Analysis of a Randomized Local Search Algorithm for LDPCC Decoding Problem

Osamu Watanabe, Takeshi Sawai, and Hayato Takahashi
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
watanabe@is.titech.ac.jp

Abstract.
We propose an approach for analyzing the average performance of a given (randomized) local search algorithm for a constraint satisfaction problem. Our approach consists of two approximation steps. we experimentally investigate a randomized algorithm for LDPCC decoding and discuss the reliability of these approximations.

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. Well known 3SAT is a typical example of such problems.

A "local search" is simple yet one of the important algorithmic approaches for solving such constraint satisfaction problems. Here we focus on "randomized" local search algorithms, local search algorithms making some algorithmic decision randomly. For example, for 3SAT Problem, an algorithm given in Figure 1 is one of the typical examples of such algorithms. (This algorithm is one of the *WalkSAT* family [MSK97]. The idea of this algorithm can be found in [Pap91].) Though very simple, it has been known that this randomized local search algorithm works well. More precisely, when parameters S and t are chosen appropriately, `RandSAT` succeeds to find a satisfying assignment (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.

It may be difficult to give a rigorous mathematical analysis of 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 the average/random execution of the algorithm by a simple random process.
II. Approximate this process by simple probabilistic recurrence formulas.

For our technical discussion we will consider one constraint satisfaction problems — LDPCC Decoding Problem — and a randomized local search algorithm for this problem. Using the above

```
program RandSAT(F = C_1 ∧ C_2 ∧ ··· ∧ C_m);
  repeat S times
  [   X_1, ..., X_n ← a randomly chosen assignment;
      repeat t steps
      [   if F = 1 then output the current assignment and halt;
          choose one unsat. clause C_k of F randomly,
              and one variable X_i in C_k randomly;
          flip the value of X_i;
      ]
  ]
program end.
```

Figure 1: Local search algorithm for 3SAT

approach, we analyze the average performance of this algorithm. We give some experimental evidences that these approximations are reasonable ones. Their theoretical analysis is our future open problem.

2 LDPCC$(3,4)$ Decoding and Our Goal

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

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

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

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

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, $1 \leq j \leq m$, and
- each x_i appears exactly three times (which implies $m = 3n/4$).

Output: An assignment to $x_1, ..., x_n$ satisfying (3) with the smallest number of 1's. (See also an explanation of the next subsection.)

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]) 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[1].

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.

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. However, from the average-case scenario we will assume (see below), the actual requirement for the solution is an assignment that satisfies (3) *and* that has exactly pn 1's. (One can prove, for small p, such an assignment is unique, for almost all input instances. Throughout this paper, by pn we mean $\lfloor pn \rfloor$.) Thus, the problem should not be regarded as an NP type optimization problem; rather, it should be regarded as an NP type search 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*$(3, 4)$ or *LDPCC* 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 the following probabilistic situation has been usually considered, see, e.g., [Mac99].

---

[1]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.

program RandDECODE(H, $(c_1, ..., c_m)$); % Denote $(c_1, ..., c_m)$ by $\boldsymbol{c}$.
($\mathtt{x}_1$, ..., $\mathtt{x}_n$) $\leftarrow$ (0,0, ..., 0); % Denote $(\mathtt{x}_1, ..., \mathtt{x}_n)$ by $\mathtt{x}$.
repeat t steps
[**if** $H\mathtt{x} = \boldsymbol{c}$ **then** output the current assignment and halt;
choose *one* variable $\mathtt{x}_i$ at random from those with the highest "penalty";
flip the value of $\mathtt{x}_i$;
program end.

Figure 2: Randomized local search algorithm for LDPCC Decoding Problem

The Average-Case Scenario for LDPCC Problem:
An input instance 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 $\boldsymbol{c} = H\boldsymbol{v}$.
Then the goal is to compute an assignment to $\boldsymbol{x} = (x_1, ..., x_n)$ that has pn 1's and that satisfies (3) for given H and $\boldsymbol{c}$.

2.2 Our Randomized Decoder

For solving LDPCC Decoding Problem, we consider a randomized local search algorithm given in Figure 2. (In this algorithm the outer loop of the algorithm of Figure 1 is omitted.)

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 on* $\mathtt{x}_i$ is a equation of (3) having the variable $\mathtt{x}_i$. We say that a variable $\mathtt{x}_j$ is *related to* $\mathtt{x}_i$ if it appears in one of the check equations on $\mathtt{x}_i$. 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.)

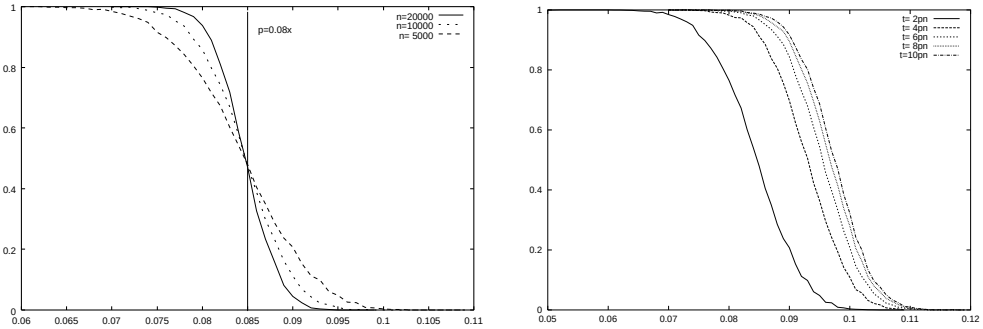
Consider any point of an execution of the algorithm on H and $\boldsymbol{c}$. The *penalty* of a variable $\mathtt{x}_i$ is the number of check equations on $\mathtt{x}_i$ that are not satisfied under the current assignment to variables $\mathtt{x}_1, ..., \mathtt{x}_n$. (Precisely speaking, "penalty" is not for a variable but for the current assignment to the variable. But we simply say, e.g., the penalty of $\mathtt{x}_4$.)

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)
$$0 + \left\{ \begin{aligned} 0+1+1 &= 0 \\ 1+0+0 &= 1 \\ 0+0+0 &= 0 \end{aligned} \right.$$

Algorithm's goal is to compute the assignment given as (b). For example, since all variables are initially assigned 0, the penalty of $\mathtt{x}_4$ (under the initial assignment) is 1. On the other hand,



(a) p vs. success prob. ($t = 2pn$) (b) p vs. success prob. ($n = 5000$)

Figure 3: Average-case performance of `RandDECODE`

if only $\mathtt{x}_4$ and $\mathtt{x}_{21}$ are assigned 1 and the other eight related variables are assigned 0, then $\mathtt{x}_4$ has penalty 3. Note the following simple facts: (i) penalty is an integer between 0 and 3, (ii) the situation that a variable $\mathtt{x}_i$ has penalty 0 means that all three check equations on $\mathtt{x}_i$ are satisfied (i.e., consistent with the given syndrome), but (iii) it does not necessary mean that $\mathtt{x}_i$ and its all related variables are assigned correctly. The algorithm terminates if and only if all variable have penalty 0. Although it does not necessary mean that the correct noise vector $\boldsymbol{v}$ (used to produce the given syndrome $\boldsymbol{c}$) is found, it is quite unlikely that the algorithm terminates by finding some other noise when small t (e.g., $t = 2pn$) is used. (This property is provable for, e.g., $t = 2n$ from the fact that for almost all H, there is no pair of vectors $\boldsymbol{v}$ and $\boldsymbol{v}'$ both having at most $2pn$ 1's that give the same syndrome w.r.t. H.)

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 3 (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 syndrome computed from a random noise vector with pn 1's for various p. We used the step bound $t = 3pn$ in this experiment. It shows that the algorithm succeeds with high probability when $p < 0.08$ and fails with high probability when $p > 0.09$. Figure 3 (b) indicates that the success probability is improved by using larger step bound t. (Three lines in Figure 3 (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 3 (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 randomly chosen matrices and noise vectors.)

As shown in these graphs, the success probability drops sharply at some noise probability, and by using a larger step bound, we can postpone this drop though a limit seems to exist. Let us call this dropping point a "success threshold"; more precisely, a *success threshold* is the noise probability such that the success probability becomes 0.5. Here for a technical goal of our analysis, we try to estimate this success threshold for $n = 5000$ and various step bounds t.

For our analysis, we modify the algorithm `RandDECODE` slightly. We introduce the notion of "weight"; each variable is assigned weight, which is simply W^{g-1}, where g is the penalty of the variable. W 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, e.g., $W \approx n$, 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 our following experiments, we will use $W = 1000$ for $n = 5000$, and we will analyze the algorithm under this modification.

3 Analysis

First let us see what the algorithm does at each step. Suppose at some step of the execution, a variable $\mathtt{x}_4$ is chosen as a flipped variable. The state of $\mathtt{x}_4$ is specified by its current value and the values of nine related variables, i.e., the variables appearing three check equations on $\mathtt{x}_4$. Here instead of using 0 or 1 values, we use symbols ○ and × indicating respectively the current value is correct and wrong w.r.t. the target solution. We also add penalty information; that is, the state of each variable is specified by a pair of its penalty and correctness. Suppose, for example, that the states of $\mathtt{x}_4$ and its related variables are changed as follows by flipping the value of $\mathtt{x}_4$.

$$
\begin{array}{c}\mathtt{x}_4\\(2,\times)\end{array}
\left\{\begin{array}{ccc}\mathtt{x}_2 & \mathtt{x}_{11} & \mathtt{x}_{15}\\ (1,\bigcirc) & (2,\times) & (3,\times)\\ (1,\bigcirc) & (1,\bigcirc) & (2,\times)\\ (2,\bigcirc) & (1,\bigcirc) & (1,\bigcirc)\end{array}\right.
\quad \overset{\text{flip } \mathtt{x}_4}{\Longrightarrow} \quad
\begin{array}{c}\mathtt{x}_4\\(1,\bigcirc)\end{array}
\left\{\begin{array}{ccc}\mathtt{x}_2 & \mathtt{x}_{11} & \mathtt{x}_{15}\\ (0,\bigcirc) & (1,\times) & (2,\times)\\ (2,\bigcirc) & (2,\bigcirc) & (3,\times)\\ (1,\bigcirc) & (0,\bigcirc) & (0,\bigcirc)\end{array}\right.
$$

Like the above, a tuple of $1 + 9$ penalty-correctness pairs is called a *variable configuration*. By this flip, the variable configuration of $\mathtt{x}_4$ is changed. Note that the variable configuration of every related variable is also changed by the flip. That is, one step of the algorithm is to change the variable configurations of a flipped variable and its related variables.

Note here that for a flipped variable, new penalty after the flip is determined by its current penalty. For example, as in the above example, if the penalty of a flipped variable is 2, then it becomes 1 after the flip. Similarly, penalty 3 becomes 0, and penalty 1 becomes 2. On the other hand, the penalty of each related variable is changed depending on the status of the check equation on the flipped variable that it appears. The penalty of a related variable gets decremented (resp., incremented) if the variable appears in an unsatisfied (resp., satisfied) equation. For example, the penalty of $\mathtt{x}_2$ is changed from 2 to 1 because it appears in an unsatisfied equation, i.e., an equation that is not consistent with the given syndrome. On the other hand, since the second equation is satisfied, the penalty of every variable in the second equation gets incremented by the flip.

In our experiments, we focus on the number of variables of each variable configuration type. That is, we trace how those numbers are changed during the execution. But in the following explanation we consider, for the sake of simplicity, a tuple $(n_0^+, n_1^+, n_2^+, n_3^+, n_0^-, n_1^-, n_2^-, n_3^-)$ of eight numbers, where each n_g^+ (resp., n_g^-) denotes the number of variables with penalty g that is correctly (resp., wrongly) assigned. We will call this tuple a *numerical configuration*.

From this view point, each step of the algorithm is to update this numerical configuration. For example, by flipping $\mathtt{x}_4$ as above, n_2^+ is decremented, and n_1^+ is incremented. Besides the penalty of every related variable is also changed. For example, due to the penalty change of $\mathtt{x}_2$, n_1^+ gets decremented and n_0^+ gets incremented. Similarly, the numerical configuration must be updated based on the penalty changes for the other eight variables.

3.1 Approximation I

Suppose that we can calculate expected numerical configuration at a given step within reasonable accuracy. Then we have enough information on the average behavior of the algorithm. For example, by computing the step when n_0^+ reaches to n, the total number of variables, we can estimate the average number of steps until the algorithm terminates. This is our ultimate goal. Towards this goal, in this paper, we propose two step approximations.

The first approximation is to simulate the execution by a simple Markov type random process. Note that, at each step, the penalty and the correctness of a flipped variable is *randomly* determined; more specifically, a correctly (resp., wrongly) assigned variable with penalty g is chosen as a flipped variable with probability $P_g^+ = n_g^+ W^{g-1}/W_{\text{total}}$ (resp., $P_g^- = n_g^- W^{g-1}/W_{\text{total}}$), where $W_{\text{total}} = (n_1^+ + n_1^-) + (n_2^+ + n_2^-)W + (n_3^+ + n_3^-)W^2$.

On the other hand, the penalty and the correctness of related variables depend on the structure of the check equations on the flipped variable, i.e., sets of variables appearing in these equations. These sets are determined by H that is *randomly* chosen but *fixed* prior to the execution. Here for the first approximation, we assume that these related variables are chosen randomly at each step from all variables with a "legitimate" state. In other words, we select, in our approximation, penalty-correctness pairs for the related variables randomly at each step. Suppose, for example, some wrongly assigned variable with penalty 2 is chosen as a flipped variable. Since it's penalty is 2, two check equations are not satisfied (i.e., inconsistent with the syndrome), and one check equation is satisfied (i.e., consistent with the syndrome). That is, the status of the flipped variable and its check equations is as follows (see the left).

$$(2,\times)\left\{\begin{array}{lll} \boxed{1}\ \boxed{2}\ \boxed{3} & \leftarrow & \text{unsat.} \\ \boxed{4}\ \boxed{5}\ \boxed{6} & \leftarrow & \text{unsat.} \\ \boxed{7}\ \boxed{8}\ \boxed{9} & \leftarrow & \text{sat.} \end{array}\right. \quad \Leftarrow \quad \begin{array}{lll} \boxed{1},\boxed{2},\boxed{3} & = & (1,\bigcirc),\ (0,\bigcirc),\ (0,\bigcirc) \\ \boxed{4},\boxed{5},\boxed{6} & = & (2,\times),\ (1,\bigcirc),\ (2,\times) \\ \boxed{7},\boxed{8},\boxed{9} & = & (0,\bigcirc),\ (1,\times),\ (1,\bigcirc) \end{array}$$

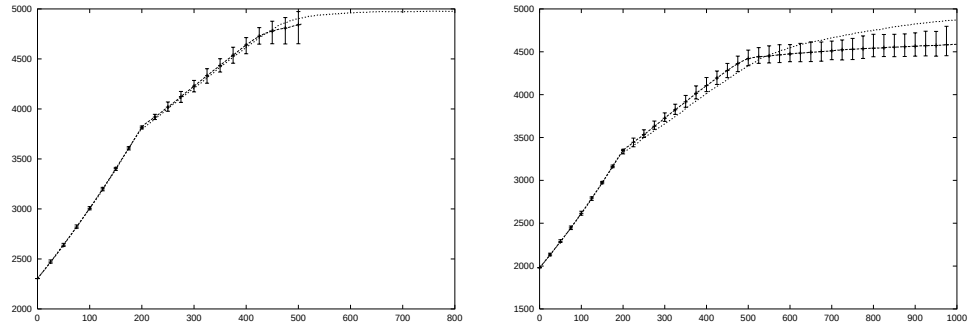
Then we choose a state (i.e., a penalty-correctness pair) for each related variable. Consider, for example, the first equation. Since the flipped variable is wrongly assigned and the first equation is unsatisfied, "legitimate" correct/wrong states for three variables in the equation are $\bigcirc\bigcirc\bigcirc$, $\bigcirc\times\times$, $\times\bigcirc\times$, ..., etc. Hence we choose, e.g., $(1,\bigcirc),(1,\bigcirc),(2,\bigcirc)$ with probability $P_{1,1,2}^{+++}|_{\text{unsat}} = n_1^+ \cdot n_1^+ \cdot n_2^+ / N_{\text{unsat}}$, where N_{unsat} is the total number of legitimate three penalty-correctness pairs for an unsatisfiable equation, which is calculated as follows.

$$N_{\text{unsat}} \ = \ n^+n^+n^+ + n^+n^-n^- + n^-n^+n^- + \cdots,$$

where $n^+ = n_0^+ + n_1^+ + n_2^+ + n_3^+$ and $n^- = n_0^- + n_1^- + n_2^- + n_3^-$. Once the states of nine related variables are chosen, we can update the numerical configuration according to the rule explained above.

In summary, we simulate the execution of the algorithm by repeating the following simple random process for t times (or until $n_0 = n_0^+ + n_0^-$ reaches n).

(1) Choose one penalty-correctness pair (g,s) for a flipped var. with prob. P_g^s.
(2) For each check equation with status $state \in \{\text{sat.}, \text{unsat.}\}$ choose three penalty-correctness pairs $(g_1,s_1),(g_2,s_2),(g_3,s_3)$ with prob. $P_{g_1,g_2,g_3}^{s_1s_2s_3}|_{state}$.
(3) Then update the numerical configuration as the algorithm does at each step.



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

Bars indicate the max and min number at every 25 step of the real execution. A solid line is for the real execution, and a dashed line is for the simulation.

Figure 4: Real execution vs. simulation by a simple random process

The important point here is that this process of updating a numerical configuration $(n_0^+, ..., n_3^+, n_0^-, ..., n_3^-)$ is completely determined the current numerical configuration. That is, this is a simple Markov process.

Our computer experiments show that this simple random process can simulate the actual execution well (to a certain extent, more precisely speaking). For example, Figure 5 shows how the number of penalty 0 variables (i.e., $n_0 = n_0^+ + n_0^-$) 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. (The experiments are conducted by using one fixed pair of a parity check matrix and a noise vector for size $n = 5000$. The initial numerical configuration is computed from this actual input[2]; that is, it is a real one. The results are the average of 20 executions (for the algorithm) and 10 executions (for the simulation). These averages are taken until the fastest execution terminates, which is for keeping enough number of trials. All experiments that will be shown below follow this style.)

For all the other numbers showing some aspects of the state, our experiments show that the simulation can reasonably approximate the real execution; in particular, for the case where the execution succeeds to decode. Thus, we conjecture that our first step approximation captures the average behavior of the algorithm *at least* for the case where the algorithm works on average. On the other hand, we think that by investigating why the simulation diverges from the real execution, we may be able to find out some reasoning why it fails to reach to the solution. For such investigation, we believe our next approximation would be a useful tool.

3.2 Approximation II

Though simple, our simulation of the execution is still a random process on a huge number of states. Here we propose one more approximation for analyzing this random process *analytically* or *deterministically*. The idea is simple. Instead of randomly choosing ten penalty-correctness

[2]It is possible to estimate the initial numerical configuration analytically from p; but for simplicity and comparison, we used a real value.

pairs and updating a numerical configuration $(n_0^+, ..., n_3^-)$, we simply update the numbers in the configuration according to the probabilities that it gets incremented/decremented by the flip. That is, we define a recurrence formula for updating a numerical configuration.

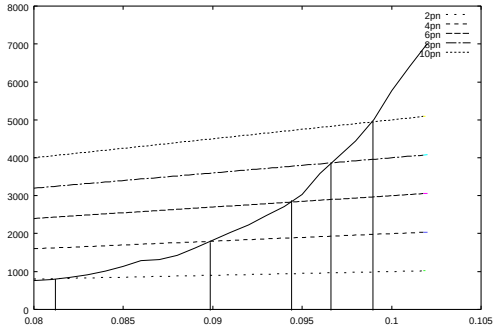
Although it is not so difficult to define such a reccurence formula, we omit stating it here because it becomes very long (even for our simpler numerical configuration). Instead we explain the idea by considering some example cases. Our random process chooses states (i.e., penalty-correctness pairs) for a flipped variable and its related variables, and then update a numerical configuration. For example, with probability P_2^-, a pair $(2, \times)$ is chosen for a flipped variable. If $(2, \times)$ were chosen, then n_2^- would get decreased by 1, and n_1^+ would get increased by 1 by the flip. Here we simply decrease n_2^- by P_2^- and increase n_1^+ by P_2^-. Similarly, the probability that $(1, \bigcirc), (1, \bigcirc), (2, \bigcirc)$ are chosen as penalty-correctness pairs for three related variables in one unsat. check equation on the flipped variable (when $(2, \times)$ is chosen for the flipped variable) is $P_{1,1,2}^{+++}|_{\text{unsat}}$, and if they were chosen, then n_1^+ would get decreased by 2, and n_2^+ would get decreased by 1, whereas n_0^+ would get increased by 2, and n_1^+ would get increased by 1. (As a total, both n_1^+ and n_2^+ would get decreased by 1, and n_0^+ would get increased by 2.) Hence, for this effect, we decrease n_1^+ and n_2^+ by $P_2^- P_{1,1,2}^{+++}$ and increase n_0^+ by $2P_2^- P_{1,1,2}^{+++}$. This is the idea of defining our updating formula.

Our computer experiments show that various numbers computed by a recurrence formula based on this idea matches to simulation results. In fact, by using the martingale technique, we can mathematically justify this approximation.

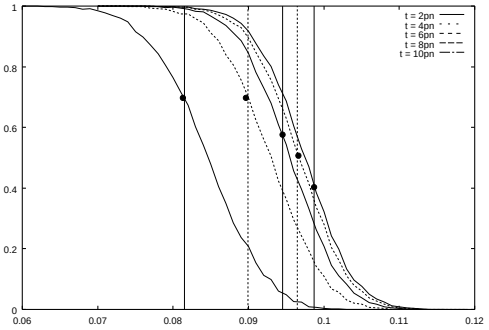
3.3 Prediction of Success Thresholds

For illustrating that our two step approximations are reasonable, we use the recurrence formula for a numerical configuration and estimate a success threshold for several step bounds $t = 2pn$, $4pn$, $6pn$, $8pn$, and $10pn$.

Suppose that our approximations are reasonably accurate. Then we have a recurrence formula, by which we can compute an average numerical configuration $(n_0^+, n_1^+, n_2^+, n_3^+, n_0^-, n_1^-, n_2^-, n_3^-)$ at each step. Thus, for a given p, we can compute the step when n_0 $(= n_0^+ + n_0^-)$ reaches to n, i.e., the step that the execution terminates; we can regard this number as the average number of steps needed for computing a noise vector with pn 1's. Figure 4 (a) shows how this number grows when noise probability p increases. Five linear lines indicate a step bound given by $t = 2pn$, $4pn$, $6pn$, $8pn$, and $10pn$. If the average number of steps exceeds this step bound, then we may consider that the algorithm fails to terminate within the step bound. That is, the intersection between each step bound line and the average step graph gives us an estimate for a success threshold for the corresponding step bound. In Figure 4 (b), we put these estimated success thresholds into the graph of Figure 3 (b). This illustrates that our prediction of success thresholds is reasonable. (It would be perfect if all black dots were on the success probability 0.5 line.)



(a) Estimation of the average number of steps



(b) Fig. 3 (b) with predicted success thresholds

Figure 5: Prediction of success thresholds

References

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

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

[MSK97] D. McAllester, B. Selman, and H. Kautz, Evidence for invariants in local search, in *Proc. AAAI'97*, MIT Press, 321–326, 1997.

[Pap91] C.H. Papadimitriou, On selecting a satisfying truth assignment, in *Proc. 32nd IEEE Sympos. on Foundations of Computing*, IEEE, 566–574, 1997.