信州大学
大学院総合理工学研究科

修士論文

全域木で表される検査行列を用いた sum-product 復号法の数値計算による性能評価

指導教員　西新 幹彦 准教授

専攻　工学専攻
分野　情報数理・融合システム分野
学籍番号　24W6043H
氏名　田村 涼真

2026 年 2 月 17 日

目次

1　はじめに

現代社会において，インターネットや電波を介した情報のやりとりが盛んに行われている．その際，送信者から受信者に情報を正しく送るために誤り訂正技術が導入されている．情報を伝送する際，符号器では誤り訂正のための冗長性が符号語に追加され，復号器では誤りを検出し訂正するためにその冗長性が利用される．誤り訂正技術は，我々の生活において必要不可欠であり，幅広く利用されている．通信を行う上で代表的な情報の符号化方法と復号方法として，LDPC 符号と sum-product 復号法がある．これらはそれぞれ生成行列，検査行列といった行列を用いる．検査行列は，2 部グラフで表現することができる．2 部グラフとは，頂点が 2 つの独立した集合に分かれ，異なる集合の頂点同士の間に限って辺が存在するグラフである．文献 [1] より，検査行列を 2 部グラフに変換した際，長さの短いループ（閉路）が存在すると，sum-product 復号法による検査行列の復号性能が劣化することが知られている．

高性能な検査行列の構成法に関する従来研究として，文献 [2] ではプロトグラフ LDPC 符号が提案されている．この符号は，小規模な 2 部グラフ（プロトグラフ）を基とし，M 次拡大を行うことで大規模な 2 部グラフを構成する手法である．具体的には，基となる検査行列の 1 成分を $M \times M$ の置換行列に，0 成分を $M \times M$ の全零行列に置換することで，所望の符号語長の検査行列を得る．この構成により，プロトグラフ LDPC 符号の漸近性能解析が容易になる．しかし，プロトグラフ LDPC 符号は必ずしも全域木構造を持つとは限らず，グラフ内にループを含む場合がある．

そこで本研究では，ループによる性能劣化の影響を排除し，検査行列の構造的な違いが復号特性に与える影響を純粋に評価することを目的とした．そのために，2 部グラフが全域木となるような検査行列を作成した．これにより，ループの影響を除外し，検査行列の大きさを統一した上で復号性能を比較することが可能となり，復号性能の優劣を分ける構造的特徴を分析することができた．また，得られた知見に基づき，高性能な検査行列を効率的に作成する方法について考察した．

一般に，長いループを含む検査行列は良好な復号性能を示し，実際に広く用いられている検査行列にもループが含まれている．このような検査行列に対しては，sum-product 復号法による厳密な周辺化計算が保証されず，近似周辺化アルゴリズムとして sum-product 復号法が用いられる．一方で，本研究では，検査行列の構造的な違いが復号特性に与える影響を明確に評価するため，sum-product 復号法による厳密な周辺化計算を重視し，ループを排除するという条件を設定した．

2　誤り訂正符号の復号方法の原理

本章では，本研究で用いた通信路モデルと線形符号の原理について述べる．本章では，文献 [1, 3, 4, 5] に基づき，通信路や復号方法の原理について述べる．

2.1　通信路

本研究で想定した通信路は，図 1 のような 2 元対称通信路である．2 元対称通信路は，通信路として最もシンプルな通信路である．送りたいシンボルは 0, 1 の 2 つであり，確率 p で異なるシンボルに変わってしまう通信路である．通信路モデルは，入力信号を表す確率変数 X と出力信号を表す確率変数 Y との間の条件付き確率を利用して記述される．つまり通信路の条件付き確率は，通信路が与えられた条件での信号の伝送確率を示す．

2.2　線形符号

本研究は，線形符号の構成に関する研究である．本節では，文献 [3] に基づいて線形符号の概要を説明する．2 元ベクトル空間 $\mathbb{F}_2$ のある部分空間を C と表記する．これを $\mathbb{F}_2$ 上の線形符号と呼ぶ．符号 C の次元を k とするとき，符号 C はベクトル空間であるから，k 個の基底ベクトル $(\boldsymbol{g}_1, \boldsymbol{g}_2, \ldots, \boldsymbol{g}_k)$ により張られる．2 元 $k \times n$ 行列 G を

$$G \triangleq \begin{pmatrix} \boldsymbol{g}_1 \\ \boldsymbol{g}_2 \\ \vdots \\ \boldsymbol{g}_k \end{pmatrix} \tag{1}$$

と定義する．この行列 G を C の生成行列と呼ぶ．この定義より，符号 C の任意の符号語 c は

$$c = m_1\boldsymbol{g}_1 + m_2\boldsymbol{g}_2 + \cdots + m_k\boldsymbol{g}_k \tag{2}$$

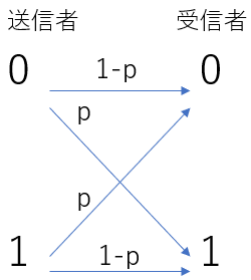

図 1　2 元対称通信路

のように基底ベクトルの線形結合として表せる．よって情報ベクトル $\boldsymbol{m} \triangleq (m_1, m_2, \ldots, m_k)$ を符号化したい場合，ベクトル $\boldsymbol{m}$ と生成行列 G の積

$$c = \boldsymbol{m}G \tag{3}$$

を計算すれば良い．このように生成行列が定まると情報ベクトルと符号語間に 1 対 1 の写像が定まる．次に，$m = n - k$ とする．2 元 $m \times n$ 行列 H がランク m を持ち，

$$GH^T = 0 \tag{4}$$

（H^T は行列 H の転置行列を表す）を満たすとき，H を符号 C の検査行列と呼ぶ．同じ符号 C を定義する生成行列 G と検査行列 H に成り立つ関係を考える．ある符号の $\mathbb{F}_2$ 上の生成行列を

$$G = (I_k \ P) \tag{5}$$

と仮定する．ここで I_k は $k \times k$ 単位行列であり，P は $k \times (n-k)$ 行列である．また，行列 P の ij 要素を $p_{ij} \in \mathbb{F}_2$ とする．ここで，行列

$$H = (-P^t \ I_{n-k}) \tag{6}$$

を考える．この H は右側に単位行列を含むため，ランクが $n - k = m$ であり，また

$$GH^T = -P + P = 0 \tag{7}$$

が成立する．したがって，定義より H は符号 C の検査行列となる．ここで生成行列 G の i 行目の行ベクトル g_i と検査行列 H の j 行目の行ベクトル h_j の積に注目する．このとき，$g_i h_j^t = -p_{ij} + p_{ij} = 0$ が任意の i において成り立つ．

G により生成される符号の符号語を $\sum_{i=1}^{k} c_i g_i \ (c_i \in \mathbb{F}_2)$ としたとき，

$$H\left(\sum_{i=1}^{k} c_i g_i\right)^t = H\left(\sum_{i=1}^{k} c_i {g_i}^t\right) \tag{8}$$

$$= \left(\sum_{i=1}^{k} c_i h_1 {g_i}^t, \sum_{i=1}^{k} c_i h_2 {g_i}^t, \ldots, \sum_{i=1}^{k} c_i h_{n-k} {g_i}^t\right) \tag{9}$$

$$= 0 \tag{10}$$

が成り立つ．上式の変形では，関係 ${g_i {h_j}^t} = 0$ を利用している．この関係は，生成行列 G により定義される線形符号 C_G の任意の符号語 c が，検査行列 H に対しても $Hc^t = 0$ を満たすことを意味する．すなわち，C_G に属する符号語はすべて検査行列 H が定義する符号に含まれる．以上より，検査行列 H の定義する線形符号を C_H とすると，$C_G \subset C_H$ がいえる．次に

それぞれの符号の含む符号語数を考える．C_G については 2^k である．H のランクは $n-k$ であり，C_H の符号語数も 2^k である．したがって，$C_G = C_H$ とわかる．すなわち，生成行列 G と検査行列 H は同じ線形符号を定義している．また上式の関係式は，生成行列から検査行列またはその逆を求める手法を与えている．

本研究では，検査行列のみ作成した．その理由は，2.4 節で述べる．

2.3 LDPC 符号

LDPC 符号は，疎な検査行列により定義される線形符号である．文献 [1] に基づいて説明する．疎な行列とは，行列内の非零要素の数が非常に少ない行列を指す．本研究では，$\mathbb{F}_2$ 上の 2 元 LDPC 符号のみを扱うため，以下のように定義される．2 元 $m \times n$ 行列 $H(0 < m < n)$ が与えられたとき，この行列を検査行列とみなすことにより，2 元線形符号 C が

$$C \triangleq \{x \in \mathbb{F}_2^n : Hx^t = 0\} \tag{11}$$

と定義される．H が疎な行列の場合，C は LDPC 符号と呼ばれる．疎な行列を検査行列として用いる理由は，LDPC 符号の復号特性に関係する．LDPC 符号の復号には，sum-product アルゴリズムが利用される．本研究で作成した検査行列は 2 部グラフに変換したときループがなく，全域木となっているため，行列内の非零要素の数が非常に少ない．したがって，全域木で表される検査行列は LDPC 符号の検査行列として適用可能である．

2.4 sum-product 復号法

LDPC 符号に対する代表的な復号法として sum-product 復号法がある．本節では，本研究で用いた確率領域 sum-product 復号法について文献 [1, 4] に基づいて説明する．

2 元 m 行 n 列の行列 H を復号したい LDPC 符号の検査行列とする．また，検査行列 H の m 行 n 列目要素 h_{mn} と表記する．整数の集合 $\{1, 2, \ldots, n\}$ の部分集合 $A(m)$, $B(n)$ を

$$A(m) \triangleq \{n : h_{mn} = 1\} \tag{12}$$

$$B(n) \triangleq \{m : h_{mn} = 1\} \tag{13}$$

と定義する．すなわち，$A(m)$ は検査行列 H の m 行目において，1 が立っている列インデックスの集合を意味し，$B(n)$ は検査行列 H の n 列目において 1 が立っている行インデックスの集合を指す．

受信語 $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ を受信したものと仮定する．また，2 元対称通信路を表す条件付き確率を

$$P(y|x)\ (x, y \in \mathbb{F}_2) \tag{14}$$

とする．ここで，x はこの復号法で推定する送信シンボルを表す変数である．確率領域 sum-product 復号法の詳細は次のようになる．

1. **ステップ 1(初期化)**$H_{mn} = 1$ を満たす全ての組 (m,n) に対して $q_{mn}(0) = 1/2$, $q_{mn}(1) = 1/2$ と初期設定する．また，反復回数のカウンタとする変数を $l = 1$ とし，最大反復回数を l_{max} に設定する．
2. **ステップ 2(行処理)**$m = 1, 2, \ldots, M$ の順に $H_{mn} = 1$ を満たす全ての組 (m,n) に対して，次の更新式を利用して $r_{mn}(0)$ と $r_{mn}(1)$ を

$$r_{mn}(0) = K \sum_{c_i \in F_2, i \in A(m)\backslash n \sum c_i = 0} \left\{ \prod_{n' \in A(m)\backslash n} q_{mn'}(c'_n) P(y'_n | c'_n) \right\} \tag{15}$$

$$r_{mn}(1) = K \sum_{c_i \in F_2, i \in A(m)\backslash n \sum c_i = 1} \left\{ \prod_{n' \in A(m)\backslash n} q_{mn'}(c'_n) P(y'_n | c'_n) \right\} \tag{16}$$

と更新する．ここで定数 K は，$r_{mn}(0) + r_{mn}(1) = 1$ が成り立つように定められる正規化定数とする．また，表記 $A(m)\backslash n$ は n を除く $A(m)$ に含まれる全ての変数が全ての可能な値 (定義域内のすべての値) をとることを意味する．本来，差集合は $A(m)\backslash\{n\}$ と表記すべきだが，表記を簡単にするためにこの記法を用いるものとする．
3. **ステップ 3(列処理)**$n = 1, 2, \ldots, N$ の順に $H_{mn} = 1$ を満たす全ての組 (m,n) に対して，次の更新式を利用して $q_{mn}(0)$ と $q_{mn}(1)$ を更新する．

$$q_{mn}(0) = K' \prod_{m' \in B(n)\backslash m} r_{m'n}(0) \tag{17}$$

$$q_{mn}(1) = K' \prod_{m' \in B(n)\backslash m} r_{m'n}(1) \tag{18}$$

ここで K' は $q_{mn}(0) + q_{mn}(1) = 1$ となるように定められる正規化定数とする．
4. **ステップ 4(一時推定語の計算)** $n = 1, 2, \ldots, N$ について

$$Q_n(0) = K'' P(y_n | x_n = 0) \prod_{m' \in B(n)} r_{m'n(0)} \tag{19}$$

$$Q_n(1) = K'' P(y_n | x_n = 1) \prod_{m' \in B(n)} r_{m'n(1)} \tag{20}$$

$$\hat{x} = \begin{cases} 0 & \text{if} Q_n(0) \geq Q_n(1) \\ 1 & \text{if} Q_n(0) < Q_n(1) \end{cases}$$

を計算する．ここで得られる $(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$ を一時推定語と呼ぶ．定数 K'' は $Q_n(0) + Q_n(1) = 1$ となるように定められる規格化定数である．

5. **ステップ 5(パリティ検査)** 一時推定語が符号語になっているかどうかを検査する．もし $(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$ が
$$(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)H^T = 0 \tag{21}$$
を満たせば，$(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$ を推定語として出力し，アルゴリズムを終了する．
6. **ステップ 6(反復回数のカウント)** もし $l < l_{max}$ ならば，l をインクリメントしてステップ 2 に戻る．$l = l_{max}$ ならば，$(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$ を推定語として出力し，アルゴリズムを終了する．

本研究では，確率的な挙動を多数回の試行によって近似するシミュレーションではなく，全ての符号語がデコーダに到着した場合を網羅的に考慮し，復号結果を厳密に求める数値計算を行った．また，シンボル 0 と 1 を等価に扱うため，復号過程で複数のシンボルの尤度が同一となった場合には，そのシンボル数に応じて受信語の復号誤り確率に 1/2 の重みを乗ずることとし，0 と 1 の扱いに偏りが生じないよう配慮した．さらに，本研究は 2 元対称通信路を扱っており，シンボル 0 と 1 が同じ確率構造で伝送されるため，符号語に対して「全ゼロメッセージを送った場合」の復号性能を計算すれば，他の全ての符号語についても同じ復号性能が得られる．この対称性に基づき，本研究では数値計算を全ゼロメッセージの場合に限定して行った．

3 グラフの生成方法

本章では，sum-product 復号法の正確な計算に必要な行列の性質について述べる．そして，そのような行列の生成方法について，本研究で実装したものについて説明する．

3.1 線形符号と 2 部グラフの対応

まず，検査行列を生成するために 2 部グラフを導入した．2 部グラフとは，頂点が 2 つの独立した集合に分かれ，異なる集合の頂点同士の間に限って辺が存在するグラフである．2 部グラフを構成するノード（頂点）は 2 つのグループに分かれ，それぞれ M 個と N 個のノードからなる．各グループに属するノードをそれぞれ変数ノードと関数ノードと呼ぶ．この 2 部グラフでは，$M \times N$ 行列の検査行列が与えられる．2 部グラフの例を図 2 に示す．2 部グラフで，j 番目の変数ノードと i 番目の関数ノードが辺で結ばれていたとき，検査行列の i 行 j 列目要素を 1 とする．これを全ての辺で適応させることで，2 部グラフに対する検査行列を生成できる．よって，図 2 のような 2 部グラフでは検査行列は，
$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{22}$$

となる．このように定義された 2 部グラフでは，j 番目の変数ノードの次数は検査行列の j 列目の重みに等しく，i 番目の関数ノードの次数は検査行列の i 行目の重みに等しくなる．[1]

本研究で用いる sum-product 復号法は，図 3 に示すように，2 部グラフにおける任意の変数ノード 1 点を基点としてつり上げ，ツリー構造の葉から根へメッセージを伝播させることで，各ビットの事後確率を算出する．そのため，異なる 2 部グラフであっても基点から見たツリー構造が同一であれば，sum-product 復号法が算出する確率値は等しくなり，結果として検査行列の復号性能も等しくなる．したがって，本研究の数値計算では，各検査行列に対応するツリー構造 (グラフの形状) のみを対象とすれば十分である．一方で，sum-product 復号法では，ツリー構造内にループが存在すると，sum-product アルゴリズムでの正確な計算が保証されず，グラフが非連結である場合，符号語が複数の独立した部分構造に分離され，検査行列が表す符号全体の関係を十分に反映できなくなる．すなわち，全ノードが連結していることで検査行列が表す符号全体の関係を完全に表現でき，任意のシンボル間でメッセージ伝播が可能となる．以上より，本研究で対象とするグラフは，ループを含まず，かつ全ノードが連結した構造に限定した．このように，ループを持たず，かつ全ノードが連結したグラフは，グラフ理論において全域木と呼ばれる．全域木に基づく検査行列を用いることで，sum-product 復号法が本来想定しているツリー構造上での厳密な計算を保証できる．これらの性質に基づき，本研究では 2 部グラフ全てを列挙するのではなく，各検査行列に対して得られるツリー構造 (グラフの形状) のみを列挙する手法を考案した．次節では，その手法について詳細に述べる．

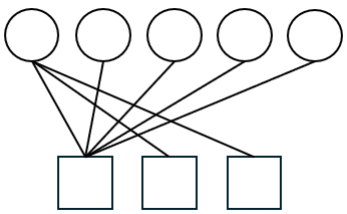

図 2　2 部グラフの例

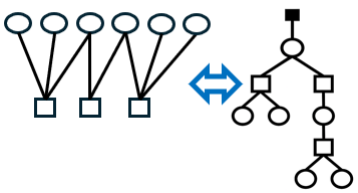

図 3　2 部グラフとツリー構造

3.2 全域木の作成方法

本節では文献 [6] に基づき，全域木の作成方法について，具体例を用いてその手順を説明する．この手法では，検査行列のサイズが指定されたとき，その行列で考えられる，グラフの形を全て列挙する．また，作成されるグラフは，前節で述べた全域木の条件満たすものである．

図 4 は本研究で実装した，全域木作成の流れを図示したものである．この例では，4×3 行列について，グラフを 7 つ列挙している．作成方法について詳細に述べる．手順全体では，完成までの過程を階層化し，グラフの形を 0 から作成している．各ステップでは前のステップで作成した全てのグラフに対して，変数ノードまたは関数ノードを 1 つずつ追加する．そして，追加したノードと既存のノードとをエッジでつなぐことによって，新たなグラフを作成する．この際，追加したノードと既存のノードの繋ぎ方によって複数のグラフを作成可能であるため，その全てを列挙する．そして，このような，ノードとエッジの追加を繰り返すことによって，最終的に所望のノードとエッジを持つグラフを作成することができる．

ここで図 4 にあるように，各ステップでグラフを列挙する際，元となるグラフが異なる場合でも，追加するノードとエッジによっては，同じグラフが作成されてしまうことがある．このため，各ステップでグラフを列挙したあと，すぐに次のステップに移るのではなく，列挙したグラフの中で重複しているものがないか調べ，重複があった場合削除してから，次のステップに進むことによって，効率的に行列の作成を行っている．

さらに，このような手順でグラフを構成することによって，前節で述べたループを持たず連結であるグラフを作成することが可能である．まず，各手順で追加したノードは必ず既存のノードと接続される．そのため，全てのノードは連結となる．また，ノードが n 個のグラフにおいて，全てのノードが連結でありかつ，ループを持つために必要なエッジの本数は，最低で n 本である．ここで，この手法ではノードを追加するたびに，新たに追加するエッジは 1 本となる．そのため，最終的に作成される n 個のノードを持つグラフにおいて，エッジは $n-1$ 本となる．よって，ループを作るためのエッジの本数が足りないため，上記の手順でループを作成することは不可能となる．

本研究では，上記の手法を用いて，行列生成を行っている．この手法を用いることで，行列のサイズを固定した上で，そのサイズで条件を満たした全てのグラフの形を作成できる．しかし，上記の手法には，行列サイズを拡大するにつれて，行列生成に要する時間が急激に増加するという課題があった．次節では，本研究において主に数値計算を行ったメッセージ長 2 の場合に適用可能な，より効率的なグラフの形の列挙手法を述べる．

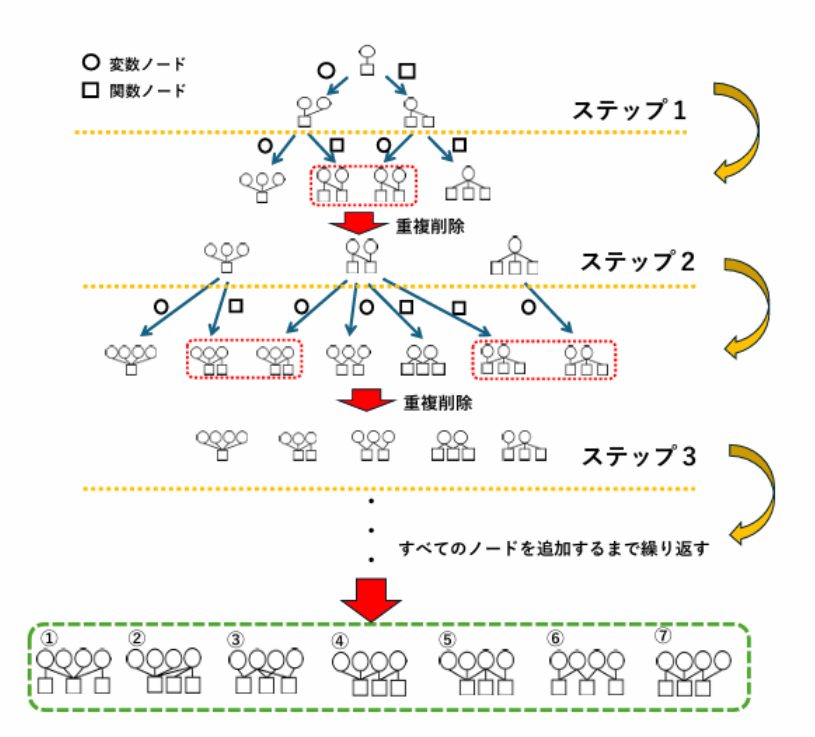

図 4　全域木の作成手順

3.3　メッセージ長 2 の場合の全域木の作成方法

前節で得られたグラフには，関数ノードが葉となり連結するグラフが複数存在していた．図 5 にその例を示す．赤ノードが変数ノード，青ノードが関数ノードを表している．図 5 では，中央部で関数ノードが葉として連結している．関数ノードが葉として存在すると，葉の関数ノードは単一の変数ノードのみを参照するため，その変数ノードに対してパリティ検査条件より値が必ず 0 に固定される．結果として，その変数ノードは自由に変動できず，符号語空間における自由度が 1 つ減少する．さらに，固定された値は sum-product 復号法のメッセージ伝播を通じて隣接する関数ノードや変数ノードに伝播するため，局所的な拘束が連鎖的に広がりやすい．これにより，検査行列の多様な誤りパターンへの対応力が低下し，復号性能が著しく劣化する．したがって，本研究では葉として連結している関数ノードを含むグラフを対象から除外した．

このとき，メッセージ長 2 の場合，条件を満たすグラフは 3 つの枝を持つ関数ノード（3 分岐ノード）を基準に作成すればよい．基準となる関数ノードに対し，残りのノードをどのよう

に分配するかを全通り列挙することで，条件を満たす全てのグラフの形を作成できる．ここで，本研究では $a-b-c$ という表記を用いて，3 分岐ノードから伸びる 3 本の枝に，それぞれ a 個，b 個，c 個のノードを連結した構造を表す．例えば，6×8 の検査行列に対応する全域木を列挙する場合には，$1-1-11, 1-3-9, 1-5-7, 3-3-7, 3-5-5$ の 5 通りの枝の分配を考えれば十分である．また，作成できる 5 通りのグラフを図 6 に示す．この手法により，前節の手法と比べて大幅に少ない探索回数で行列を作成できる．各メッセージ長でグラフの構造が異なるため，3 分岐ノードを基準に全域木を列挙できるのは，メッセージ長 2 の場合のみである．

4　復号性能の検証

本研究では，3.3 節の手法を用いて，メッセージ長が 2 の場合の検査行列の復号性能を調べた．作成した検査行列のサイズは，6×8 行列，7×9 行列，8×10 行列，9×11 行列，10×12 行列，11×13 行列，12×14 行列，13×15 行列，14×16 行列，15×17 行列，16×18 行列である．これらの行列を用いて実際に行った数値計算を説明する．

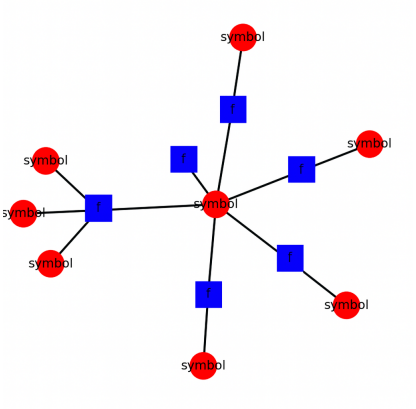

図 5　関数ノードを葉に持つグラフ

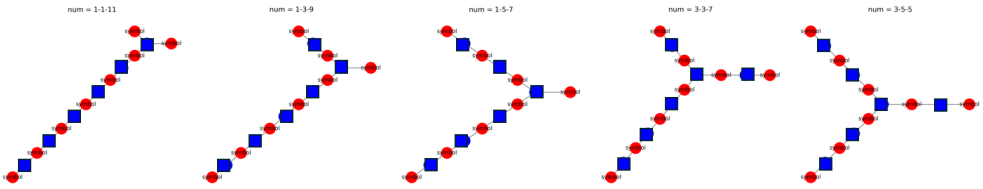

図 6　条件の下 6×8 行列で作成可能なグラフ

4.1 検証方法

本研究では，得られた検査行列を基にデコーダを作成した．2 元対称通信路での符号語の反転確率は，0.2 とした．その 2 元対称通信路，デコーダを用いて数値計算を行う．全ての符号語がデコーダに到着した場合を想定し，数値計算を行った．また，シンボル 0 と 1 を等価に扱うため，尤度が等しい場合には，尤度が等しくなったシンボル数に応じて，受信語の復号誤り確率に 1/2 の重みを乗ずることにした．これにより，復号処理における 0 と 1 の扱いに偏りが生じない．さらに，本研究では，2 元対称通信路を用いており，シンボル 0 と 1 の対称性が成り立つことから，全ゼロメッセージの場合に限定した．そして，検査行列の復号誤り確率を求めることで復号性能を調べた．

4.2 結果と考察

検査行列の復号誤り確率を数値計算により評価したところ，性能の良い検査行列のグラフの形，性能の悪い検査行列のグラフの形が存在した．表 1 に，6×8 行列に対する数値計算結果を示す．表 1 は，3 分岐ノードを基点としたときのグラフの構造と，その復号誤り確率との対応を表している．6×8 行列で最も性能が良かったグラフの形は，$3-5-5$ の構造であり，復号誤り確率は，0.155896 であった．実際のグラフの形を図 7 に示す．図 7 は中心に位置している 3 分岐ノードを基点に各エッジに 3 個，5 個，5 個のノードが連結しており，$3-5-5$ の構造となっていることが確認できる．

ここで本研究では，数値計算の結果とグラフの構造を総合的に比較した結果，性能の良否は，特定の構造的特徴と強く関係していることが明らかになった．その特徴により，本研究で行った検査行列は 3 つのグループに分類することができた．本研究では，列数に注目し，検査行列の「列数を 3 で割った余り」，すなわち mod3 に基づいて 3 つに分類した．なお，本研究では検査行列の形状を整理するにあたり，便宜的に「mod3」で分類して記述している．この分類はあくまで説明上の都合によるものであり，復号性能上の本質的な意味を持つものではない．

表 1　6×8 行列の数値計算の結果

グラフの形 (6×8)	復号誤り確率
$1-1-11$	0.37085
$1-3-9$	0.18211
$1-5-7$	0.21881
$3-3-7$	0.25971
$3-5-5$	0.15590

まず，列数が 2(mod3) のとき，3 分岐ノードに連結するノード数の「最大値」と「最小値」の差が 2 となる構造が最も高い性能を示した．6×8 行列は，この分類に属し，性能が最良であった構造は $3-5-5$ であり，この特徴と一致している．同様の傾向は，9×11 行列，12×14 行列，15×17 行列でも確認された．

次に，列数が 1(mod3) のとき，3 分岐ノードに連結するノード数の最大値と最小値の差が 6 となる構造が唯一存在し，これが常に最高性能を示した．これに分類される 8×10 行列の数値計算結果を表 2 に示す．8×10 行列で最も性能が良かった行列は，3 分岐ノードに連結するノード数の最大値と最小値の差が 6 となる構造の $3-5-9$ であった．この傾向は，11×13 行列，14×16 行列の場合でも同様に確認された．

最後に，列数が 0(mod3) のとき，列数が 1(mod3) の場合と同じく，3 分岐ノードに連結するノード数の最大値と最小値の差が 6 となる構造が最も高性能であった．これに分類される 7×9 行列の数値計算結果を表 3 に示す．7×9 行列で最も性能が良かった行列は，3 分岐ノードに連結するノード数の最大値と最小値の差が 6 となる構造の $1-7-7$ であった．ただし，

表 2　8×10 行列の数値計算の結果

グラフの形 (8×10)	復号誤り確率
$1-1-15$	0.36666
$1-3-13$	0.14856
$1-5-11$	0.20308
$1-7-9$	0.14604
$3-3-11$	0.22234
$3-5-9$	0.11249
$3-7-7$	0.18666
$5-5-7$	0.13178

表 3　7×9 行列の数値計算の結果

グラフの形 (7×9)	復号誤り確率
$1-1-13$	0.38134
$1-3-11$	0.15057
$1-5-9$	0.24922
$1-7-7$	0.12751
$3-3-9$	0.20125
$3-5-7$	0.14787
$5-5-5$	0.20518

列数が 1(mod3) の場合と異なるのは，そのような構造が 2 種類存在することである．7×9 行列では，差が 6 の構造として $1-7-7$，$3-3-9$ の 2 つが作成できるが，性能が良い検査行列の構造は，$1-7-7$ であった．両者の違いとして，本研究では「全体的に冗長かどうか」が重要な要因であると考察した．$1-7-7$ の構造は，$3-3-9$ の構造と比較して長いエッジが 2 本存在するため，冗長性が分散している．計算結果では，長いエッジが 1 本存在するよりも，適度に長いエッジが 2 本存在する場合の方が高性能であるという傾向が見られた．よって，列数が 0(mod3) の分類では，3 分岐ノードに連結しているノードの差が 6 である 2 つのグラフの内，適度に長い 2 本のエッジを持つグラフが最も高性能である．同様の傾向は，10×12 行列，13×15 行列，16×18 行列についても確認された．

一方，最も性能の悪い検査行列の構造は共通しており，$1-1-$ 残り全てのノード という構造だった．例として図 8 に 6×8 行列における最悪構造の検査行列を挙げる．これは 1 本のエッジのみが極端に長く，冗長性が局所的に集中しているため，復号性能を大きく劣化させていると考える．以上より，メッセージ長 2 の場合の性能の良い行列，悪い行列の特徴を明確に解明することができた．

4.3 今後に向けて

今後の課題は，メッセージ長の拡大である．しかし，メッセージ長が拡大すると，作成される検査行列の構造が大きく変化するため，メッセージ長 2 の場合に得られた構造的特徴をそのまま流用することは難しい．そこで，節 3.2 の手法を用いて，8×11 行列を作成し，復号性能を評価した．数値計算の結果，最も性能の良い検査行列は，図 9 のようなグラフの形をしていた．図 9 に示すように，全体的に冗長性が分散した構造を持っていた．このことから，メッ

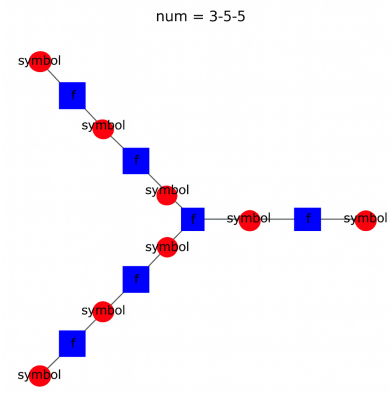

図 7　6×8 行列で最も性能が良いグラフ

セージ長を拡大させた場合においても，「全体的な冗長性の分散」が高性能な検査行列を構成する上で重要な要素であると考えられる．

一方，文献 [3] より，長さの短いループは検査行列の復号性能を著しく劣化させるが，長いループであれば性能に及ぼす影響が小さいことが知られている．実際に，一般に用いられている高い訂正能力を持つ検査行列も 2 部グラフに変換した際，ループを有している．そこで，本研究で発見された高性能なグラフ同士を連結し，大きなループを持つ 1 つのグラフを作成した場合，その検査行列も高い性能を示す可能性があると考えた．今後の研究として，全域木で表される高性能な検査行列の 2 部グラフを連結して新たなグラフを作成し，その復号性能を評価することが挙げられる．

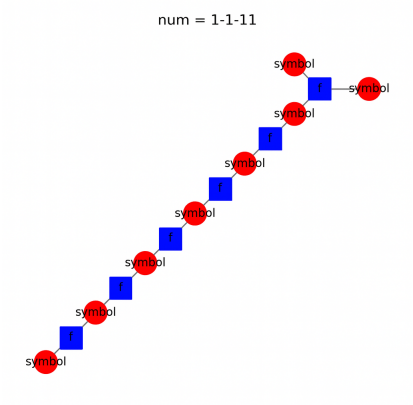

図 8　6 × 8 行列で最も性能が悪いグラフ

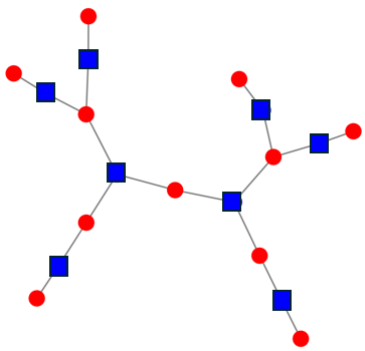

図 9　8 × 11 行列で最も性能が良いグラフ

5　まとめと今後の課題

本研究では，sum-product 復号法による厳密な周辺化計算を重視し，あえてループを排除した条件下において，検査行列の構造的な違いが復号特性に与える影響に着目した．そのため，全域木で表される検査行列について sum-product 復号法を用いた数値計算により復号性能を評価した．メッセージ長 2 の場合，検査行列を 3 種類に分類して考察した結果，性能の良い検査行列に共通する構造上の特徴を明らかにすることができた．特に，グラフ全体に冗長性が分散している構造（「全体的に冗長」な構造）が復号性能の向上に寄与することが分かった．今後は，より大きなメッセージ長に対して同様の解析を行い，高性能な検査行列の構造的特徴をさらに解明したい．そして，任意のメッセージ長でも高い復号性能を示す検査行列を作成する手法を確立したい．

謝辞

本研究を行うにあたって，丁寧なご指導を賜りました指導教員の西新幹彦准教授に深く感謝申し上げます．また，数多くのご意見をしてくださった西新研究室の方々に深く感謝申し上げます．

参考文献

[1] 和田山正，低密度パリティ検査符号とその復号法，トリケップス， 2002 年.

[2] 内川浩典,「低密度パリティ検査符号（LDPC 符号）-Robert G. Gallager 先生の 2020 年日本国際受賞に寄せて」, 電子情報通信学会　基礎・境界ソサイエティ Fundametals Review, 14 巻，3 号，pp.217–228，2021 年.

[3] 和田山正，誤り訂正技術の基礎，森北出版，2010 年.

[4] 飯島一貴,「パケット間隔で情報を送る通信路に対する誤り訂正符号の構成に関する考察」, 信州大学大学院理工学系研究科修士論文（指導教員：西新幹彦），2017 年 3 月.

[5] 塚田芳寿,「誤り訂正符号としてのラテン方陣の復号方法について」，信州大学大学院総合理工学研究科修士論文（指導教員：西新幹彦），2021 年 3 月.

[6] 南澤航,「制約付き乱数に基づく情報源符号化の性能検証に向けて」，信州大学大学院総合理工学研究科修士論文（指導教員：西新幹彦），2025 年 2 月.

[7] C 言語による乱数生成,
`https://omitakahiro.github.io/random/random_variables_generation.html`,
2025 年 12 月閲覧.

付録 A　ソースコード

A.1　全域木を作成するプログラム

文献 [6] を参考にして，プログラムを作成した．

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import networkx.algorithms.isomorphism as iso
import copy
from datetime import datetime

print(datetime.now())

ROW = 11
COLUMN = 13

all_edges = []

for i in range(COLUMN):       #頂点がすべてつながっている検査行列を作成
    for j in range (COLUMN, ROW + COLUMN, 1):
        all_edges.append((i,j))



#print(all_edges)

kouho_edges = []

kouho_edges.append([all_edges[0]])



#print(kouho_edges)

#print(len(kouho_edges))
#print(len(all_edges))
#print(kouho_edges[0][0][1])

count = 0

new_kouho_edges = []
new_kouho_edges.append([all_edges[0]])

graph_list = []


G = nx.Graph()
G.add_edge(all_edges[0][0], all_edges[0][1])
G.add_node(all_edges[0][0])
G.add_node(all_edges[0][1])

G.nodes[all_edges[0][0]]['parity'] = 'symbol'
G.nodes[all_edges[0][1]]['parity'] = 'f'
graph_list.append(G)


while count < ROW + COLUMN - 2:
    kouho_edges = new_kouho_edges
    print("count = ", count)
    new_kouho_edges = []
    tuika_suruyatu = []
    new_graph_list = []
    hash_list = []
```

```
    for n in range(len(kouho_edges)):
        eraban = []
        tuika_suruyatu.append([])

        for l in range(2):  #○を追加するか□を追加するか
            check = ROW + COLUMN  #あり得ない数字
            kouho_edges[n].sort(key = lambda x: x[l]) #小さい順に

            for i in range(len(kouho_edges[n])):
                eraban.append(kouho_edges[n][i][not l]) #すでに追加してあるものを抽出

            eraban_list = list(set(eraban))  #同一のものを削除

            for i in range(len(kouho_edges[n])):
                if check != kouho_edges[n][i][l]:  #同一のノードの追加無し
                    for j in range(len(all_edges)):
                        if kouho_edges[n][i][l] == all_edges[j][l]  and (all_edges[j][not l] in eraban_list) == False:
                            tounyu = copy.deepcopy(kouho_edges[n])
                            #print("tou", tounyu)
                            #print("all", all_edges[j])
                            tounyu.append(all_edges[j])
                            new_kouho_edges.append(tounyu)
                            #print(new_kouho_edges)
                            #print("koukooou", kouho_edges)
                            check = kouho_edges[n][i][l]
                            #tuika_suruyatu[n].append((all_edges[j][not l] ,all_edges[j]))

                            gura = copy.deepcopy(graph_list[n])
                            gura.add_edge(all_edges[j][0], all_edges[j][1])
                            gura.add_node(all_edges[j][not l])

                            if all_edges[j][not l] < COLUMN:
                                gura.nodes[all_edges[j][not l]]['parity'] = 'symbol'
                                #print("pow")

                            else:
                                gura.nodes[all_edges[j][not l]]['parity'] = 'f'
                                #print("piw")

                            new_graph_list.append(gura)
                            hash_list.append(nx.weisfeiler_lehman_graph_hash(gura, node_attr = "parity"))
                            break

                    nm = iso.categorical_node_match("parity", "f")

                    #print(len(new_graph_list))
                    if len(new_graph_list) > 1:
                        for j in reversed(range(0, len(new_graph_list) - 1, 1)):
                            #print("ooooooooooooooooo")
                            if (hash_list[-1] == hash_list[j]):
                                if (nx.is_isomorphic(new_graph_list[-1], new_graph_list[j], node_match = nm) == True):
                                    del new_graph_list[-1]
                                    del new_kouho_edges[-1]
                                    del hash_list[-1]
                                    break

    graph_list = copy.deepcopy(new_graph_list)
    count += 1
    #print("honto",len(new_kouho_edges))
    #print(new_kouho_edges)
    #print("honto",len(new_kouho_edges))
#print("del",del_list)
#print(new_kouho_edges)
print(len(new_kouho_edges))
print(datetime.now())

"""
for i in range(len(new_kouho_edges)):
    print(new_kouho_edges[i])
```

```
"""

del_list = []


for i in range(len(new_kouho_edges)):
    check_pari = np.zeros((ROW, COLUMN))
    for j in range(len(new_kouho_edges[i])):
        check_pari[new_kouho_edges[i][j][1] - COLUMN][new_kouho_edges[i][j][0]] = 1

    if ROW != np.linalg.matrix_rank(check_pari):
        del_list.append(i)

a = 0

for i in del_list:
    del new_kouho_edges[i - a]
    a += 1


f = open('tree3.txt', 'w')
f.write(str(COLUMN))
f.write("\t")
f.write(str(ROW))
f.write("\t")
f.write(str(len(new_kouho_edges)))
f.write("\n")
f.close()

f = open('tree3.txt', 'a')
for i in range(len(new_kouho_edges)):
    for j in range(len(new_kouho_edges[i])):
        for l in range(2):
            f.write(str(new_kouho_edges[i][j][l]))
            f.write("\t")
        f.write("\n")
f.close()
```

A.2 メッセージ長 2 の場合の全域木を作成するプログラム

```
import matplotlib.pyplot as plt
import networkx as nx
import math

# ノード数設定
symbol_count = 18  # 赤ノード (symbol)
f_count = 16       # 青ノード (f)
total_nodes = symbol_count + f_count

def get_valid_branch_configurations(total_nodes):
    """3 つの奇数の和が total_nodes になる構成を列挙"""
    configs = set()
    for a in range(1, total_nodes, 2):
        for b in range(1, total_nodes, 2):
            c = total_nodes - a - b
            if c >= 1 and c % 2 == 1:
                config = tuple(sorted([a, b, c]))
                configs.add(config)
    return [list(config) for config in sorted(configs)]

def generate_fixed_graph(branch_lengths):
    """
    赤ノード: 0〜symbol_count-1
    青ノード: symbol_count〜symbol_count+f_count-1
    中心ノードは青ノードの先頭 (symbol_count) とする
```

```
    """
    G = nx.Graph()
    center = symbol_count  # 青ノードの先頭が中心

    G.add_node(center, parity='f')

    red_nodes = list(range(0, symbol_count))
    blue_nodes = list(range(symbol_count, total_nodes))

    red_idx = 0
    blue_idx = 1

    edge_list = []

    for branch_len in branch_lengths:
        prev = center
        for i in range(branch_len):
            if i % 2 == 0:
                # 赤ノード
                if red_idx >= len(red_nodes):
                    raise IndexError("赤ノード数が足りません")
                node = red_nodes[red_idx]
                red_idx += 1
                parity = 'symbol'
            else:
                # 青ノード
                if blue_idx >= len(blue_nodes):
                    raise IndexError("青ノード数が足りません")
                node = blue_nodes[blue_idx]
                blue_idx += 1
                parity = 'f'

            G.add_node(node, parity=parity)
            G.add_edge(prev, node)

            #出力順序を常に「赤ノード 青ノード」に固定
            if parity == 'symbol':
                edge_list.append((node, prev))  # node = red, prev = blue
            else:
                edge_list.append((prev, node))  # prev = red, node = blue

            prev = node

    return G, edge_list

def layout_radial(G, branch_lengths):
    """枝を 120 度ずつ回転させてレイアウト"""
    pos = {}
    center = symbol_count
    pos[center] = (0, 0)

    angle_step = 2 * math.pi / 3
    branch_id = 0

    node_ids = [n for n in G.nodes if n != center]
    idx_pointer = 0

    for branch_len in branch_lengths:
        angle = angle_step * branch_id
        for i in range(branch_len):
            r = (i + 1) * 1.5
            x = r * math.cos(angle)
            y = r * math.sin(angle)
            pos[node_ids[idx_pointer]] = (x, y)
            idx_pointer += 1
        branch_id += 1

    return pos
```

```
def draw_and_save_all(symbol_count, f_count,
                      output_image="all_branch_graphs.png",
                      output_edge="all_branch_edges.txt"):

    usable_nodes = symbol_count + f_count - 1
    configs = get_valid_branch_configurations(usable_nodes)
    num_graphs = len(configs)

    cols = 5
    rows = (num_graphs + cols - 1) // cols

    fig, axes = plt.subplots(rows, cols, figsize=(cols * 4, rows * 4))
    axes = axes.flatten()

    all_edges = []

    for ax, config in zip(axes, configs):
        try:
            G, edges = generate_fixed_graph(config)
        except IndexError as e:
            print(f"スキップ: {e} → 枝構成 {config}")
            ax.axis('off')
            continue

        pos = layout_radial(G, config)
        node_colors = ['red' if G.nodes[n]['parity'] == 'symbol' else 'blue' for n in G.nodes]

        nx.draw(G, pos, ax=ax, node_color=node_colors, node_size=250, edge_color='gray', with_labels=False)
        nx.draw_networkx_labels(G, pos=pos,
                                labels={n: G.nodes[n]['parity'][0].upper() for n in G.nodes},
                                font_size=8, ax=ax)
        ax.set_title(f"num = {'-'.join(map(str, config))}", fontsize=10)
        ax.axis('off')

        all_edges.extend(edges)

    for ax in axes[num_graphs:]:
        ax.axis('off')

    plt.tight_layout()
    plt.savefig(output_image, dpi=300)
    plt.close()
    print(f"グラフ画像を保存しました: {output_image}")

    #エッジファイル書き込み（赤 青 の順）
    with open(output_edge, "w") as f:
        f.write(f"{symbol_count} {f_count} {num_graphs}\n")
        for u, v in all_edges:
            f.write(f"{u}\t{v}\n")
    print(f" エッジ情報を保存しました: {output_edge}")
if __name__ == "__main__":
    draw_and_save_all(
        symbol_count,
        f_count,
        output_image="all_branch_graphs.png",
        output_edge="all_branch_edges.txt"
    )
```

A.3 sum-product 復号法を用いたデコーダで復号誤り確率を数値計算によって求めるプログラム

文献 [4,7] を参考にして，プログラムを作成した．

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "MT.h"
#include <stdbool.h>
#define ROW 16    //行
#define COLUMN 18 //列
#define LOOP_SUM_PRODUCT 19 /*sum-product 復号の反復回数設定 */
#define FLIP 0.2
#define KAISU 524288  /*2 の LOOP_SUM_PRODUCT 乗*/

double rfx[2][ROW][COLUMN];
double qxf[2][ROW][COLUMN];
double g[2];
double cc[2];
double ss[2];
double gg[2];

int Hon;  //エッジの数


double w[2][2] = { 0.8, 0.2, 0.2, 0.8 };


int parity[ROW][COLUMN];
int generator[COLUMN][COLUMN - ROW];

void decoder(int y[], int x_hat[])
{
    int i, j, n;
    int k, m, l, q;
    int length;
    int sum;
    int max1;
    //double max2;
    //int likely_word;
    double check_sum[2];
    int index[COLUMN];
    double normal_const;
    //int x_hat[COLUMN];



    //検査行列の 1 がある部分に初期値を入れる
    for (i = 0; i < ROW; i++) {
        for (j = 0; j < COLUMN; j++) {
            for (l = 0; l < 2; l++) {
                rfx[l][i][j] = 0.0;
                if (parity[i][j] == 0) {
                    qxf[l][i][j] = 0;
                    //printf("%f",qxf[l][i][j]);
                }
                else {
                    qxf[l][i][j] = 1.0 / 2.0;
                    //printf("%f",qxf[l][i][j]);
                    //printf("%d,   %d\n", i ,j);
                }
                //printf("%f", qxf[l][i][j]);
            }
        }

    }

    for (int loop = 0; loop < LOOP_SUM_PRODUCT; loop++) {
        //r の更新 (M f → x )
        for (i = 0; i < ROW; i++) {
            for (j = 0; j < COLUMN; j++) {
                //検査行列が 1 であるところを探す (列ごと)
                if (parity[i][j] == 0) continue;
```

```
        for (l = 0; l < 2; l++) {
            check_sum[l] = 0.0;
        }


        length = 0;

        //index には列中 1 の位置を格納（今見ている 1 以外）
        //length は列中の 1 の個数
        for (q = 0; q < COLUMN; q++) {
            if ((q == j) || (parity[i][q] == 0)) continue;
            index[length] = q;
            length++;
        }

        //繰返し回数を決める
        max1 = 1 << length;  //全通りの回数の設定　ビットシフト
        //printf("%d\n", max1);

        for (m = 0; m < max1; m++) {
            for (l = 0; l < 2; l++) {
                ss[l] = 1.0;
            }
            for (l = 0; l < 2; l++) {
                sum = 0;
                for (n = 0; n < length; n++) {
                    sum ^= (m >> n) & 1;
                }
                if ((sum ^ l) != 0) continue;
                for (n = 0; n < length; n++) {
                    ss[l]
                        *=
                            qxf[(m >> n) & 1][i][index[n]] *
                            w[y[index[n]]][(m >> n) & 1];

                        //printf("%f\n", ss[l]);
                }
                check_sum[l] += ss[l];
                //printf("%f", check_sum[l]);

            }

        }

        normal_const = 0.0;
        for (l = 0; l < 2; l++) {
            normal_const += check_sum[l];
        }
        //printf("%f", check_sum[l]);
        for (l = 0; l < 2; l++) {
            rfx[l][i][j] = check_sum[l] / normal_const;
            //printf("%f", rfx[l][i][j]);
        }

    }
}



for (j = 0; j < COLUMN; j++) {
    for (i = 0; i < ROW; i++) {
        if (parity[i][j] == 0) continue;
        for (l = 0; l < 2; l++) {
            cc[l] = 1.0;
        }

        for (k = 0; k < ROW; k++) {
            if ((k == i) || (parity[k][j] == 0)) continue;
```

```
                    for (l = 0; l < 2; l++) {
                        cc[l] *= rfx[l][k][j];
                    }
                }

                normal_const = 0.0;

                for (l = 0; l < 2; l++) {
                    normal_const += cc[l];
                }

                for (l = 0; l < 2; l++) {
                    qxf[l][i][j] = cc[l] / normal_const;
                }
            }
        }

        for (j = 0; j < COLUMN; j++) {
            for (l = 0; l < 2; l++) {
                gg[l] = 1.0;
                for (i = 0; i < ROW; i++) {
                    if (parity[i][j] == 0) continue;
                    gg[l] *= rfx[l][i][j];
                }
                gg[l] *= w[y[j]][l];
            }

            normal_const = 0.0;

            for (l = 0; l < 2; l++) {
                normal_const += gg[l];
            }


            for (l = 0; l < 2; l++) {
                g[l] = gg[l] / normal_const;
                //printf("%f\n", g[l]);
            }

            //if (g[0] == g[1]) {
                //printf("[%d]\n", __LINE__);
            //}
            //x_hat[j] = (g[0] > g[1]) ? 0 : 1;
            //x_hat[j] = (g[0] > g[1]) ? 0 : ((g[0] == g[1]) ? 3 : 1);
            //printf("%d\n", x_hat[j]);
            //putchar('\n');

            if (fabs(g[0] - 0.5) <= 0.000000000001) {
                x_hat[j] = 3;
            }
            else if (g[0] > g[1]) {
                x_hat[j] = 0;
            }
            else {
                x_hat[j] = 1;
            }

        }


    }
    return;
}

double check(int x[], int x_hat[])
{
    double p = 1;
    for (int i = 0; i < COLUMN; i++) {
        if (x_hat[i] == 3) {
```

```
            p = p / 2;
        }
        else if (x[i] != x_hat[i]) {
            p = 0;
            break;
        }
    }
    return p;

}

double decoding_error()
{
    int x[COLUMN];   //符号語 all0
    int y[COLUMN];   //受信語 全通り
    int x_hat[COLUMN];

    double sum = 0;   //確率格納


    for (int i = 0; i < COLUMN; i++) {
        x[i] = 0;
    }

    for (int num = 0; num < 1 << (COLUMN); num++) {  //受信語を全通り作成する  2 の (COLUMN 乗)
        int temp = num;

        for (int i = COLUMN - 1; i >= 0; i--) {
            y[i] = temp % 2;
            temp /= 2;
        }

        decoder(y, x_hat);

        int third = 0;
        for (int i = 0; i < COLUMN; i++) {
            if (x_hat[i] == 3) {
                third = 3;
                break;
            }
        }
        if (memcmp(x, x_hat, COLUMN * sizeof(int)) == 0 || third == 3) {
            double pro = 1;
            for (int i = 0; i < COLUMN; i++) {
                if (x[i] != y[i]) {
                    pro *= w[0][1];
                }
                else {
                    pro *= w[0][0];
                }
            }
            sum += pro * check(x, x_hat);
        }
    }

    double ans = 1 - sum;    //符号語 all0 のときの復号誤り確率

    return ans;
}

int main(void) {
    int i, j;
    int row, column;
    int pari;
    double num[100000];
    int generator[COLUMN][COLUMN - ROW];

    init_genrand(10);
```

```
    for (i = 0; i < ROW; i++) {
        for (j = 0; j < COLUMN; j++) {
            parity[i][j] = 0;
        }
    }

    FILE* fp;

    // 読み込みモードでファイルを開く
    if ((fp = fopen("sample.txt", "r")) == NULL) {
        printf("\a no open[%d]\n", __LINE__);
        exit(1);
    }

    (void)fscanf(fp, "%d %d %d", &column, &row, &pari);

    Hon = row + column - 1;

    for (i = 0; i < pari; i++) {
        for (int i0 = 0; i0 < ROW; i0++) {
            for (int j0 = 0; j0 < COLUMN; j0++) {
                parity[i0][j0] = 0;
            }
        }
        for (j = 0; j < Hon; j++) {
            (void)fscanf(fp, "%d %d", &column, &row);
            parity[row - COLUMN][column] = 1;
        }
        //encoder(parity, generator);
        num[i] = decoding_error();
        //printf("%f\n", num[i]);
    }

    //for (int i = 0; i < ROW; i++) {   作成した行列の出力確認
        //for (int j = 0; j < COLUMN; j++) {
            //printf("%d\n", parity[i][j]);
        //}
    //}

    FILE* kekka;
    if ((kekka = fopen("kekka.txt", "w")) == NULL) {
        printf("\a no open [%d]\n", __LINE__);
        exit(1);
    }
    for (i = 0; i < pari; i++) {
        fprintf(kekka, "%f\t", num[i]);
        fprintf(kekka, "\n");
    }
    //printf("error: %g\n", error);

    fclose(kekka);
    fclose(fp);


    return 0;

}
```