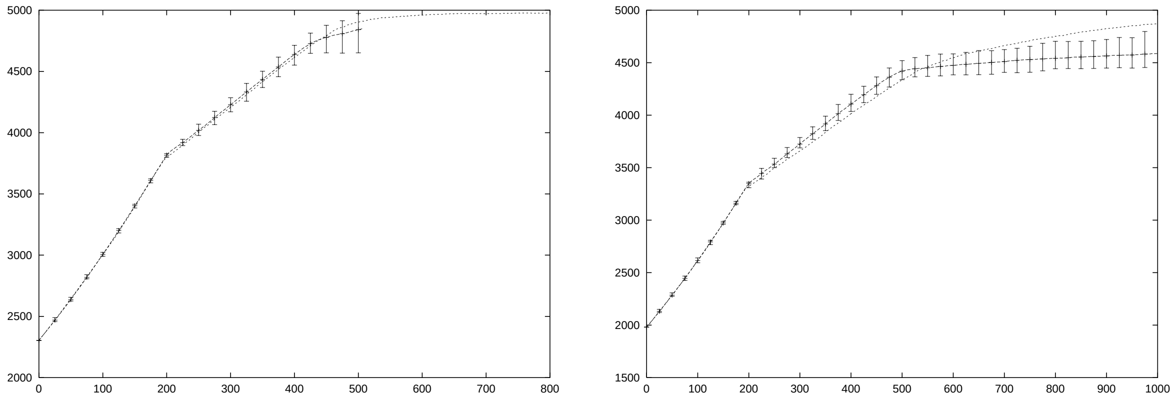
The first step of our analysis is to approximate the above random process by a much simpler one. Consider the same example as above. Flipping (the value of) the variable $\mathtt{x}_4$ causes a change in the configurations of all related variables; these related variables are determined by the parity check matrix H that is randomly chosen but fixed prior to the execution. But here we assume that related variables are chosen randomly *at each step* from all "legitimate" variables. Clearly, a variable $\mathtt{x}_i$ cannot be related to $\mathtt{x}_4$ unless it has a check equation of one of the following types. (Here $\boxed{\times}$ indicates the variable $\mathtt{x}_4$.) The other variables are considered "legitimate" and related variables are chosen uniformly at random from such variables.

$$\boxed{\times} \left\{ \begin{array}{ccc} \bigcirc & \times & \times \\ \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc \end{array} \right. \quad \Leftarrow \quad \begin{array}{c} \text{if } \mathtt{x}_i \text{ is } \bigcirc \\ \bigcirc\ \boxed{\times}\ \times\ \times \\ \bigcirc\ \boxed{\times}\ \bigcirc\ \bigcirc \end{array} \qquad \begin{array}{c} \text{if } \mathtt{x}_i \text{ is } \times \\ \times\ \boxed{\times}\ \times\ \bigcirc \\ \end{array}$$

In other words, instead of choosing related variables following the random relationship between variables fixed prior to an execution, related variables are chosen randomly *at each step*. In this simple random process, the name of variables are no longer important, and an internal state of an execution is expressed by a tuple $(n_1, ..., n_k)$ where each n_i is the number of variables with the ith configuration. (We consider some ordering on variable configurations, and denote them by $\mathrm{conf}_1, ..., \mathrm{conf}_k$. Assume in the following that conf_1 is the configuration with no wrong variable. Hence, the algorithm terminates when n_1 reaches to n.) Then the simple random process simulating an execution of the algorithm is to repeat the following for $2pn$ times or until n_1 becomes n.

(1) Choose one configuration *randomly* according to its weight. (We can naturally extend the notions of "penalty" and "weight" for configurations.)

(2) For the chosen configuration (let it be conf_i), choose nine configurations that are related to conf_i *uniformly at random* from all legitimate configurations.



(a) $p = 0.08$ (b) $p = 0.1$

The trace of the number of penalty 0 variables. A solid line is the average of 20 real executions, and a dashed line is the average of 10 simulations. Bars indicate the max and min numbers at every 25 step of the real executions.

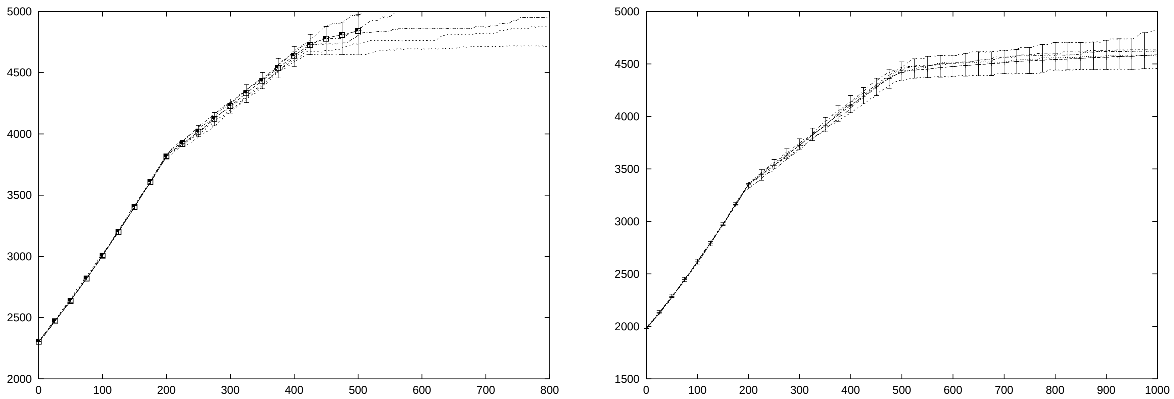
Figure 5: # of penalty 0 var.s: real execution vs. simulation by a simple random process

(3) Update $(n_1, ..., n_k)$, for simulating the step that a variable with conf_i is flipped.

Our computer experiments show that this simple random process can simulate a random execution of the algorithm. For example, Figure 5 shows how the number of penalty 0 variables grows during the execution. The simulation matches to the real execution quite well for $p = 0.08$. On the other hand, for $p = 0.1$, some difference can be seen towards the end, though the approximation is still reasonable as a whole.

Some Remark on Experiments: (The same remark applies for all experiments shown below.)

The experiments are conducted by using one fixed pair of a parity check matrix and a noise vector for size $n = 5000$. Initial numbers $(n_1, ..., n_k)$ for the simulation are computed from this



(a) $p = 0.08$ (b) $p = 0.1$

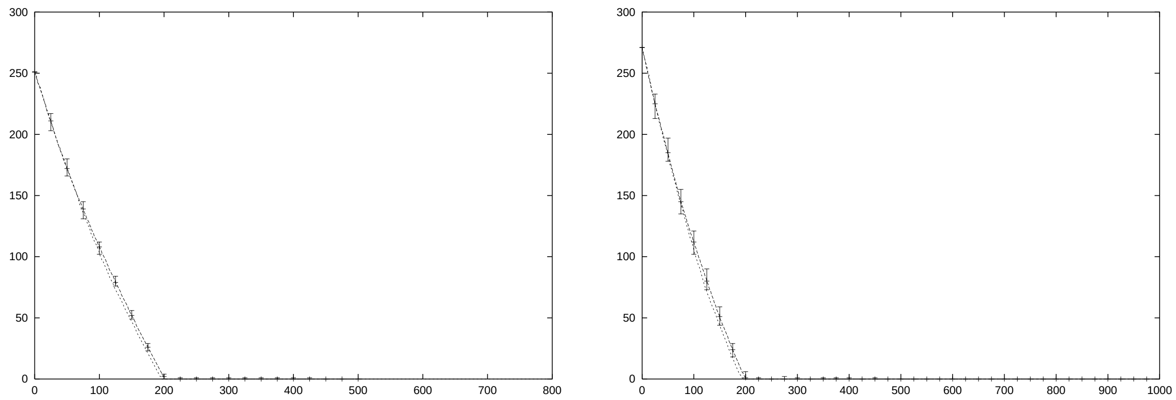
The trace of the number of penalty 0 variables. A solid line is the average of 20 executions, and the other lines are the results of some 4 executions respectively.

Figure 6: # of penalty 0 var.s: average and four executions

pair; that is, they are real ones. (It is possible to estimate $(n_1, ..., n_k)$ analytically from p; but for simplicity and comparison, we used real values.) The results are the average of 20 executions (for the algorithm) and 10 executions (for the simulation). When taking an average, we used data up to the smallest steps among all executions (resp., simulations). This is why the solid line in the graph of Figure 5 (a) stops at about the 500th step. Figure 6 shows the results of some four executions.

Let us see some other examples. Figure 7 shows how the number of penalty 3 variables decreases during the execution. Since such variables have a large weight, they are flipped first; Figure 7 shows this point clearly. Also note that the growth rate of penalty 0 variables changes (Figure 5) at the point when the number of penalty 3 variables becomes (almost) zero. For an extreme example, we consider the number of wrong variables with penalty 1. Figure 8 shows the results of some three executions and the average of 10 simulations. As we can see, the number of such variables are not large, and the difference is more visible. But the simulation can still approximate the real executions for $p = 0.08$. On the other hand, for $p = 0.1$, the simulation seems to give underestimated numbers. It is our future subject to analyze a reason for such difference.

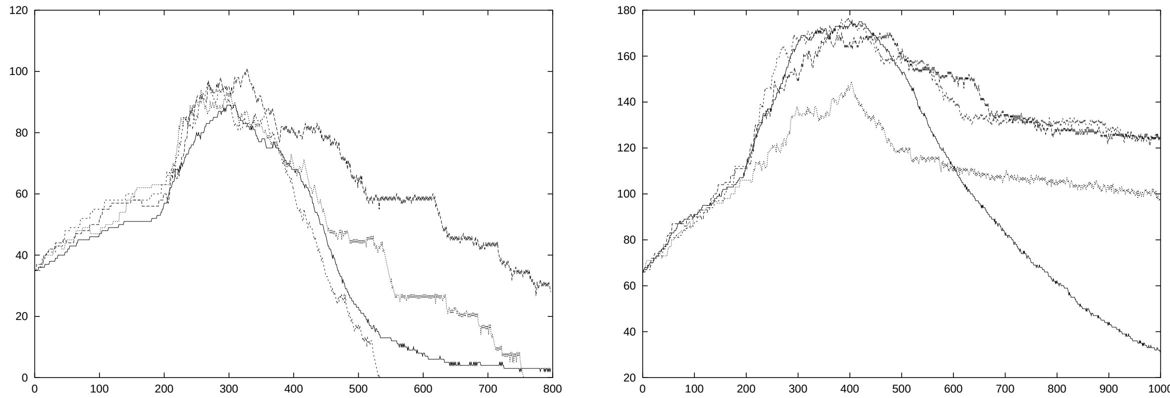
In general, these experiments show that our simulation can reasonably trace the number of variables of various types, particularly for the case when an execution succeeds to decode. We conjecture that the simulation captures the average behavior of the algorithm *at least* for the case when the algorithm works on average. We also think that by investigating why the simulation diverges from real executions, we may be able to find out some reasoning why they fail to reach to the solution. For such investigation, we believe our next approximation would be a useful tool.



(a) $p = 0.08$ (b) $p = 0.1$

The trace of the number of penalty 3 variables. A solid line is the average of 20 real executions, and a dashed line is the average of 10 simulations. Bars indicate the max and min number at every 25 step of the real executions.

Figure 7: # of penalty 3 var.s: real execution vs. simulation



(a) $p = 0.08$ (b) $p = 0.1$

The trace of the number of wrong variables with penalty 1. A solid line (i.e., the one with not so many zigzags) is the average of 10 simulations. The other lines are the results of some three executions respectively.

Figure 8: # of wrong var.s with penalty 1: simulation and three executions

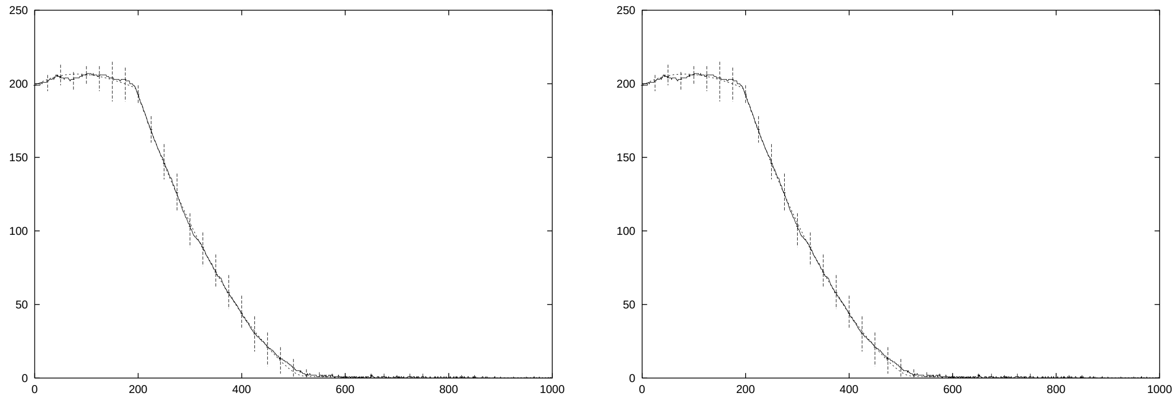
3.2 Approximation II

Though simple, our simulation is still a random process. Here we propose one more approximation for analyzing this random process *analytically* or *deterministically*. The idea is simple. Instead of randomly choosing some configuration and updating $(n_1, ..., n_k)$, we simply increase/decrease n_i by the probability that conf_i is chosen for flipping and the probability that conf_i is chosen as one of related configurations. These probabilities are computable from $(n_1, ..., n_k)$. That is, we can define a recurrence formula for updating $(n_1, ..., n_k)$. We hope that such a formula can approximate the average of $(n_1, ..., n_k)$.

$$\times \left\{ \begin{array}{ccc} \circ & \circ & \circ \\ \circ & \circ & \circ \\ \times & \circ & \circ \end{array} \right.$$

Clearly, a probability calculation like this does not work in general. Nevertheless, our experiments show that it works. For example, consider an experiment tracing the number n_9 of the 9th configuration conf_9 (see right). As shown in Figure 9, an estimation computed by the recurrence formula matches to the simulation result. This similarity can be observed for all the other configurations. Therefore, we conjecture that our probabilistic recurrence formula can approximate, to a certain extent, simulation results, and we propose to use this approximation for investigating the relation between the real execution and the simulation. A mathematical analysis of this conjecture is our future subject.

Since we now have a recurrence formula for estimating $(n_1, ..., n_k)$, we can estimate the average number of steps until n_1 reaches to n, i.e., the algorithm reaches to the solution. Figure 10 shows how this number grows when noise probability p increases. Then by using lines for step bounds, we can estimate noise probability such that the algorithm with each step bound fails. Although there is some difference, these estimations are good upper bounds for the real bounds of Figure 4 (b). While a more detailed investigation is necessary to solve the difference, even this estimation may be helpful for improving the algorithm and/or designing good parity check matrices [RSU01].



(a) $p = 0.08$ (b) $p = 0.1$

The trace of the number of the 9th configurations. A solid line is the average of 20 simulations, and a dashed line is the result of a calculation by a recurrence formula. Bars indicate the max and min number at every 25 step of the simulations.

Figure 9: # of conf_9: simulation vs. calculation by a recurrence formula

Acknowledgments

We thank Prof. Y. Kabashima and members of his research group (in particular, , Dr. T. Murayama and Dr. K. Nakamura) for their encouragements and their help as sources on LDPCC and Belief Propagation techniques.

References

[Gal62] R.G. Gallager, Low density parity check codes, *IRE Trans. Inform. Theory*, **IT-8**(21), 21–28, 1962.

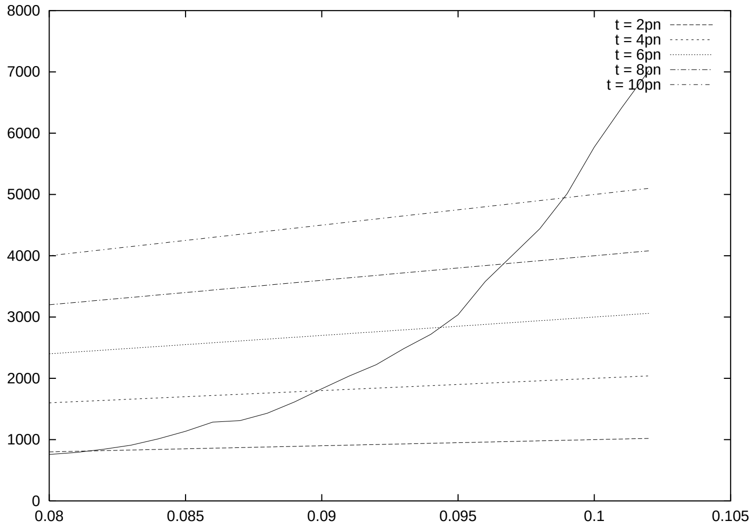


Figure 10: Estimated average number of steps to reach the solution

[Gen98] I. Gent, On the stupid algorithm for satisfiability, Report APES-03-1998, `http://www.cs.strath.ac.uk/~apes/reports/apes-03-1998.ps.gz`, 1998.

[Gu92] J. Gu, Efficient local search for very large-scale satisfiability problems, *SIGART Bulletin* 3, 8–12, 1992.

[HS00] H. Hoos and T. Stützle, Local search algorithms for SAT: An empirical evaluation, in *SAT 2000: Highlights of Satisfiability Research in the Year 2000* (I. Gent, etal eds.), IOS Press, 43–88, 2000.

[KP92] E. Koutsoupias and C.H. Papadimitriou, On the greedy algorithm for satisfiability, *Information Processing Letters*, 43(1), 53–55, 1992.

[MKSV00] T. Murayama, Y. Kabashima, D. Saad D, and R. Vicente, *Phys. Rev. E*, **62**, 1577–1591, 2000.

[Mac99] D. MacKay, Good error-correcting codes based on very sparse matrices, *IEEE Trans. Inform. Theory*, **IT-45**(2), 399–431, 1999.

[RSU01] T. Richardson, M. Shokrollahi, and R. Urbanke, Design of capacity-approaching irregula low-density parity-check codes, *IEEE Trans. Inform. Theory*, **IT-47**, 619-637, 2001.

[Sch99] U. Schöning, A probabilistic algorithm for k-SAT and constraint satisfaction problems, in *Proc. of the 40th Ann. IEEE Sympos. on Foundations of Comp. Sci.* (FOCS'99), IEEE, 410–414, 1999.

[SLM92] B. Selman, H. Levesque, and D. Mitchell, A new method for solving hard satisfiability problems, in *Proc. of the 10th National Conference on Artificial Intelligence* (AAAI'92), MIT Press, 440–446, 1992.