# ベッチ数スペクトルによる<br>複雑ネットワークの解析

神山 直也

平成20年2月26日

# 目次

# 図 目 次

# 表 目 次

# 第1章　序章

## 1.1　はじめに

従来、多くの分野において物質や現象を理解する場合は、よりミクロな視点で理解することで、それから構成される物質や現象はボトムアップ式に理解できると考えられていた。しかし近年ではマクロな視点を抜きにしては物質や現象の働きは解明し難いと認知されている。現在、マクロな視点を得るために、大局的かつ様々な対象に適用できる汎用的な解析手法が求められており、大局的かつ汎用的な解析を可能にするモデルとして、ノードとリンクからなるネットワークモデルが存在する。現実をモデル化した複雑なネットワークの性質を一般的に解析することができれば、多くの不可解な現象を解明できると期待される。

具体的に見ていくと、我々の身の回りの多くのモノや概念について、ネットワークモデルを定義することが出来る。幾つか例を挙げると

- 空港と航路からなる航路ネットワーク
- 論文と引用関係かななる論文引用ネットワーク
- 人と友好関係からなる友人ネットワーク (図 1.1)
- 化学物質と化学反応からなる代謝ネットワーク

といったものがある。昨今の研究で、様々な異なる実体や現象から抽象化されたネットワークを解析すると、多くの共通した特徴を持つ事が分かっている。これら特徴は、あらゆる分野のネットワークについて確認されている。そこで、任意の複雑ネットワークについて有効に解析する手法が存在すれば、他分野において有効な解析手法となり得る事が分かる。

## 1.2　複雑ネットワークの特徴

ここで、多くの複雑ネットワークが持つ特徴についていくつか例を挙げると、

- スケールフリー性
- スモールワールド性
- クラスター性

と言った性質がある。

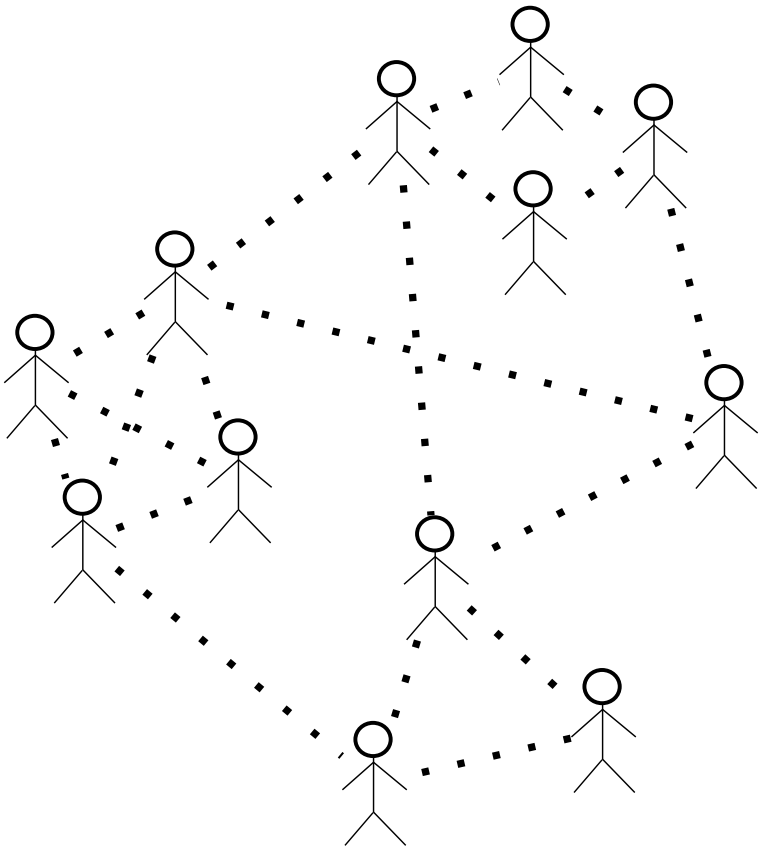

図 1.1: 友好関係

### 1.2.1 スケールフリー性

WWW(World Wide Web) を例に挙げると、個人サイトのような少数のリンクを持つサイトが多数を占める一方、Yahoo、Google と言った検索エンジン等ごく一部のサイトは非常に多くの HP とリンクされていがること分かる。

WWW の例に示すような、大多数が少数のリンクを持つ一方、ごく一部が非常に多くのリンクを持つといった現象は多くのネットワークにおいて見られる。この性質をスケールフリーという [1]。この様子を横軸にリンクの数（次数)、縦軸に出現頻度を取った次数分布で示すと図 1.2 のようになる。

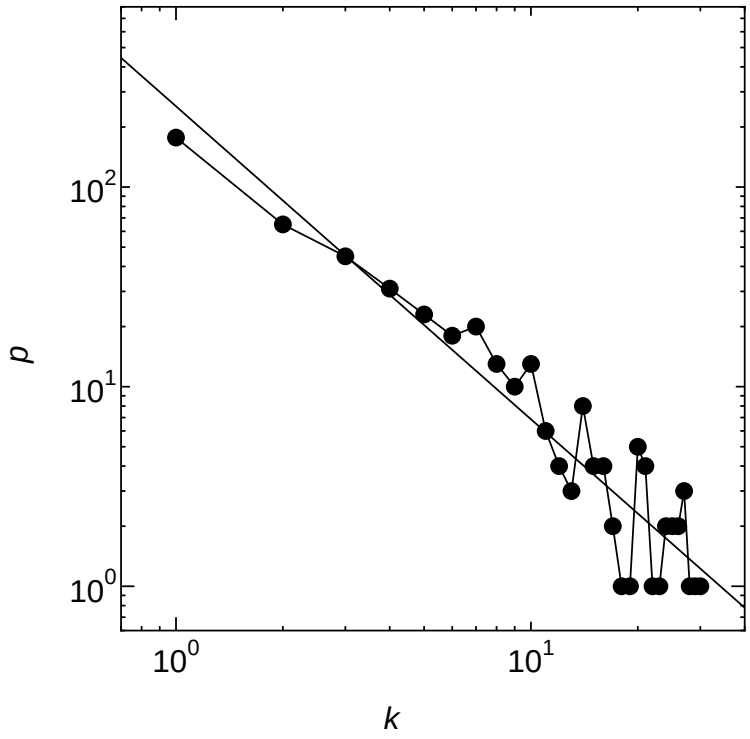


図 1.2: スケールフリーを示す次数分布

このとき、この次数分布は

$$P(k) \propto k^{-\gamma} \tag{1.1}$$

と表され、$\gamma$ をベキ指数と呼ぶ。図 1.2 のベキ指数は $\gamma = 1.6$ である。表 1.1 に実世界ネットワークのベキ指数の例を示す [2]。

### 1.2.2 スモールワールド性

人間関係を例に挙げると、知り合い同士を辿ることで、世界中の誰とでも数人を経由するだけで繋がっているという性質である。実際に社会心理学者スタンレー・ミルグラムが 1967 年にスモールワールド実験を行っている。この実験はアメリカ

表 1.1: 実世界ネットワークのベキ指数

| ネットワーク | ベキ指数 |
|---|---|
| WWW( World Wide Web ) | 1.9-2.7 |
| インターネット | 2.1-2.5 |
| 映画俳優の競演ネットワーク | 2.3-3.1 |
| 性的関係のネットワーク | 3.2-3.4 |
| タンパク質の反応ネットワーク | 2.4-2.5 |

国内に限定した実験であり、追試にこそ失敗しているが、スモールワールド性については実証されたと考えられている。スモールワールド性を示す指標として、ネットワークの直径や平均最短経路長といった指標がある。ここでは、平均最短経路長 $L$ について解説する。

まずノード間の距離はノード ( $i, j$ ) 間の最短経路の長さ $l_{ij}$（経由するリンク数）と定義される。平均最短経路長 $L$ はすべてのノードの組み合わせを考えたときの平均の距離であり、ノード数を $n$ とした場合

$$L = \frac{1}{{}_nC_2} \sum_{i,j} l_{ij} \quad (i \neq j) \tag{1.2}$$

となる。また、ネットワークの直径とは、すべてのノードの組み合わせを考えたときの最大の距離である。

### 1.2.3 クラスター性

ランダムグラフにおいてすべてのリンクはランダムに生成されるので、リンク数 $\gg$ ノード数 でなければ、二つのリンク先同士もまたリンクされている確率は非常に低い。しかし現実には、友人 A と友人 B が居た場合に、友人 A と友人 B もまた友人関係である確率は高い。このように、現実のネットワークにおいて、三角形の関係が多く含まれている事が多々あり、この性質をクラスター性という。

クラスター性を示す指標として、クラスター係数 $C$ がある。まず、ノード毎のクラスター係数を $C_i$ とし、次数（接続されているリンク数）を $k_i$ とすると、

$$C_i = \frac{p_i}{{}_kC_2} \tag{1.3}$$

となる。ここで、$p_i$ はリンクでつながれた ${}_kC_2$ 個の頂点のうち任意の二つと、頂点 $i$ で構成される三角形の個数である。

次に，ノード数を $n$ とすると、全体のクラスター係数 $C$ は、

$$C = \frac{1}{n} \sum_{i}^{n} C_i \tag{1.4}$$

で与えられる。

## 1.3 ネットワークの種類

## 1.4 ネットワーク生成モデル

### 1.4.1 ER モデル

ランダムグラフを生成することが出来るモデルである。

頂点数を $N$、各頂点間の接続確率を $p$ とする。リンクの生成判定は、${}_NC_2 = n(n-1)/2$ 個すべての頂点について独立に行う。これにより、ランダムネットワークが生成される。またこのとき、ある頂点の次数が $k$ になる確率は 2 項分布に従い、

$$p(k) =_{N-1} C_k p^k (1-p)^{N-k-1} \tag{1.5}$$

となる。また、$n \to \infty$ において、

$$p(k) \cong \frac{e^{-\lambda}\lambda^k}{k!} \tag{1.6}$$

とポアソン分布に近似される。

### 1.4.2 BA モデル

スケールフリーネットワークを生成することが出来るモデルであり、重要な性質として以下の要素が挙げられる。

- 成長するネットワーク
- 優先的選択

BA モデルとは、時間経過と共にノードを追加する（成長する）というダイナミクスを考慮したモデルである。また、ノードが追加される際に、適当なノードをリンクを構成するが、その時に、次数の高いノードと優先的にリンクされる。つまり、次数の高いノードはさらに次数が高くなるということである。ここで、非常に大きな次数を持つノードの事をハブと呼ぶ。

具体的な式を例にあげる。先ず、初期状態として $m_0$ 個のノードからなる完全グラフ $K_{m_0}$ を考える。次に頂点を一つずつネットワークに追加をしていく。ここで、頂点が $n$ 個とし、ある頂点 $v_{\mathrm{i}}(1 \leq i \leq n)$ の次数を $k_{\mathrm{i}}$ とする。その時、新しく追加された頂点と $v_{\mathrm{i}}$ がリンクされる確率を、

$$\Pi(k_{\mathrm{i}}) = \frac{k_{\mathrm{i}}}{\sum_{\mathrm{j}}(k_{\mathrm{j}})} \tag{1.7}$$

とする。

以上のルールを繰り返す事で、ネットワークがベキ乗則 $p(k) \propto k^{-3}$ を示すことが分かっている。

### 1.4.3 WS モデル

ワッツとストロガッツにより提案されたグラフ生成モデル。このモデルを用いることで，小さい平均最短経路長 $L$、大きいクラスター係数 $C$ を持つネットワークを得ることが出来る。

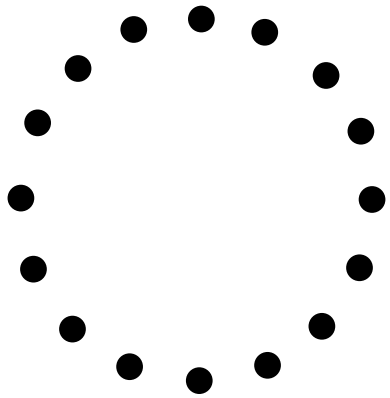

図 1.3: CW モデル-生成過程 1

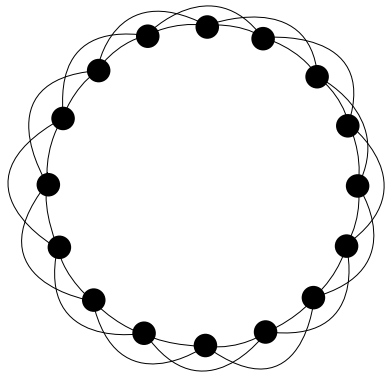

図 1.4: CW モデル-生成過程 2 ( $k = 2$ )

図 1.3 のように、$n$ 個のノードを円周上に配置する。次に、図 1.4 のように隣り合う $k$ 点とリンクを繋ぐ。そして、すべてのリンクについて、確率 $p$ でランダムに繋ぎ換えを行うと図 1.5 のようになる。$p = 1$ の時、得られるグラフはランダムグラフとなる。適切な $p$ を与えることで、クラスター係数 $C$ の減少を最小限にしつつ、平均最短経路長 $L$ を小さくすることができる。

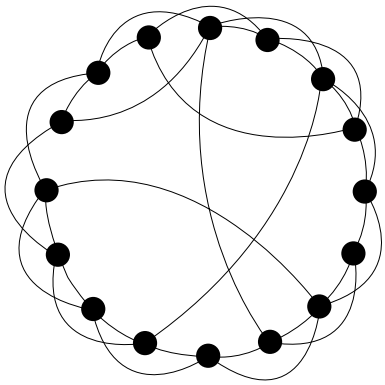

図 1.5: CW モデル-生成過程 2 ( $k = 2, p = 0.25$ )

## 1.5 メトロポリス法

### 1.5.1 概略

メトロポリス法とは、モンテカルロシミュレーションにおいて、乱数により提案された新しい状態を棄却するか採択するか判断する基準の一つである。系が状態変化したと仮定し、そのエネルギー変化を $\Delta E$ とする。このとき、確率

$$\begin{array}{ll} 1 & \Delta E \leq 0 \\ e^{-\frac{\Delta E}{k_{\mathrm{B}}T}} & \Delta E > 0 \end{array} \tag{1.8}$$

でその状態遷移を採択する手法である。ここで、$k_{\mathrm{B}}$ はボルツマン定数であり、$T$ は温度である。メトロポリス法は熱力学のボルツマン因子を一般的な系に適用したものであり、温度 $T$ のパラメータを持っている。このパラメータの意味は系によって変化する。

### 1.5.2 ネットワークに適用

ネットワークエネルギーを定義することで、メトロポリス法を用いてネットワークを変化させる事が出来る [3]。ネットワークエネルギーの定義等、シミュレーションにおける具体的な条件については 3.1.2 節に記述する。

## 1.6 トポロジー（位相幾何学）

本研究でネットワークの形状解析に用いているトポロジー理論について簡単に説明を行う。詳しくは文献 [4][5] に分かりやすい説明があるのでそれを参照されたい。

### 1.6.1 概要

トポロジーとは、柔らかい幾何学とも称され、位置と形相の幾何学を意味する。言い換えると、長さ・大きさといった量を考えず、相互の位置・繋がりのみを考慮する幾何学である。ここで例として図 1.6 を示す。位相幾何学においては、図 1.6 に示す 4 つの図形は等しくなる。

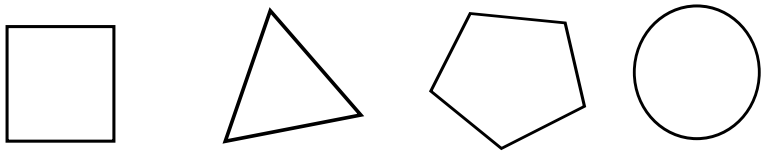

図 1.6: 位相的に等しい図形

### 1.6.2 ホモロジー群とベッチ数

#### 単体

単体とは、線分・三角形・四面体の概念を拡張したものである。

$v_0, v_1, \cdots, v_n$ を $n$ 次元空間に存在する $n+1$ 個の点とする。ここで、$n$ 個のベクトル $\overrightarrow{v_0v_1}, \overrightarrow{v_0v_2}, \cdots, \overrightarrow{v_0v_n}$ が一次独立であるとき、これを頂点 $v_i(0 \leq i \leq n)$ からなる $n$ 次元単体と言い、

$$\delta^n = (v_0, v_1, \cdots, v_n) \tag{1.9}$$

と表す。例を図 1.7 に示す。

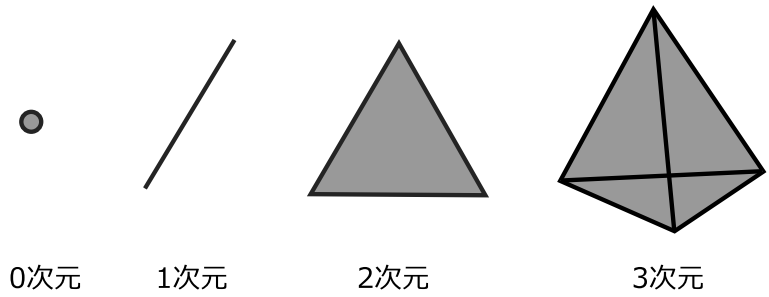


図 1.7: 単体

#### 面単体

$n$ 次元単体 $\delta$ の頂点から $m(1 \leq m \leq n+1)$ 点を選び、それらを頂点とする $m$ 次元単体 $\tau$ を考えることが出来る。このとき、この $\tau$ を面単体と言う。

### 単体的複体

前節で説明した単体を組み合わせて出来る図形であり、単体の集合 $K$ が以下の条件を満たすとき、$K$ を単体的複体という。

- ある単体 $\delta$ が $K$ に含まれるとき、$\delta$ のすべての面単体も $K$ に含まれる。
- $K$ に含まれる 2 つの単体 $\delta, \tau$ の共通部分 $\delta \cap \tau$ が空でない場合、それは $\delta$ の面単体且つ $\tau$ の面単体である。

### 鎖群

単体的複体 $K$ の $r$ 次元鎖群 ( $r$ -鎖群 ) を $C_r(K)$ と表す。$C_r(K)$ は以下のように定義される。

$$C_r(K) = \{c_r | c_r = \sum_{i=1}^{n} m_i \delta_i^r, m_i \in Z\} \tag{1.10}$$

ここで、$Z$ とは整数全体を示す。また、$m_1\delta_1^r, m_2\delta_2^r, \cdots, m_n\delta_n^r$ に対して $m_1, m_2, \cdots, m_n$ を対応させることにより、

$$C_r(K) \cong Z \oplus \cdots \oplus Z \tag{1.11}$$

となり、$C_r(K)$ の値は単体の個数によって決定する。

### 鎖複体

上記の複体を元に代数的構造を考えたものが鎖複体である。$\partial_r : C_r(K) \longrightarrow C_{r-1}(K)$ なる準同型写像を以下のように定める。

$$\partial_r := \sum_{i=0}^{r} (-1)^i (v_0 v_1 \cdots v_{i-1} v_{i+1} \cdots v_r) \tag{1.12}$$

これを、境界準同型写像と言う。

ここで、$K^n$ を $n$ -複体とすると、境界準同型写像 $\partial_r$ を用いて、

$$\begin{array}{ccccccccc} & & \partial_n & & \partial_{n-1} & & & \partial_1 & & \partial_0 & \\ 0 \to C_n(K) & \to & C_{n-1}(K) & \to & \cdots \to C_1(K) & \to & C_0(K) & \to & 0 \end{array} \tag{1.13}$$

という列が得られ、この列を $C(K)$ と表す。また、$C(K)$ を $K$ で決まる鎖複体と言う。この群はアーベル群であり、足し算及び引き算が自由に出来る代数的な群である。詳しくは群論の教科書を参考にされたい。

### ホモロジー群

$r$ 次元ホモロジー群は

$$\mathrm{H}_r(K) = \frac{\mathrm{Z}_r(K)}{\mathrm{B}_r(K)} \tag{1.14}$$

と表される。ここで、$\mathrm{Z}_r(K)$ とは $\partial_r$ の核であり、$r$ 次元輪体群と呼ばれ、 $\mathrm{B}_r(K)$ とは $\partial_{r+1}$ の像であり、$r$ 次元境界群と呼ばれる。また、$\mathrm{B}_r(K) \subset \mathrm{Z}_r(K)$ である。

ここで、二つの複体 $K_1, K_2$ が位相同型であるとする。このとき、任意の $r$ について

$$\mathrm{H}_r(K_1) \cong \mathrm{H}_r(K_2) \tag{1.15}$$

という関係が成り立つ。

具体的な計算については付録にて示す。

### ベッチ数

$\mathrm{H}_r(K)$ の階数を $r$ 次元ベッチ数といい、$\beta_r$ と表す。位相同型であればホモロジー群が同型となるので、ベッチ数もまた等しくなる。

このように、位相が変化しても、値が変化しない量を位相不変量という。

### 数値計算

ベッチ数に関して、

$$\beta_i = \dim \mathrm{Z}_i + \dim \mathrm{B}_i \tag{1.16}$$

が成り立つ。また以下も成り立つ。

$$\dim \mathrm{Z}_i + \dim \mathrm{B}_{i-1} = \dim C_i \tag{1.17}$$

さらに、

$$\dim \mathrm{B}_i = \mathrm{rank}(\partial_{i+1}) \tag{1.18}$$

である。以上をまとめると、

$$\beta_i = \dim C_i - \mathrm{rank}(\partial_i) - \mathrm{rank}(\partial_{i+1}) \tag{1.19}$$

となる。$\dim C_i$ とは $i$ 次元単体の個数であり、$\mathrm{rank}(\partial_i)$ とは写像を行列で表した時の階数である。

以上を用いることで、コンピュータを用いた数値解析を容易に行うことができる。

## 1.7 ベッチ数スペクトル

ある複体からは多次元のベッチ数を得ることができる。そこで、本研究では各次元のベッチ数成分を並べ、スペクトル表現したものをベッチ数スペクトルと称す。

## 1.8 並列コンピューティング

一つのタスク（処理) を複数のプロセッサを用いて行うことである。昨今のスーパーコンピュータは、非常に多くのプロセッサを並列に動かす事で高速な計算を可能としている。最近では、一般向けの CPU でも複数のコアを持っており、ますます並列コンピューティングの重要性が増してきている。

各プロセッサに分配されたタスクに実行順序等の制約があると、それらを並列実行することが難しくなる。故に、単純に並列化すれば速度が上がるという訳でもなく、逆に並列化処理のオーバーヘッドにより遅くなることさえある。よって、並列コンピューティングを行うためには、並列化の種類に応じて処理方法等を慎重に考えなければならない。主に並列化手法として以下のようなものがある。

- 分散メモリ並列
- 共有メモリ並列

これらについて説明を行っていく。

### 1.8.1 分散メモリ並列

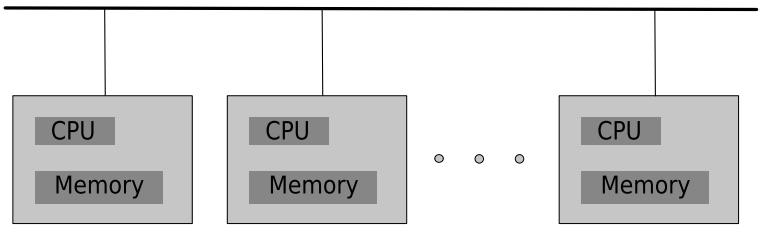


図 1.8: 分散メモリ並列

図 1.8 のように、CPU-メモリが一組になったものが、複数個接続された形態を分散メモリ並列という。基本的に、CPU は別の組のメモリを参照することはできず、全体を接続しているバスは（CPU-メモリ間の速度に比べて）低速である。故に、頻繁に情報をやりとりすると、バスの速度がボトルネックとなり十分な高速化が行われなくなる。しかし、CPU-メモリを追加していく事により容易に全体の処理能力を高めることが出来る点で後述する共有メモリ並列よりも優れている。また、大容量のメモリを使用できる点でも優れている。

#### MPI

MPI とは Message-Passing Interface の略であり、分散メモリ並列プログラミングでは一般的な API である。C/Fortran 向けであり、これを用いる事で容易かつ簡潔に分散メモリ並列による並列化を実装することが出来る。

### 1.8.2 共有メモリ並列

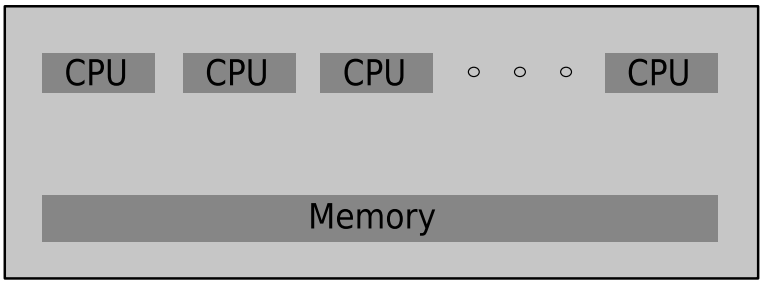


図 1.9: 共有メモリ並列

図 1.8 のように、複数の CPU が一つのメモリを共有する形態を共有メモリ並列という。情報の共有が容易であり、プログラミングは比較的簡単である。しかし、メモリアクセスの排他処理を正しく行わなければ処理結果の一貫性は保たれなくなる。CPU を増やしたりメモリを増量し規模を大きくすることで、有る程度までは処理能力が向上する。しかし、次第にメモリアクセスの排他処理のオーバーヘッドが大きくなり処理能力が向上しにくくなる。

#### OpenMP

共有メモリ並列プログラミングで一般的に用いられる API である。C/C++/Fortran で標準化されている。逐次プログラムに数行加えるだけで並列化ができるが、処理の並列性（同じメモリ領域を書き換えない）の判断はプログラムを組む人に任せられている。これを自動で行ったものが後述する自動並列化である。

#### 自動並列化

コンパイラによっては、処理の並列性判断を自動で行い、並列化を施すことができる。具体的には、以下に示すようなコードが自動並列化の対象となる。

```
int a[100];
int b[100];
int i;
for(i=0; i<100; ++i) b[i] = i * i;
for(i=0; i<100; ++i) a[i] = b[i];
```

単純な処理であれば十分に自動並列化の恩恵を受けることが出来るが、複雑なコードについては自動並列化は施されない可能性が高い。

## 1.9 目的

前述したベッチ数を用いることで、複雑なネットワークの形状・特徴を数値化することが可能となった。しかし、定義どおりに考えると、ネットワークは0次元単体（ノード）と1次元単体（リンク）しか持たないので、$\beta_0, \beta_1$ しか値が出てこない。これだけでは非常に複雑なネットワークの解析値としては物足りない。これまでの研究 [6][7] で、ネットワークを高次元の単体的複体と見なしベッチ数を解析することで有効な値が得られる事が分かっている。そこで、ベッチ数スペクトルを複雑ネットワーク解析の新しい指標として提案し、具体的に解析を行いその値を検証することで解析指標としての有効性を検討する。

# 第2章　解析プログラム

## 2.1　概要

ベッチ数スペクトルを計算するためには、式（1.19）を解く必要がある。この問題は、以下を求める事に帰着する。

- 各次元の単体数
- 境界準同型写像の行列及びその階数

本研究室で用いられていた解析プログラムでは、メモリ使用効率が優れず、大規模な解析が不可能であった。そこで、本研究では先ず解析プログラムの性能改善から試みた。

境界準同型写像行列の特徴として、零成分に対する非零成分の割合が非常に低いといった特徴を持っている。このような行列を疎行列という。この特徴を生かせば、省メモリ且つ高速化が実現できる [8]。従来の解析プログラムでは、境界準同型写像行列が二次元配列で実装されていた為にメモリ使用にかなりの無駄が生じていた。

疎行列の特徴を生かす為には、非零成分の偏りなどを考慮した処理が必要となる。今回の解析では行列の階数を知ることができればいい。そこで、疎行列の階数を求める汎用アルゴリズムを調査したが、本研究で用いる事ができそうなアルゴリズムを見つけることが出来なかったので、本研究では先ず疎行列を用いた階数算出プログラムを作成した。

## 2.2　疎行列処理

行の基本変形を行い、階段行列を生成する処理を行う。ただし、階数を求めることが出来ればいいので、完全な階段行列は生成しない。階段行列生成時に非零成分の個数が増加しメモリ使用量が増える。、処理の過程において行列中の不要な成分に割り当てられたメモリを随時解放することで、ある程度メモリ増加を抑えている。

例を表 2.1、表 2.2、表 2.3 に示す。まず、2.1 に示す行列を考える。表 2.1 は見やすさの為に零成分も記述しているが、実際には零成分はメモリ上に確保されていない。これは、配列とリスト構造を組み合わせることで実装されている。表 2.1 → 表 2.2 については一般的な行基本変形である。ここで、階段行列を生成する過程において、これ以降 1 行目の数値を参照／操作する事はない。つまり、記憶してお

表 2.1: 行列処理 step1

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 |

表 2.2: 行列処理 step2

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | -1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 |

表 2.3: 行列処理 step3

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | -1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 |

く必要が無いので、1 行目の情報をメモリ上から解放する。当然すべて 0 の行はメモリ上に領域を持たないために、メモリの解放は行われない。 次は 2 列目に着目し、次は 3 列目といったように処理を繰り返していくと最後は行列の要素がすべて解放される。この時、解放した行数がその行列の階数となる。

## 2.3 高速化

大規模なネットワークを計算するために、今回は処理の並列化を極力行うようにした。これにより、プロセッサ数 (コア数) に応じた処理速度の高速化を図る。これは、今後一般向けコンピュータにおいてマルチコア化が進んでいることを考慮すると、応用の観点からも有効であると考えられる。今回の計算内容を考えると、1.8 セクションで示す分散メモリ並列又は共有メモリ並列のどちらを用いても実現できると考えられる。今回は簡素化の為、共有メモリ並列を前提に並列化を行った。

## 2.4 現アルゴリズムの問題

今回のように、必要とするメモリ量が動的に変化する場合はリスト構造が非常に有効であり、この解析プログラムでも行列を表現する際にリスト構造を用いている。しかし、現在のコンピュータアーキテクチャーでは、リスト構造を多用すると処理速度の高速化が行いにくくなる。境界準同型写像行列は比較的規則正しい行列となるので、うまく特徴を生かせばリスト構造を用いることなく配列を用いて行列を表現でき、それにより高速化が出来ると思われる。

## 2.5 補足

クラスター係数算出プログラム、平均最短経路長算出プログラム等についても、並列化を施している。

これら解析プログラム群は C 言語を用いて実装した。極力環境依存を避け、VisualC++ / IntelCompiler (OpenMP) / GCC (OpenMP) / MinGW / FCC (OpenMP) / xlc (OpenMP) 等でコンパイル及び動作確認を行った。他にもコンパイル可能であることを確認したコンパイラはいくつか存在するが、出力された実行ファイルの動作は確認していない。

# 第3章　シミュレーション

本研究では、二つのシミュレーションを行い、それらについて解析を行う事で作成された解析プログラム及びベッチ数スペクトルの有効性についての検証を行う。

## 3.1　シミュレーション１

本シミュレーションでは、ランダムネットワークをメトロポリス法を用いて変化させる。そして、その過程に於いてベッチ数スペクトルの計算を行う。また、比較検証の為にいくつかの指標も計算する。得られた結果を検証することによりベッチ数スペクトルの有効性を示す。大まかな流れを図 3.1 に示す。

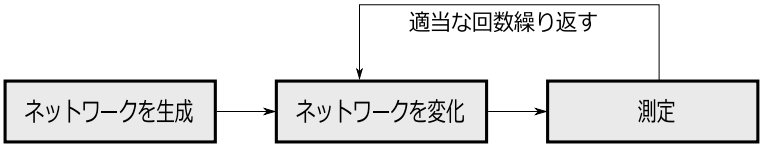


図 3.1: シミュレーションの流れ

### 3.1.1　ネットワークの生成

初期ネットワークをランダムネットワークとする。以下に主な条件を示す。

- ノード数を 1000 とする。
- リンク数を 3000 とする。
- メトロポリス法を用いて、ネットワークをスケールフリーネットワークに推移させるため [3] に、各ノードに指数分布に従い重みを与える
- 二つのノード間に、複数のリンクが張られることはない。

### 3.1.2　ネットワークの変化

1.5 セクションで紹介したメトロポリス法を用いてネットワークを変化させる [3]。以下でメトロポリス法を用いるにあたり必要なネットワークエネルギーの定義を行う。

#### ネットワークエネルギー

ノード $n_i, n_j$ の重みを $w_i, w_j$ とする。また、$n_i, n_j$ に接続されたリンクのエネルギー $L_{ij}$ を、

$$L_{ij} = -w_i w_j \tag{3.1}$$

と定義し、ネットワークエネルギー $E$ を、

$$E = \sum_{i,j} L_{ij} \tag{3.2}$$

とする。メトロポリス法においてはエネルギーの低い状態が安定するので、ネットワークエネルギーの定義より、重みの重いノードに接続されているリンクは安定し、重みの低いノードに接続されたリンクは安定しない。

#### 変化

以下の条件でリンクの繋ぎ替えを提案する。

- ノード数は一定
- リンク数は一定
- 1 本のリンクをつなぎ替える
- 二つのノード間に、複数のリンクが生じないように繋ぎ替え先を選択する

つなぎ変えた回数を時間 $t$ と表す。

これまでの研究 [3] で、パラメータ $T$ が低いほど、ネットワークがスケールフリーネットワークに変化していくと分かっている。逆にパラメータ $T$ が高いと、ネットワークはランダムネットワークに変化していくことが分かっている。

### 3.1.3 ネットワークの解析

リンクの繋ぎ替えを適当回数行う毎に、ネットワークを解析する。

## 3.2 シミュレーション２

多くの現象は、ネットワーク化した際にクラスター係数が大きくなる傾向にある。そこで、本シミュレーションでは、WS モデルを用いて高いクラスター係数数 $C$ を持つネットワークを生成し、それについてネットワーク解析を行った。

- ノード数 1000、リンク数 3000
- ノード数 1000、リンク数 12000
- ノード数 4000、リンク数 12000

以上について、辺繋ぎ替え確率 $p$ を、0.1〜0.5 で変化させ測定を行う。

## 3.3 解析手法

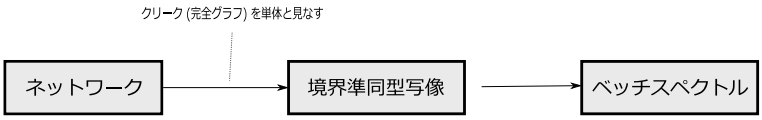


図 3.2: 解析の流れ

前述した通り、ネットワークそのものは 0, 1 次元の情報しか持たない。そこで、図 3.2 に示すとおり、クリーク（完全部分グラフ）を単体と見なす事で高次元解析を行っている。完全部分グラフとは、どのノードも他のすべてのノードとリンクされている部分グラフの事である。

式（1.19）をコンピュータを用い計算することでベッチ数の計算を行った。ここで、単体の最大次元を $c_{max}$ とすると、$i > c_{max}$ で $\beta_i = 0$ となるので、検出された単体の最大次元のベッチ数までで解析を打ち切っている。

また、比較のためにいくつかの指標についても解析を行う。本研究で解析対象とした指標は以下の通りである。

- 最短平均経路長
- 直径
- クラスター係数
- 次数分布

## 3.4 留意点

ベッチ数スペクトルは、高次元ベッチ数が値を持つ程の、非常に複雑なネットワークに対して有効であると考えられる。そこで、本研究では、解析対象のネットワークを比較的規模の大きい物に限定している。

# 第4章　結果及び検討

## 4.1　シミュレーション1

図 4.1 にネットワークエネルギーの変化を示す。横軸は辺の繋ぎ替え試行回数である。以後これを時間と称する。

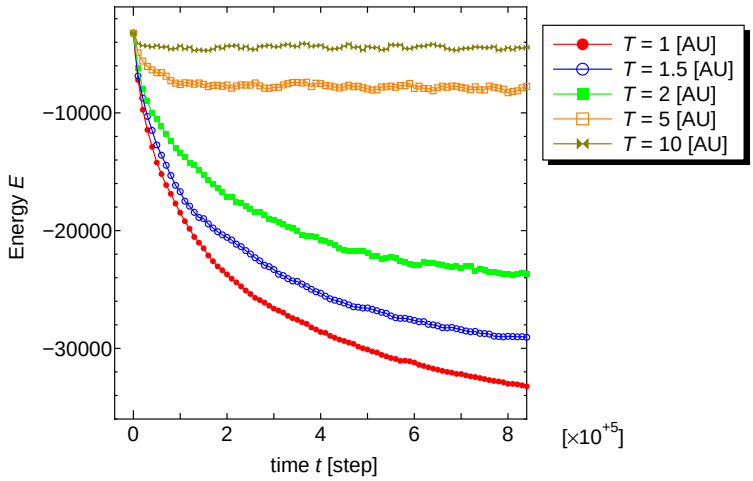


図 4.1: ネットワークエネルギー

温度 $T$ が低いほどエネルギー状態が低くなっている。初期ネットワークを図 4.2、各温度における $t = 800000$ のネットワークを図 4.3、4.4、4.5、4.6、4.7 に示す。また、各時点における次数分布を図 4.8 に示す。また次数分布を両対数グラフで表したものを図 4.9 に示す。尚、ネットワークの描画には pajek[9] を用いた。

温度が低いほど一部のノードにリンクが集中している様子が、ネットワーク図及び次数分布から読み取れる。次に、クラスター係数、平均最短経路長、直径を図 4.10、4.11、4.12 に示す。

図 4.10 より、温度 $T$ が低いほど高いクラスター係数 $C$ を持つことが読み取れる。これは、ネットワーク中に三角形形状が多く構成されている事を示している。つまり、温度が低いほど、ネットワーク中にノードの固まりが出来るという事である。

図 4.11 より、全体的には温度 $T$ が低いほど低い平均最短経路長 $L$ を持つことが読み取れる。局所的に見ると増減量のばらつきが多いと分かる。メトロポリス法は確率的に状態が決定するモデルで有ることを考えると、最短平均経路長がネットワークの微細な変化に対して敏感であると分かる。

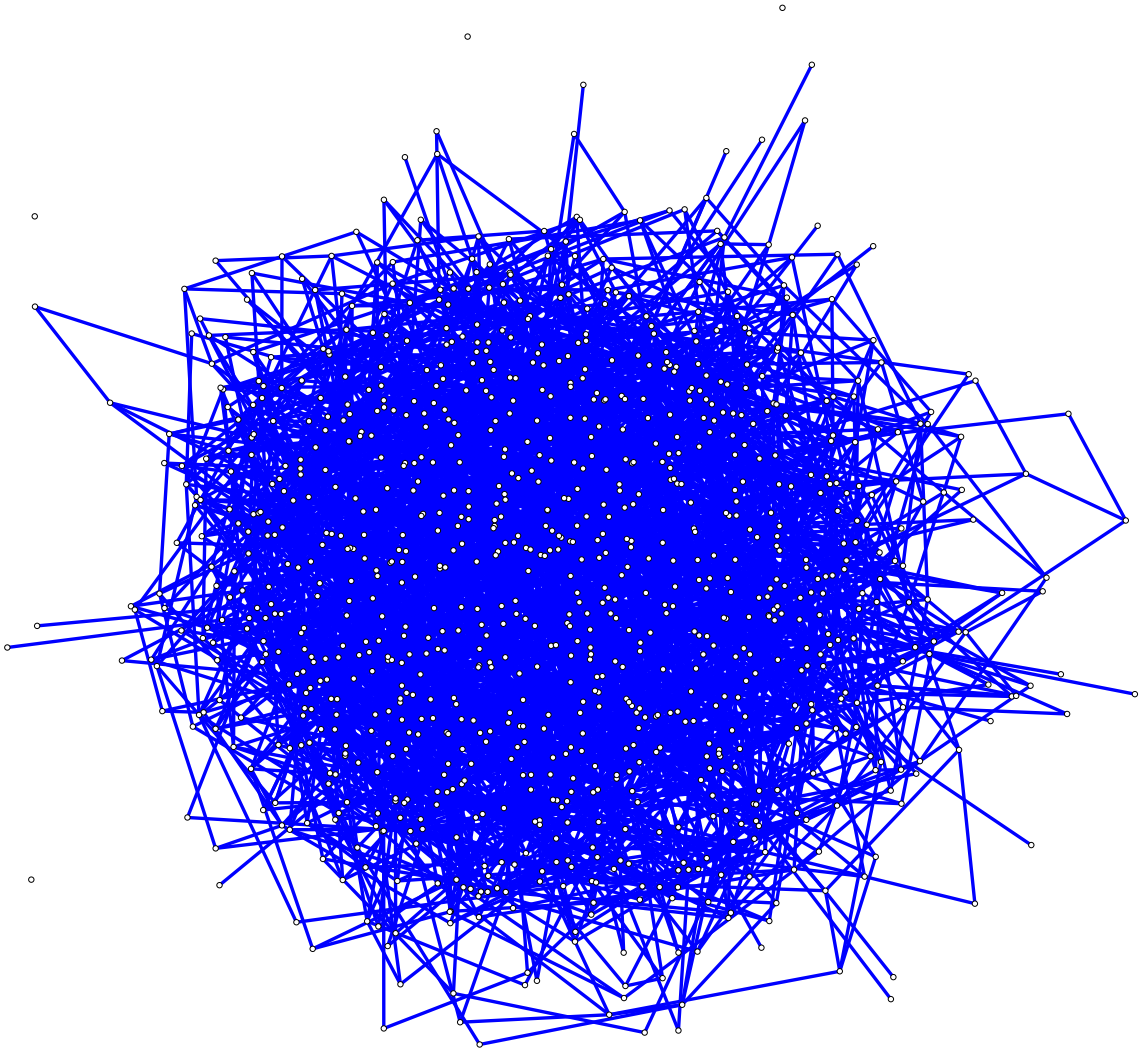

図 4.2: 初期ネットワーク

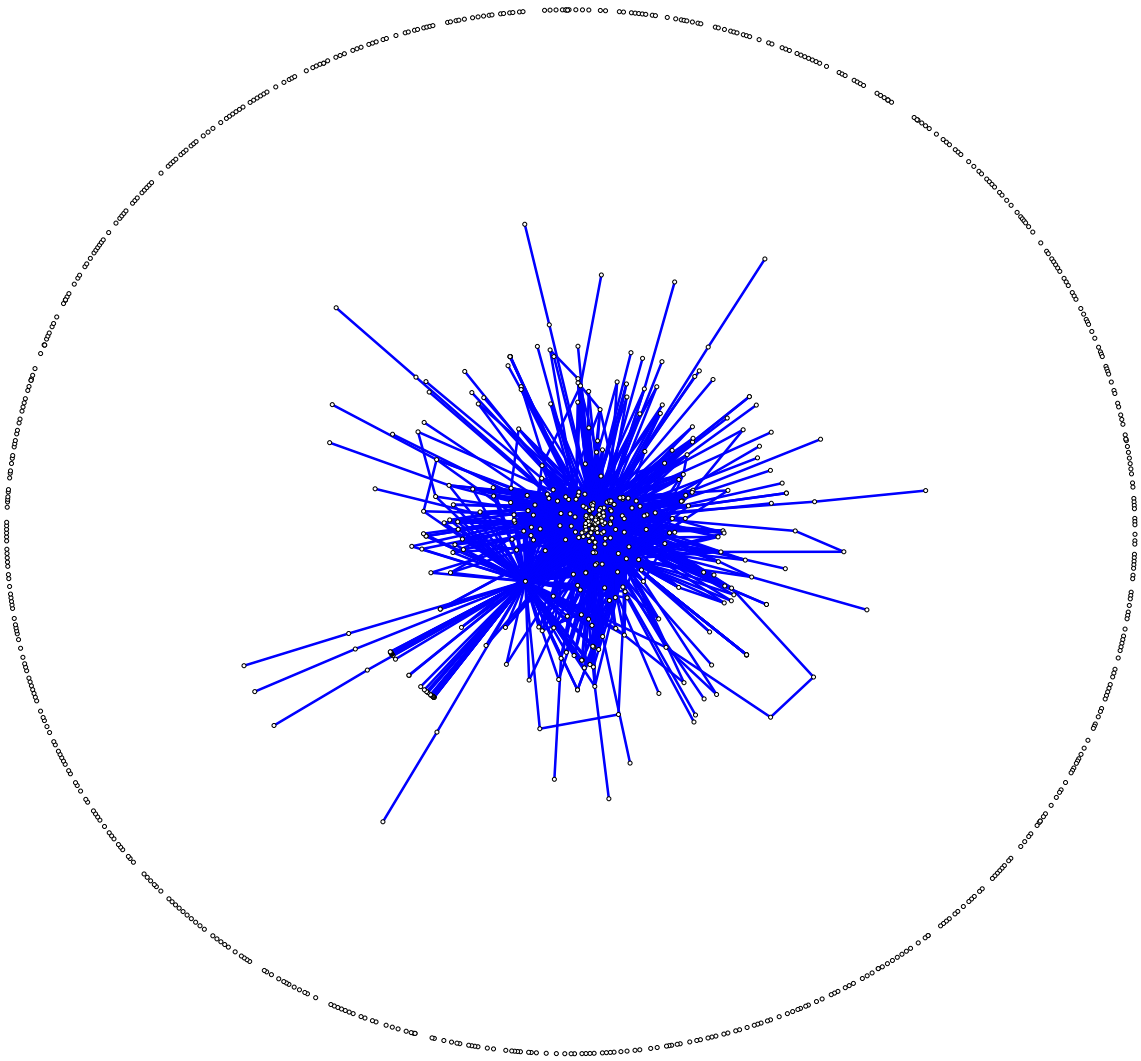

図 4.3: ネットワーク ($T = 1$)

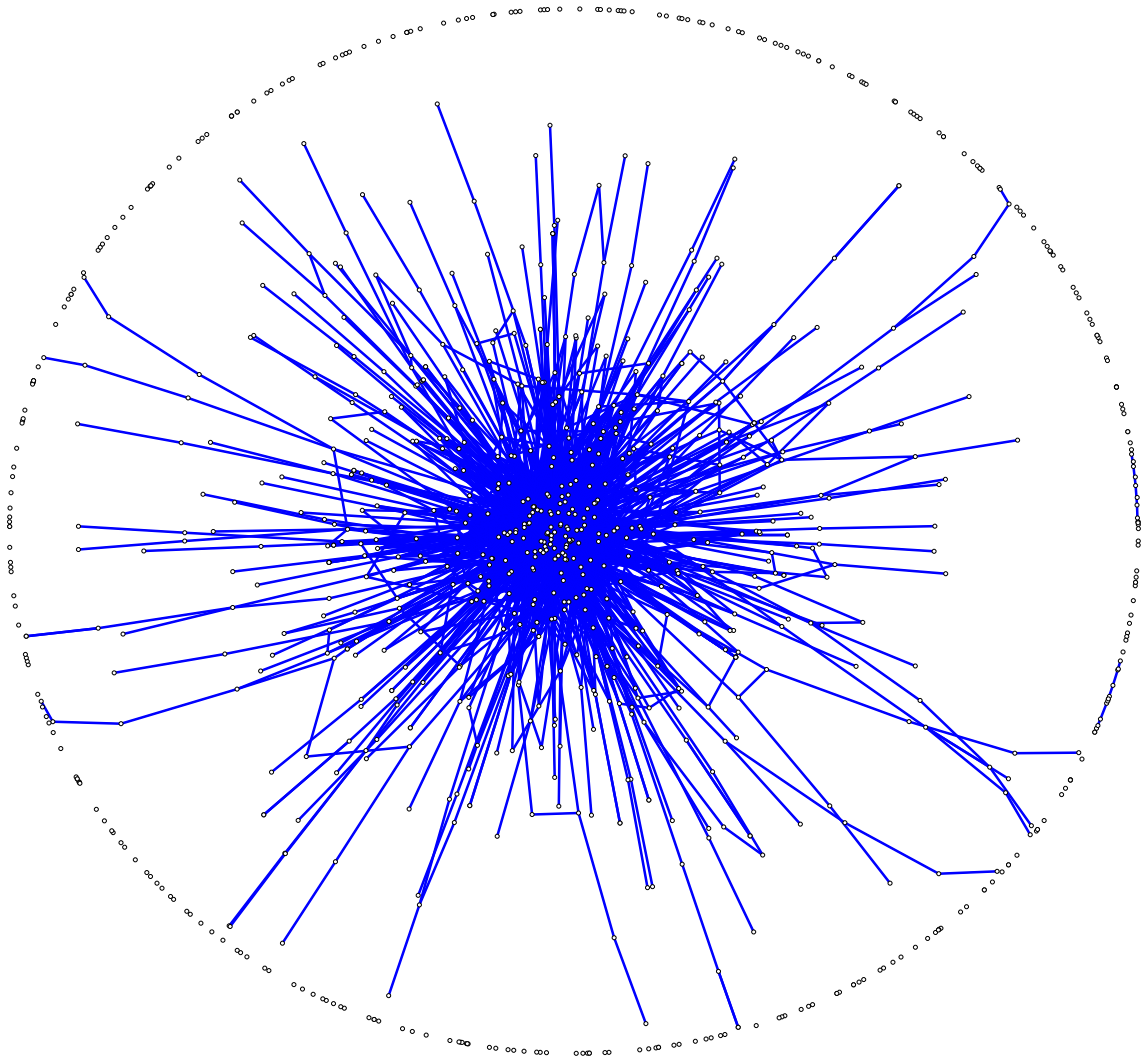

図 4.4: ネットワーク ($T = 1.5$)

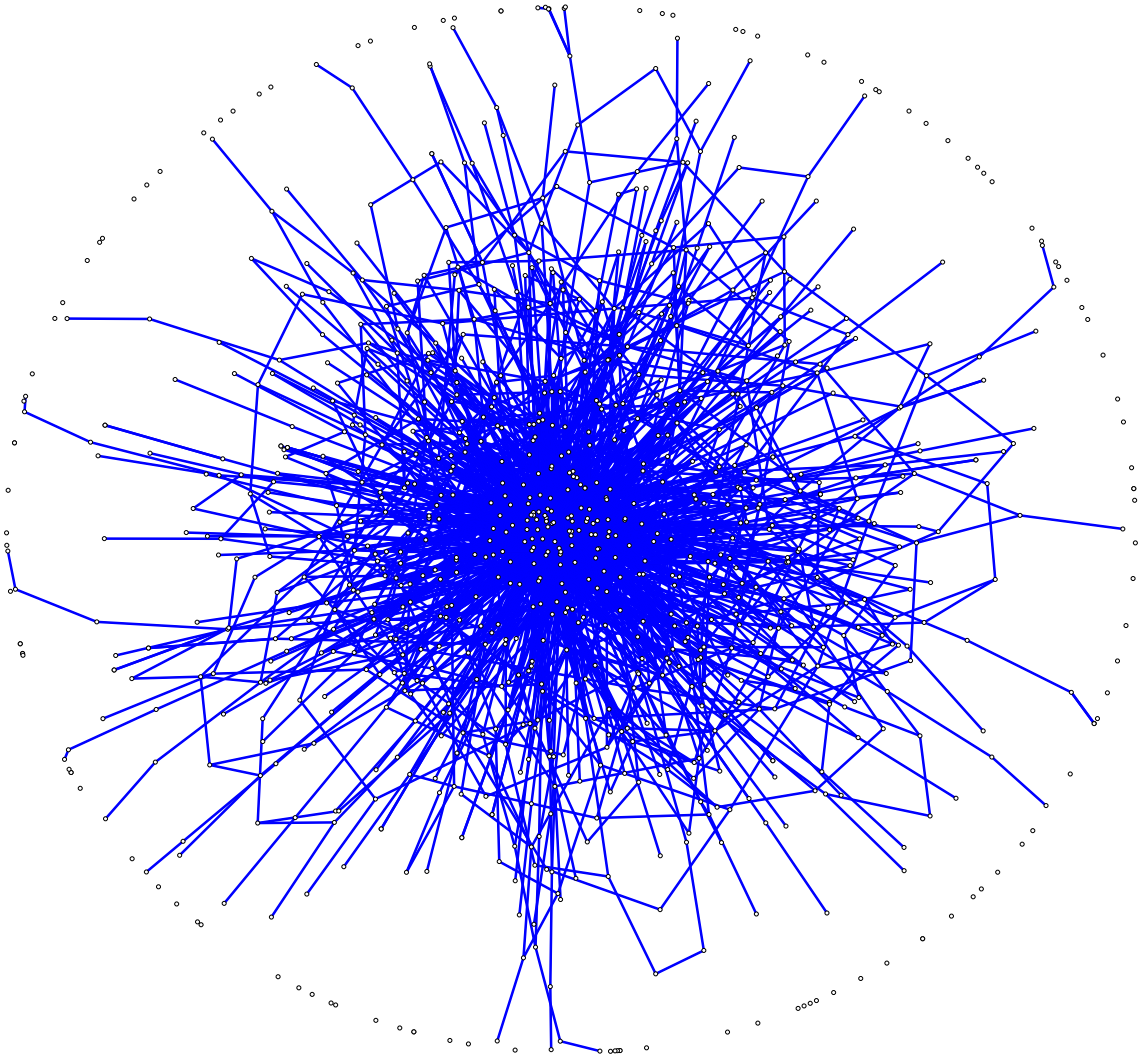

図 4.5: ネットワーク ($T = 2$)

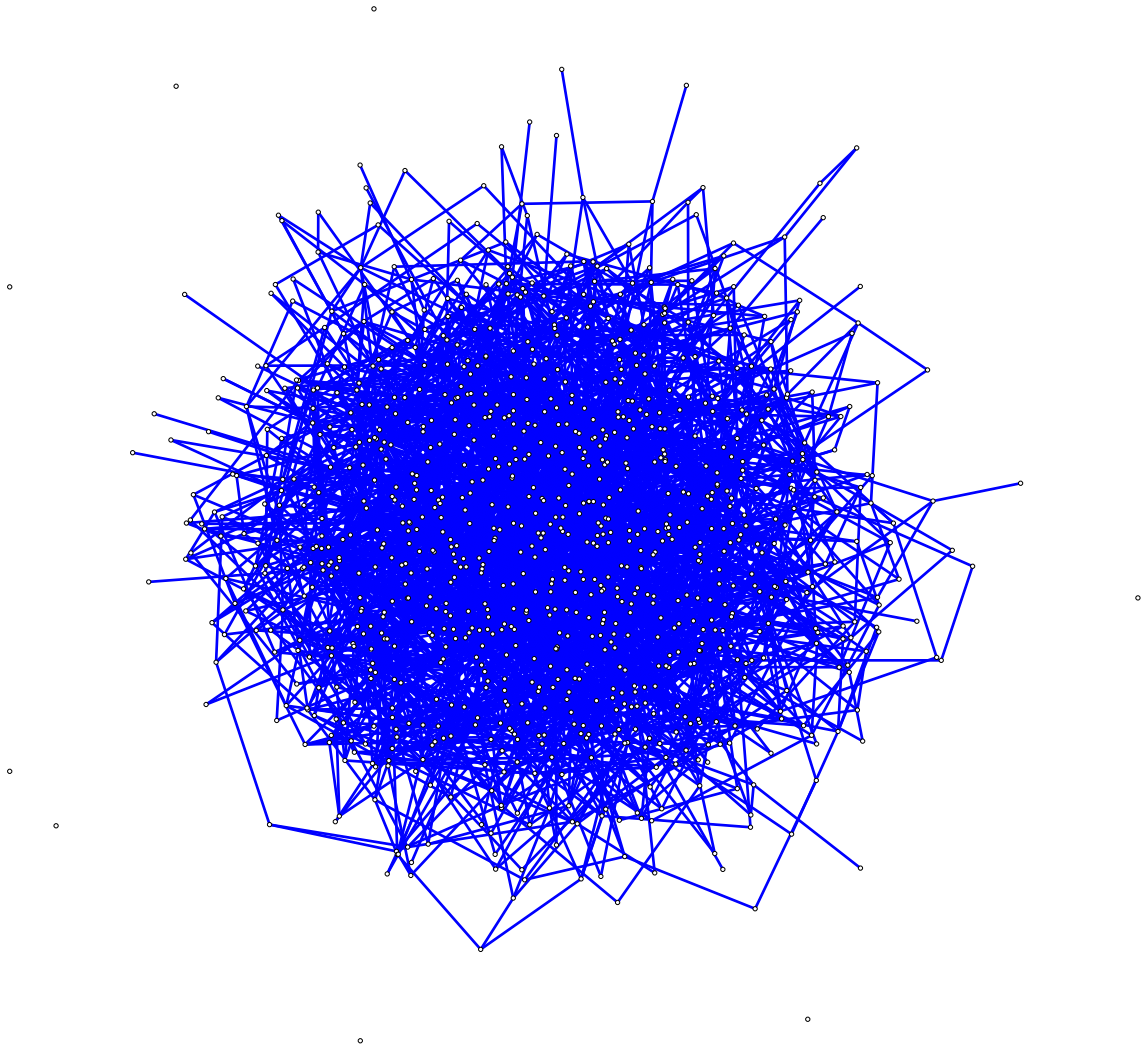

図 4.6: ネットワーク ($T = 5$)

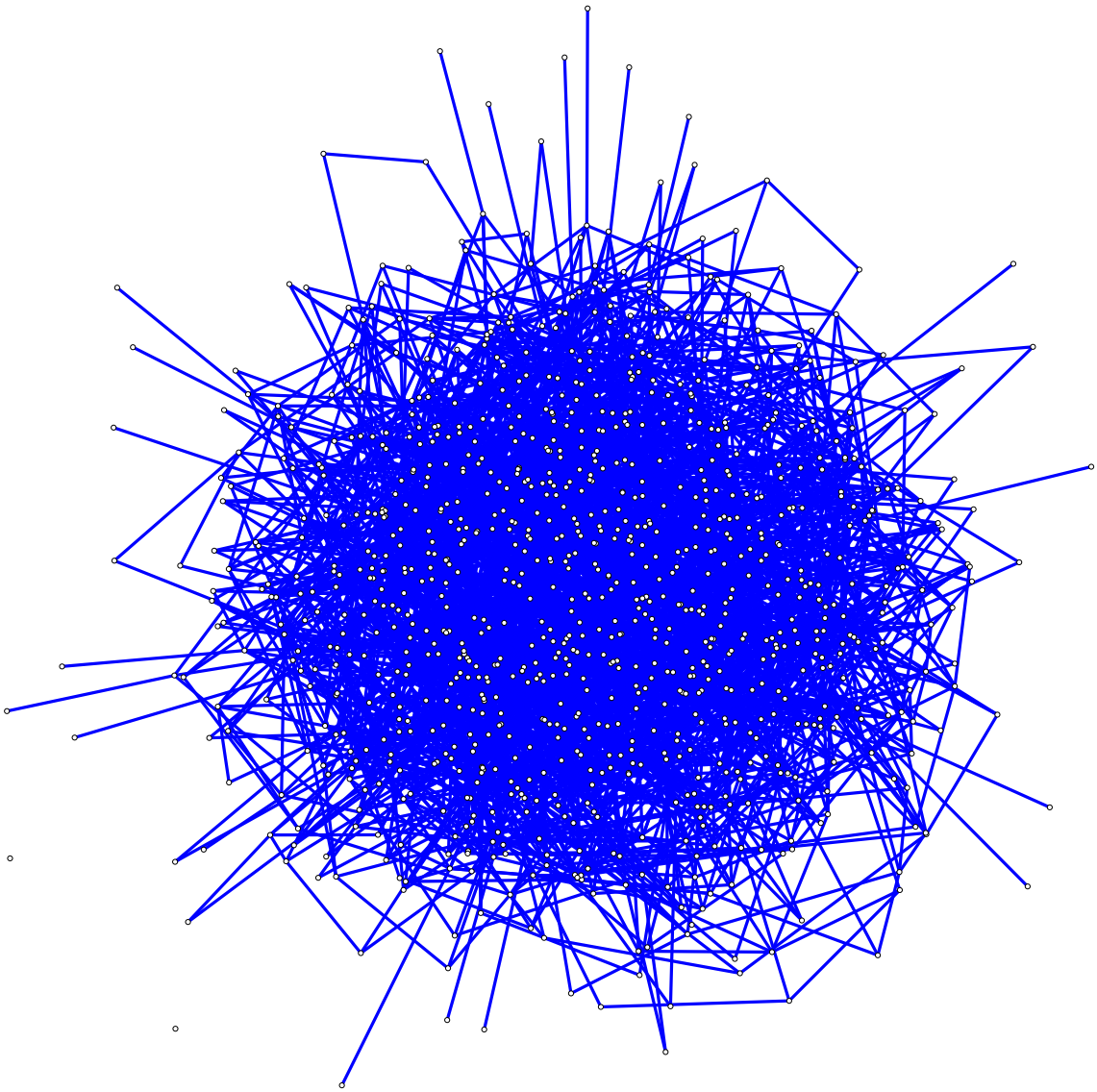

図 4.7: ネットワーク ($T = 10$)

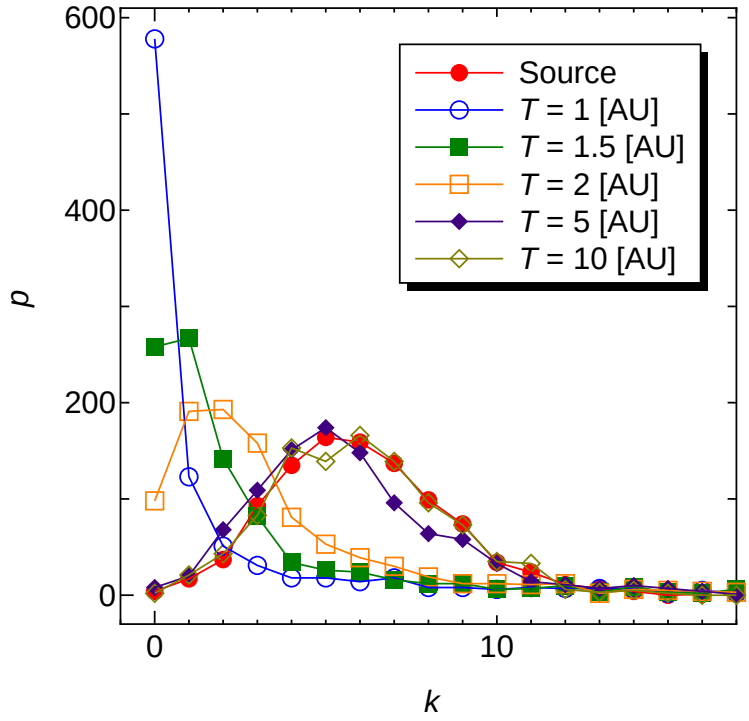


図 4.8: 次数分布

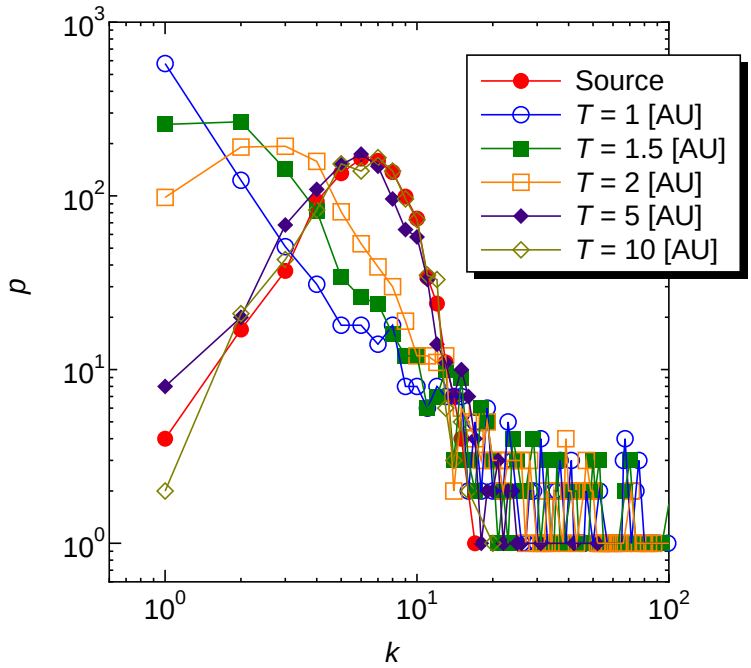


図 4.9: 次数分布 (両対数)

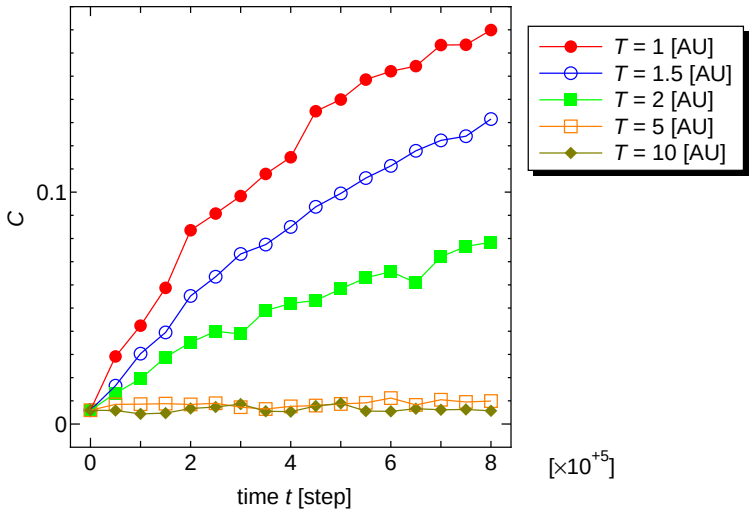


図 4.10: クラスター係数

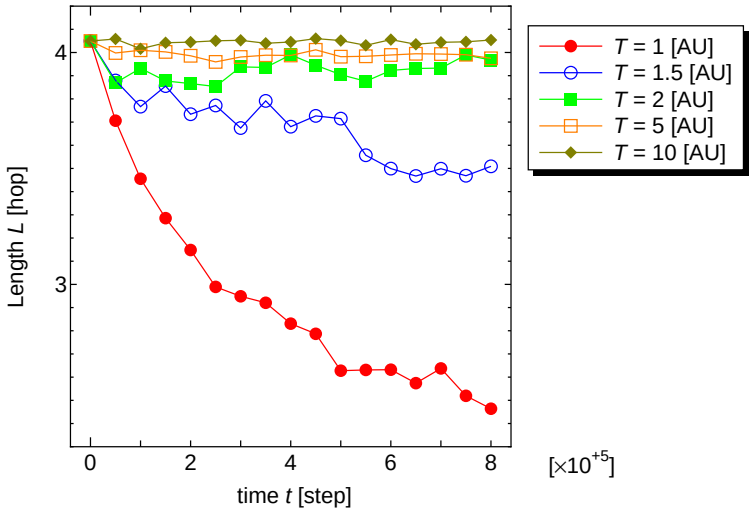


図 4.11: 平均最短経路長

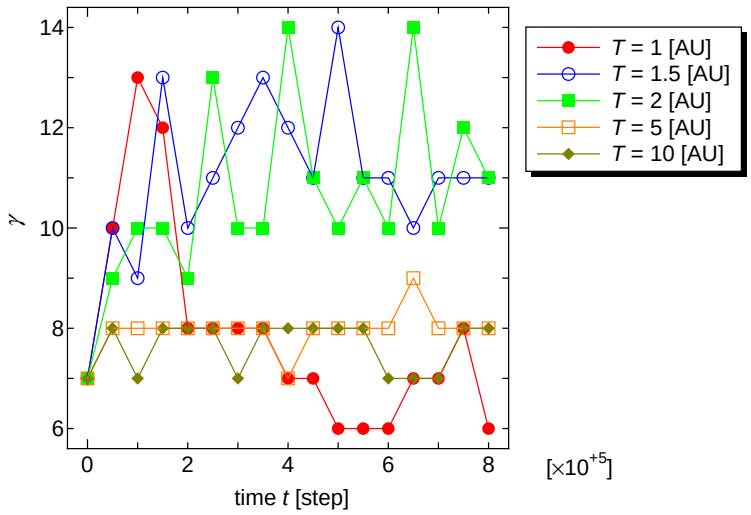


図 4.12: 直径

図 4.12 を見ると、直径は平均最短経路長よりも増減量のばらつきが激しい事が分かる。局所的な状態に大きく左右されやすことより、複雑ネットワークの全体的な傾向を見る指標としては適さないと分かる。

次に、時間に対するクリーク数変化を図 4.13、4.14、4.15、4.16、4.17 に、ベッチ数変化を図 4.18、4.19、4.20、4.21、4.22 に示す。

クリーク数変化及びベッチ数編かを観察すると、温度 $T$ が高い、即ちランダムネットワークにおいては値が殆ど変化しないことが分かる。温度 $T$ が高い場合については得られる情報が少く有用性に乏しいので、以下では $T = 1$〜$2$ について議論を行う。

温度 $T$ が低い、即ちノードが集中しスケールフリーネットワークへと推移するにつれて高い次元のクリークが出現している。高次元クリークが出現していることより、非常に多くの一カ所に集中していると言える。

次にベッチ数変化をみると、他の解析値と比較して値の増減が複雑である事が分かる。温度 $T$ が低いほど値が大きく変化している事が読み取れる。$T = 1$ に着目して見ていくと、$\beta_0$ は単調増加している事が分かる。0 次元ベッチ数はネットワークの分断数ともいえ、この場合であれば、時間の経過と共にネットワークが分断されている様子が分かる。次に、$\beta_1$ を見ると値が大きく減少して居る様子がうかがえる。1 次元ベッチ数はネットワークに開いた穴の数とも言える。本研究ではノード 3、リンク 3 からなる三角形を二次元単体とみなしているので、三角形部分はネットワークに開いた穴とは数えられない。さらにシミュレーション条件として、リンク数を一定としている事を考慮すると、$\beta_1$ の変化からは、リンクが集中し、多くの三角形を構成していることが読み取れる。$\beta_2$ 以降については、増加後ピークを迎え、減少している様子が読み取れる。高次元ベッチ数が出現するためには、高次元クリークによりサイクルが作られなくてはならないので、高次元ベッチ数の存在は、ネットワークが非常に大きな固まりを作っていることを表

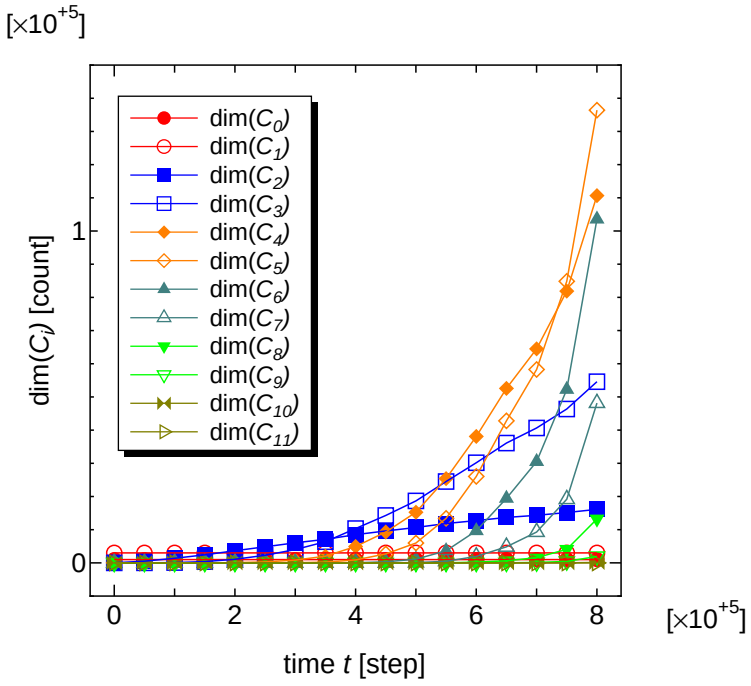


図 4.13: $T = 1$ におけるクリーク数変化

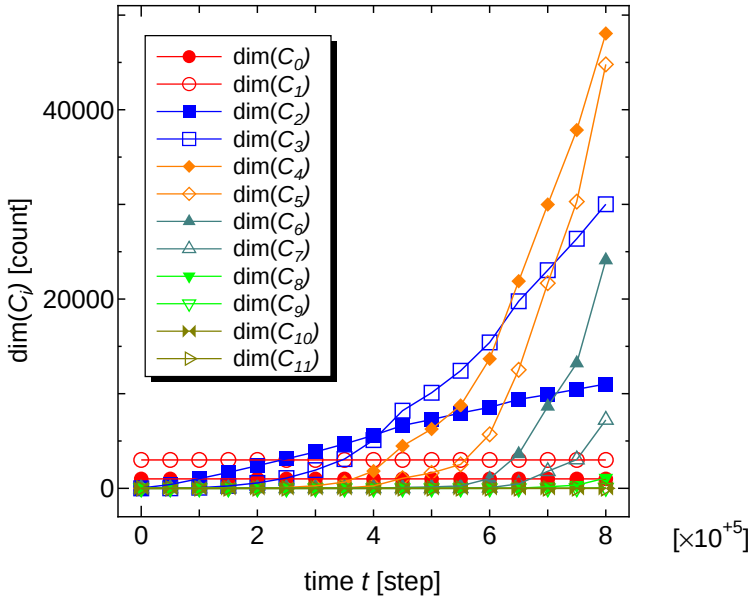


図 4.14: $T = 1.5$ におけるクリーク数変化

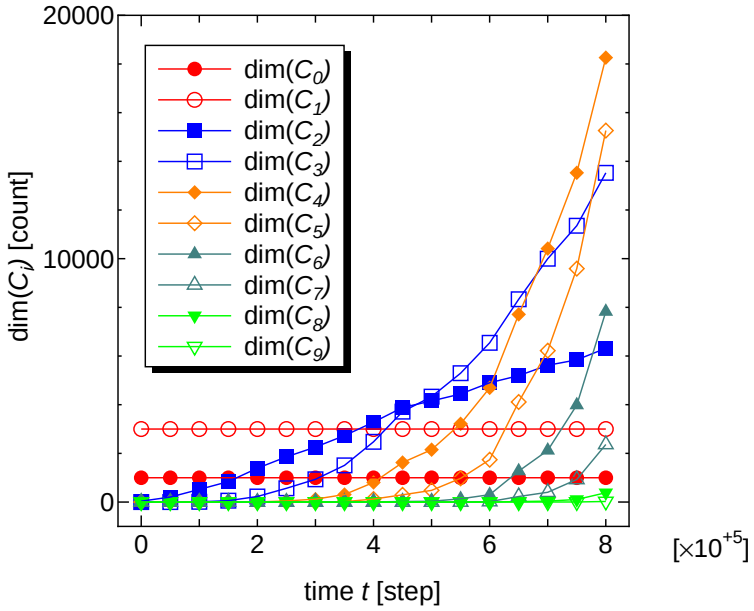


図 4.15: $T = 2$ におけるクリーク数変化

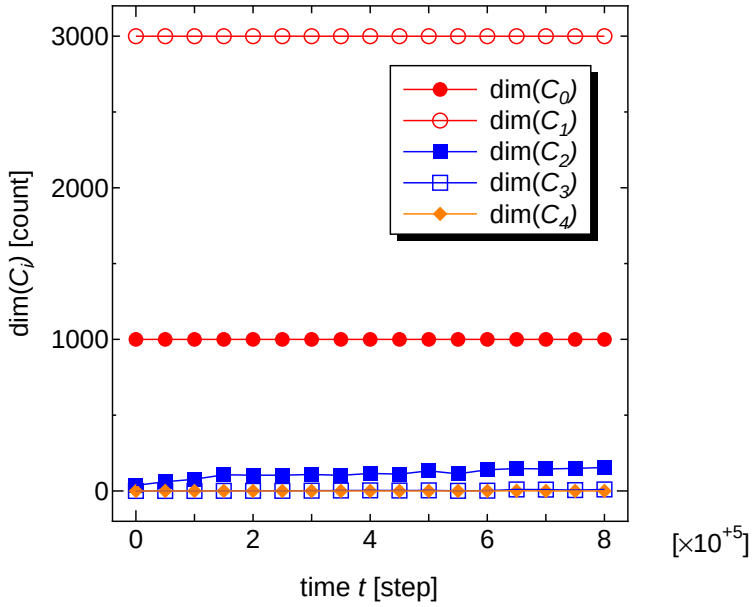


図 4.16: $T = 5$ におけるクリーク数変化

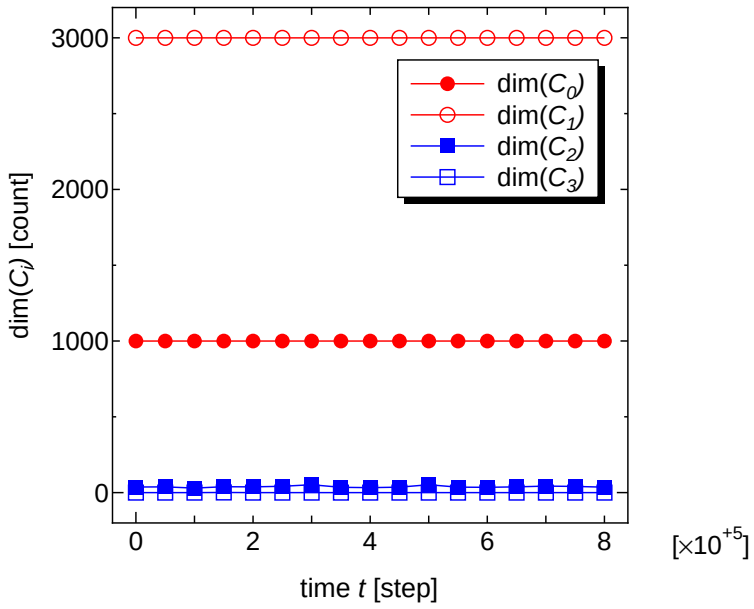


図 4.17: $T = 10$ におけるクリーク数変化

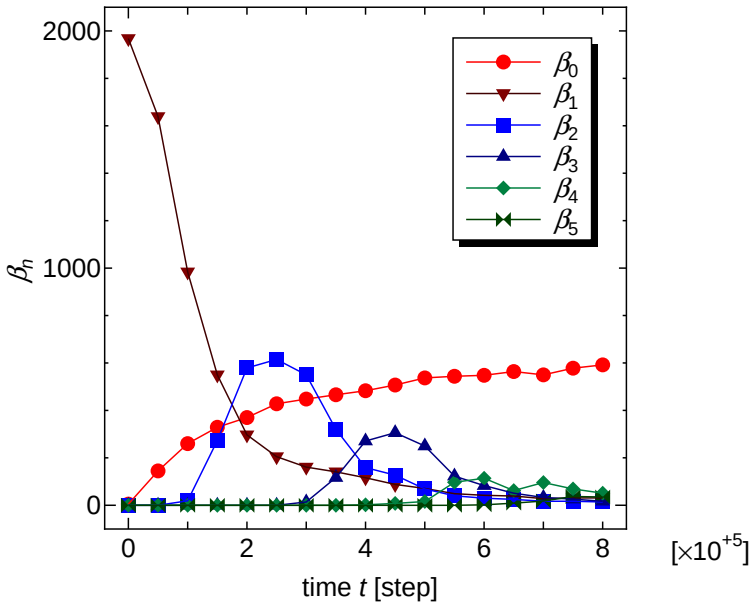


図 4.18: $T = 1$ におけるベッチ数変化

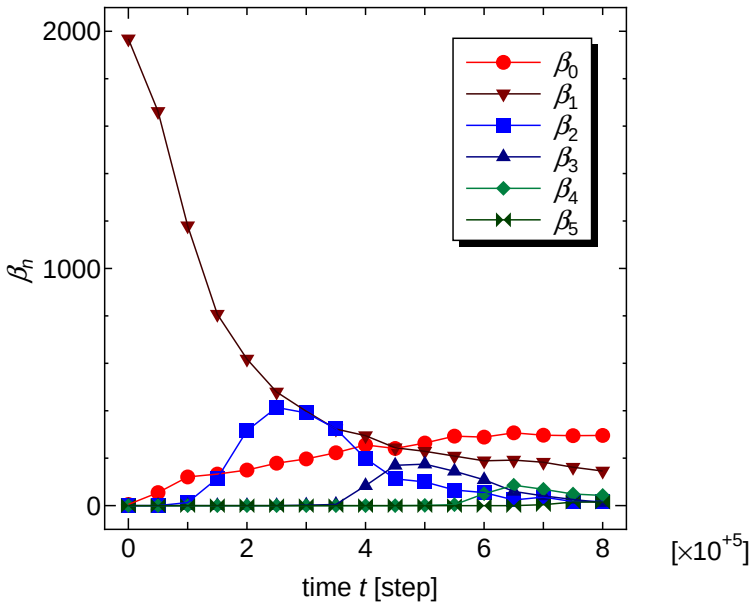


図 4.19: $T = 1.5$ におけるベッチ数変化

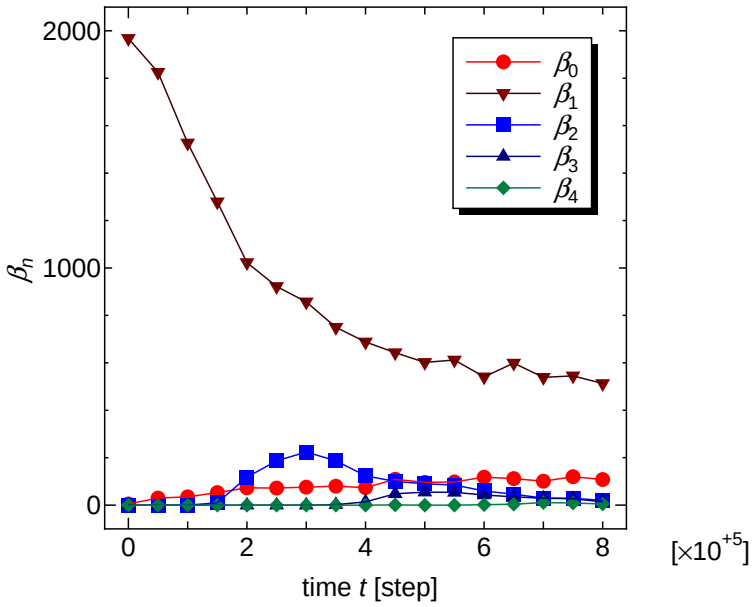


図 4.20: $T = 2$ におけるベッチ数変化

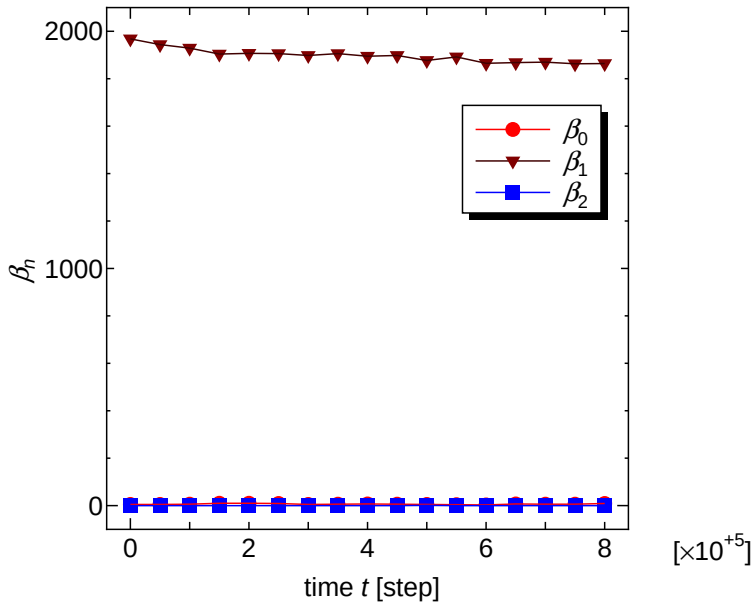


図 4.21: $T = 5$ におけるベッチ数変化

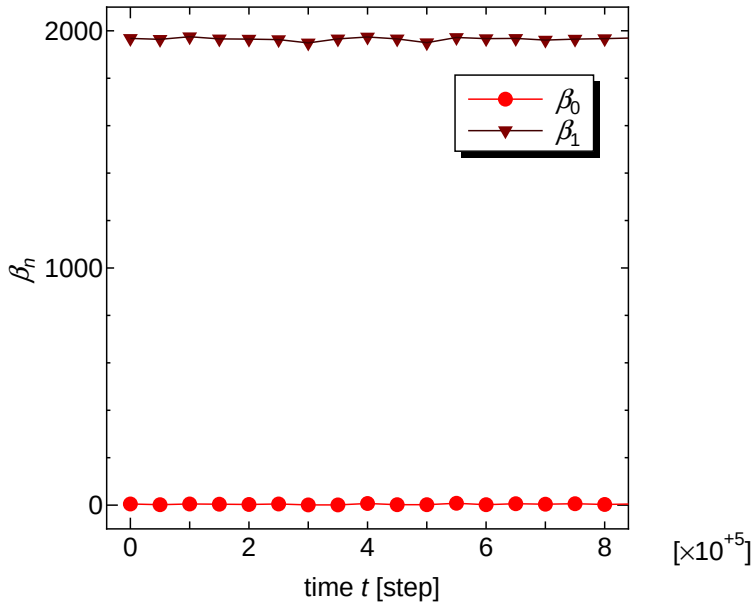


図 4.22: $T = 10$ におけるベッチ数変化

している。

$\beta_3$ に着目すると、先に述べたように増加しピークを迎えた上で減少している。この流れについて原因を考察する。先ずノードとリンクが 3 次元単体を多く構成し始め、それが集まった事により $\beta_3$ が増加し始める事は明白である。ここで、有る段階で増加が鈍化し減少に転じる理由について考える。ここまでの解析値から、ネットワークがさらにスケールフリー状態へと推移していることは明らかである。サイクルを構成する為には、適度にばらついていなくてはいけない事を考慮すると、減少に転じた理由は、サイクルを構成できないほどにノードとリンクが密集し始めたからと考えられる。多次元のベッチ数にも同様である。これより、高次元ベッチ数の存在は、単にネットワークが密集しているだけではなく、“ 隙間が無い ”ネットワーク状態である事を意味すると言える。

ベッチ数の定義、また実際に得られた結果をみても、ベッチ数間の関係から得られる知見があることが分かる。ここで、$T = 1, 1.5, 2$ について次元を横軸にとり、ベッチ数を縦軸にとったグラフを図 4.23、4.24、4.25 に示す。これが、本研究で提案するベッチスペクトルである。

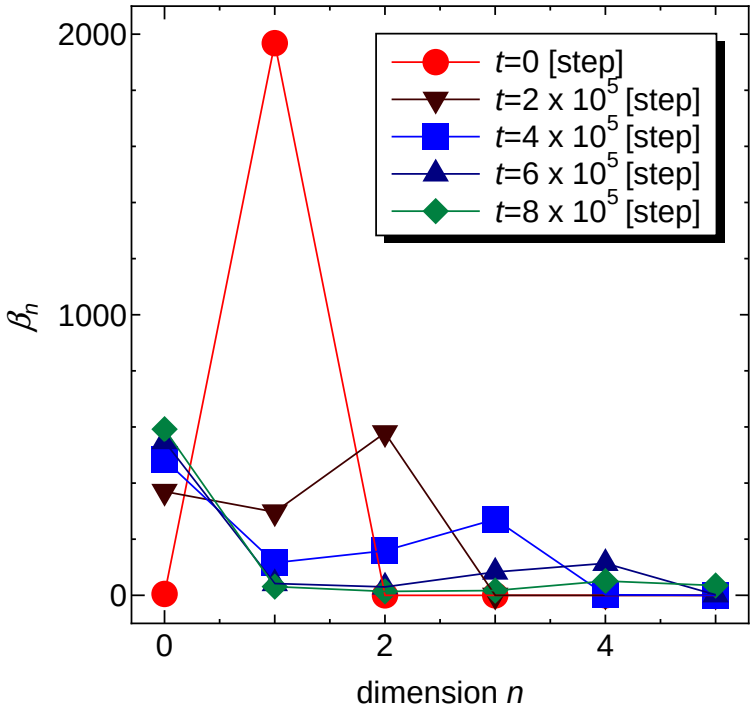


図 4.23: $T = 1$ におけるベッチスペクトル

次元間の関係については、先の議論と同様である。まとめると、0 次元成分からネットワークの分断数、1 次元以降の成分から、ネットワークの密集度を読み取ることが出来る。

実際にネットワークを解析する場合を考えると、ある瞬間におけるネットワーク状態を知るために解析することが多いと予想される。ベッチ数変化グラフは変化過程における解析としては有用であるが、ある瞬間における解析には向かない。

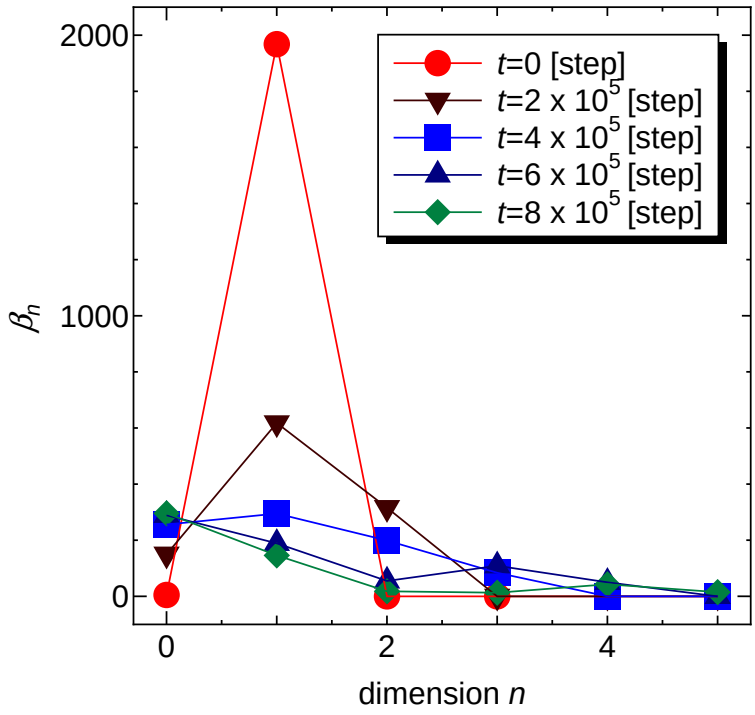


図 4.24: $T = 1.5$ におけるベッチスペクトル

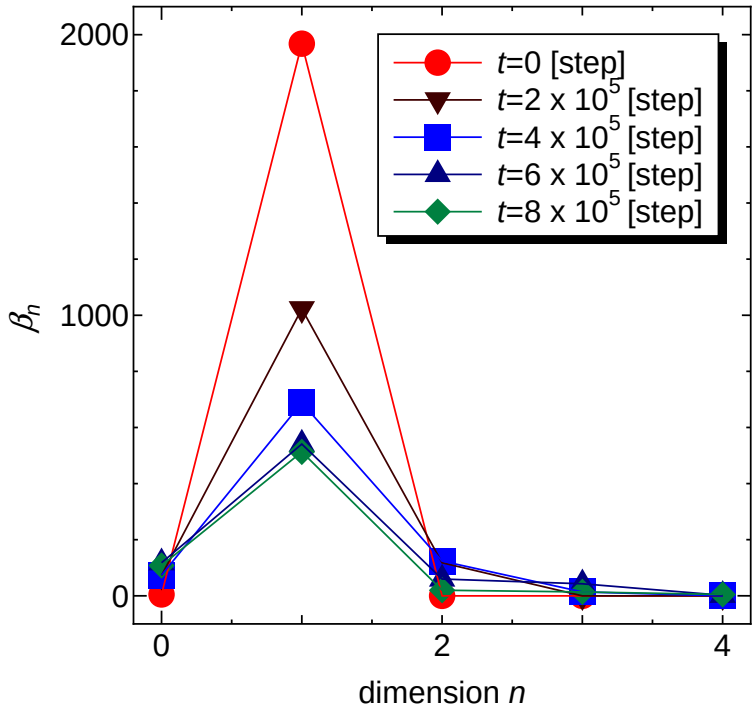


図 4.25: $T = 2$ におけるベッチスペクトル

ベッチスペクトルであれば、ある瞬間における解析値として有効に使えると考えられる。

また、次元間の関係が一目で見やすく、知見を得やすいといった特徴もある。

## 4.2 シミュレーション２

グラフ中で、1000_3 とはノード数 1000、リンク数 $1000 * 3 = 3000$ であることを表している。クラスター係数を図 4.26、平均最短経路長を図 4.26、直径を図 4.26 に示す。また、クリーク数を図 4.29、4.30、4.31 に、ベッチ数を図 4.32、4.33、4.34 に示す。

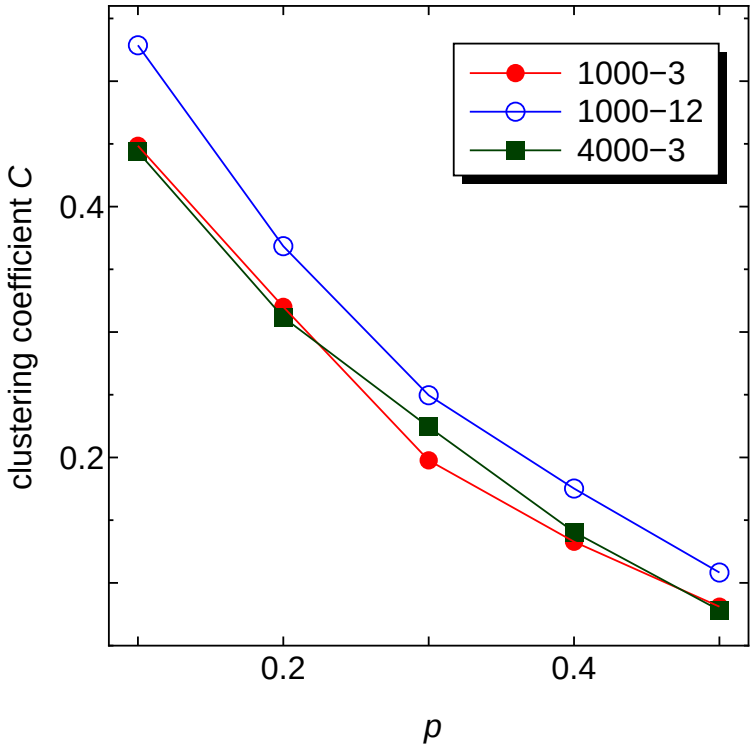


図 4.26: WS モデルにより生成されたネットワークのクラスター係数

図 4.26 から、辺繋ぎ替え確率 $p$ が増えるほどクラスター係数が減少していることが分かる。これは、$p$ が増加するほど得られるネットワークがランダムネットワークに近くなるためである。特に $p = 0.5$ では、リンクの半数が無作為に繋ぎ替えられている為に、かなり低い値となっている。

図 4.27 を見ると、辺繋ぎ替え確率 $p$ が増えるほど平均最短経路長が減少している様子が伺える。1000_12000 のネットワークはノード数に対するリンク数が多いために、平均経路長が小さな値となっていると分かる。また、ノードとリンク数の比率が同じでもノード数が少ない方が短い平均経路長を持つことが分かる。これは WS モデルの生成過程に由来していると思われている。また、図 4.28 に示された直径についても同様である。

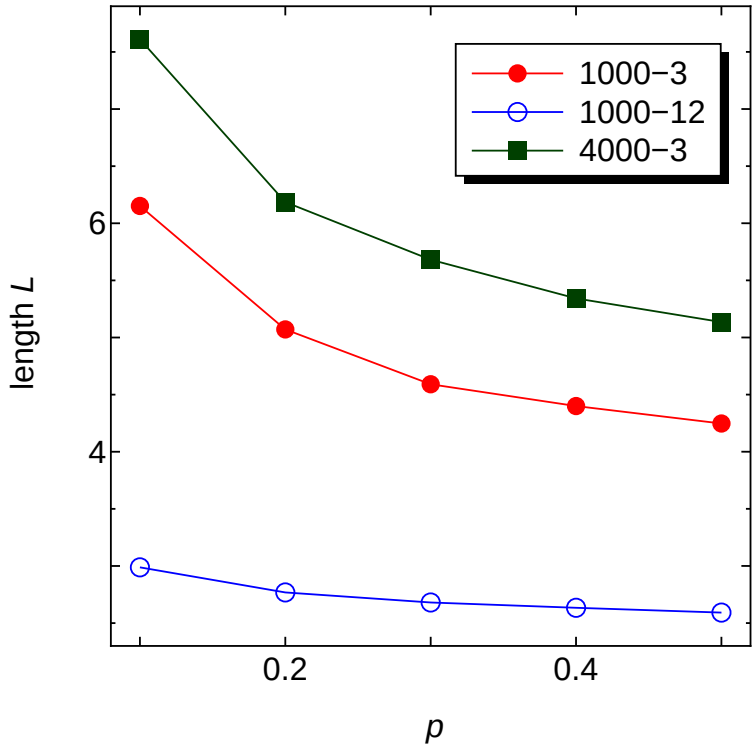


図 4.27: WS モデルにより生成されたネットワークの平均最短経路長

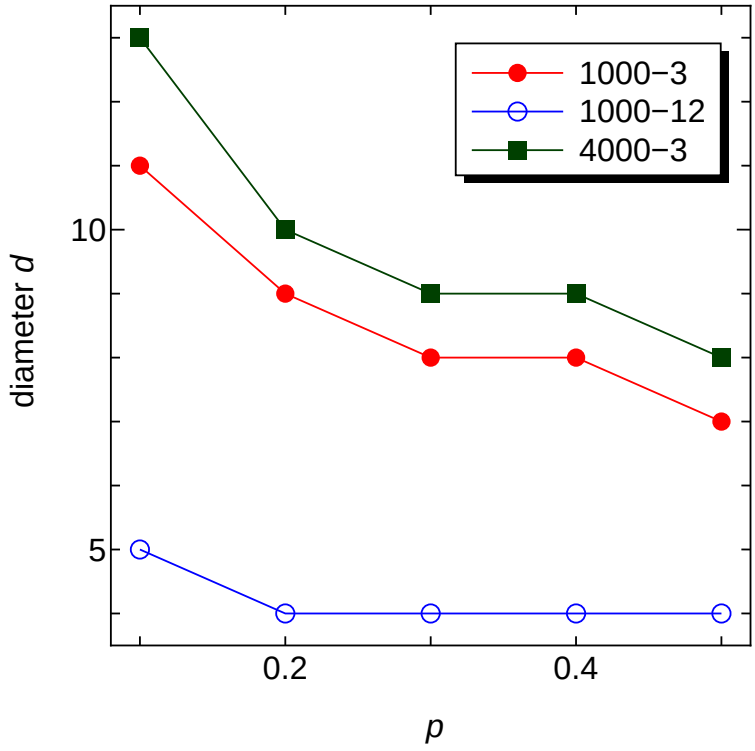


図 4.28: WS モデルにより生成されたネットワークの直径

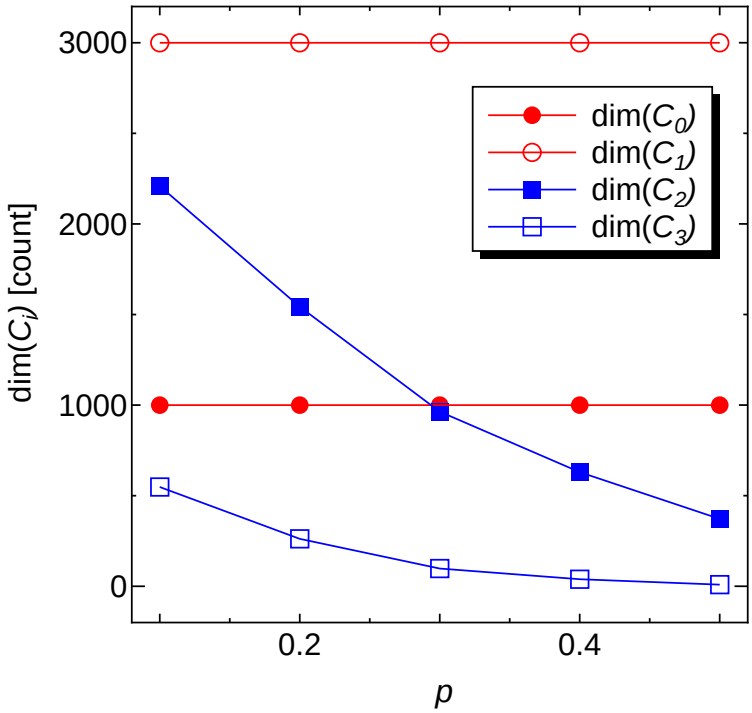


図 4.29: WS モデルにより生成されたネットワークのクリーク数 (ノード数 1000、リンク数 3000)

図 4.29、4.30、4.31 に示すクリーク数を比較すると、1000_12 のネットワークにおいて高次元のクリーク数が確認され、残り二つについては 3 次元クリークが出現しているだけである。WS モデルによるネットワーク生成過程を考えると、ノードを生成し、リンクを生成した段階（$p$ に応じて辺の繋ぎ替えを行う前）で隣り合ういくつかのノード同士でクリークを構成する事が分かる。よって実際に解析値を見ても、最も辺の繋ぎ替えを行わない $p = 0.1$ においてもっとも多くの高次元クリークが得られることが分かる。

図 4.32、4.33、4.34 に示すベッチ数を考える。1000_3 及び 4000_3 のネットワークについてはほぼ同等の傾向を示していることが分かる。この二つのネットワークは $\beta_0$ から分断されていない事が、$p$ の増加に伴い $\beta_1$ が増えてきていることより、ネットワークがランダムネットワークに近いほどノード及びリンクの密集度が低い事が分かる。次に、1000_12 のベッチ数について見ると、0 次元、1 次元については 1000_3 及び 4000_3 のネットワークと同様の議論である。$\beta_2, \beta_3, \beta_4$ については興味深い変化を示している。次元間の関係を見るために、ベッチスペクトル表記したものを図 4.35 に示す。また、一次元ベッチ数が比較的大きな値を持つので、見やすさの為に 1 次元ベッチ数を非表示としたスペクトルを図 4.36 に示す。

図 4.35、4.36 をみると、$p = 0.1$ においては値を持たないように見える。しかし、実際には $\beta_3$、$\beta_4$ において値を持っている。$p$ の値が増えるにつれ高次元成分が増え、次に低次元成分が増加している様子が伺える。高次元クリークの多く出現する $p = 0.1$ で高いベッチ数成分が出現し易いと考えるのが自然であるが、WS モデ

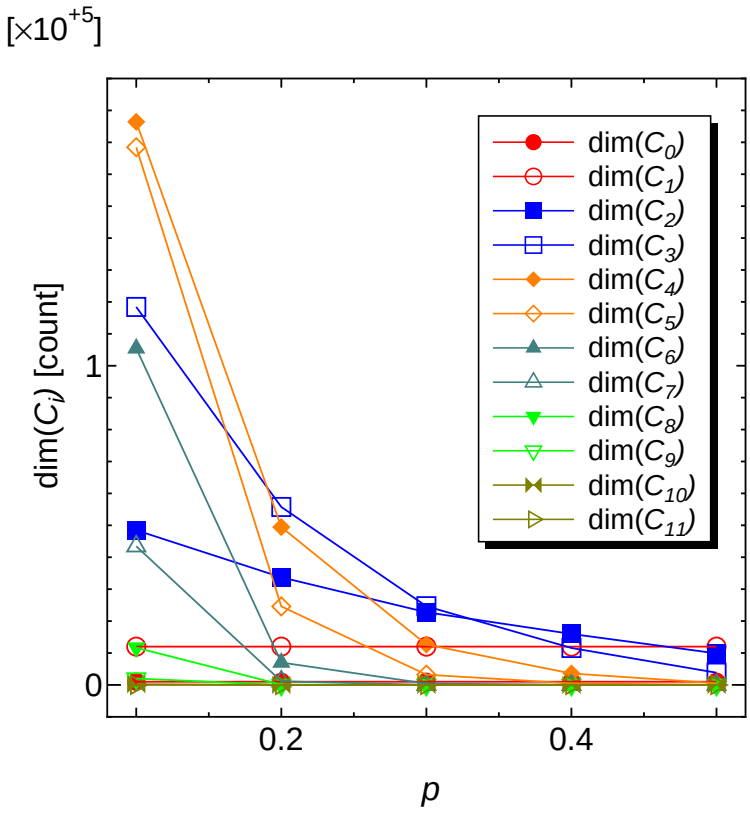


図 4.30: WS モデルにより生成されたネットワークのクリーク数 (ノード数 1000、リンク数 12000)

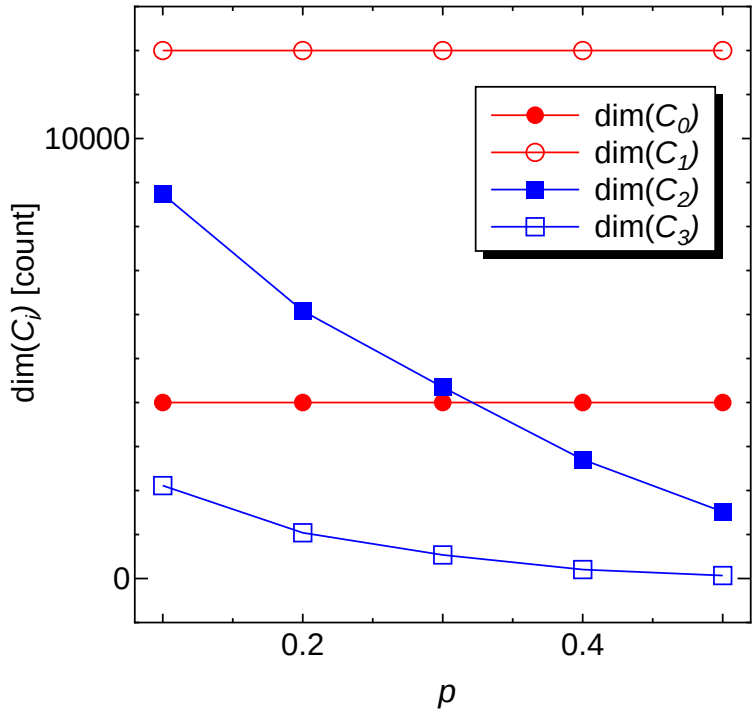


図 4.31: WS モデルにより生成されたネットワークのクリーク数 (ノード数 4000、リンク数 12000)

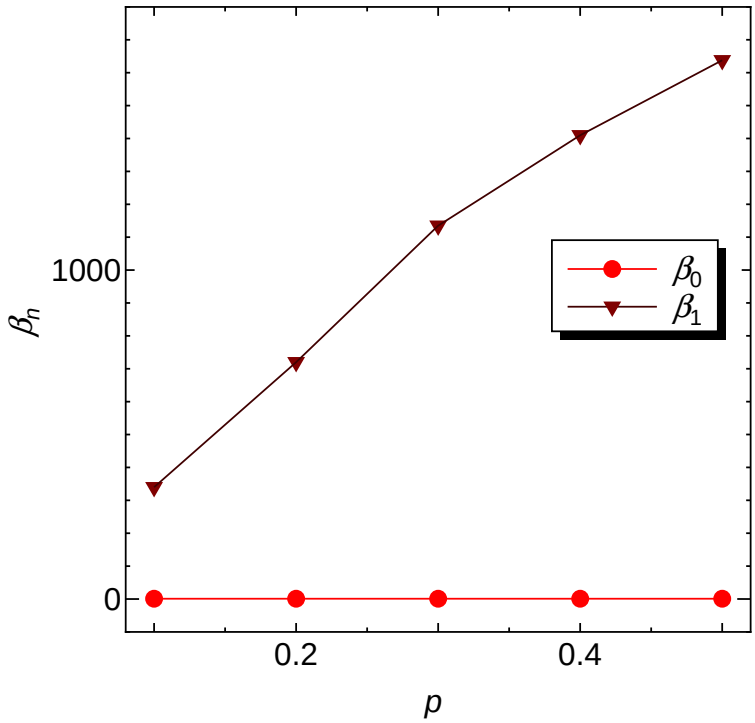


図 4.32: WS モデルにより生成されたネットワークのベッチ数 (ノード数 1000、リンク数 3000)

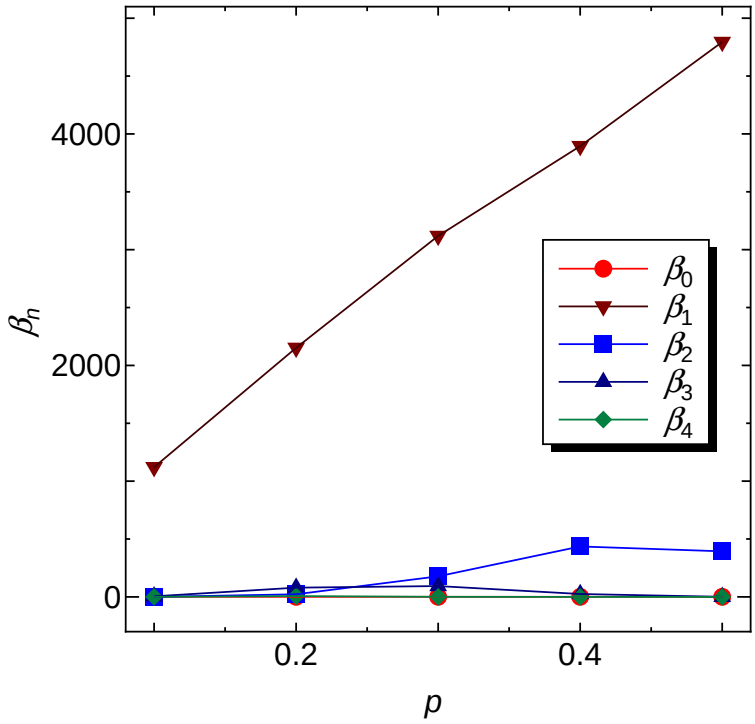


図 4.33: WS モデルにより生成されたネットワークのベッチ数 (ノード数 1000、リンク数 12000)

ルにより生成されたネットワークについては逆の傾向を示している。

これは $p$ が低い場合、規則的が非常に強いネットワークが生成され、その規則性故に高次元クリークが検出されつつも、ベッチ数が抑えられてしまうと考えられる。これは、ベッチスペクトルが規則的なネットワークの解析に適していないことを示唆している。

これについては、解析前のネットワークについて前処理を施し（規則性を適度に崩し）解析を行うことで有用な知見を得られると考えられる。

## 4.3 総評

従来から用いられている指標からは単一的な情報しか得られなかったが、ベッチスペクトルからは、分断数やノード及びリンクがどの程度密集しているかといったいくつかの知見を得ることが出来た。これはネットワークについて定量的に解析するという意味では大きな意味を持つと考えられる。しかし、シミュレーション 2 で明らかになったように、ネットワークの形状によっては全く逆の傾向を示すことが分かった。

今後、今回の研究で想定していないネットワーク形状についても調べる事で新たな知見が得られると考えられる。さらに研究を進めていき、ベッチスペクトルについて理解及びその応用方法が多く考案されれば、ベッチスペクトルは非常

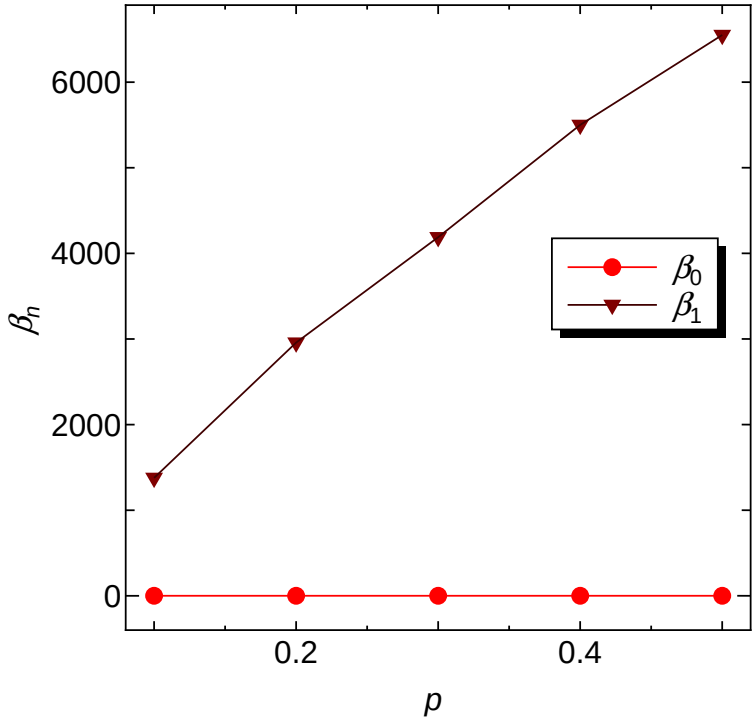


図 4.34: WS モデルにより生成されたネットワークのベッチ数 (ノード数 4000、リンク数 12000)

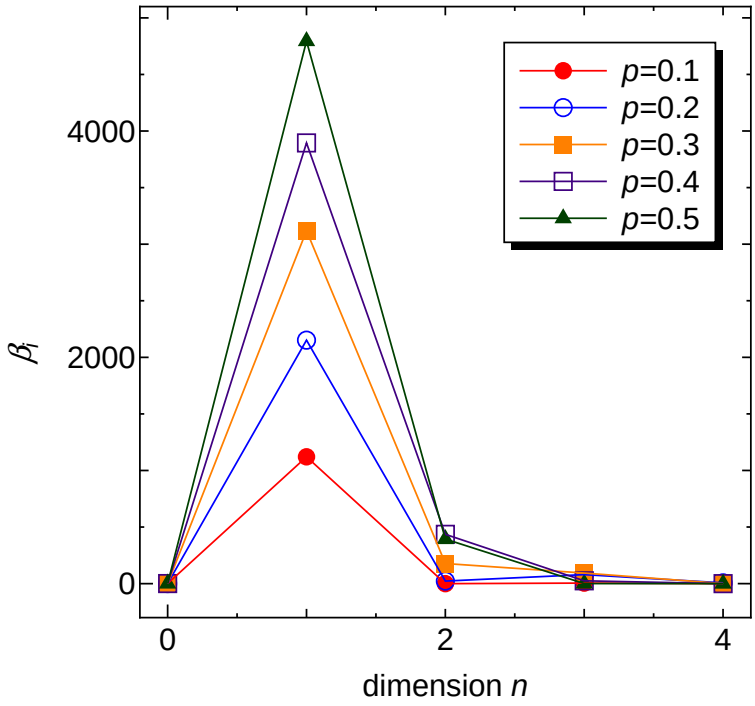


図 4.35: WS モデルにより生成されたネットワークのベッチスペクトル (ノード数 1000、リンク数 12000)

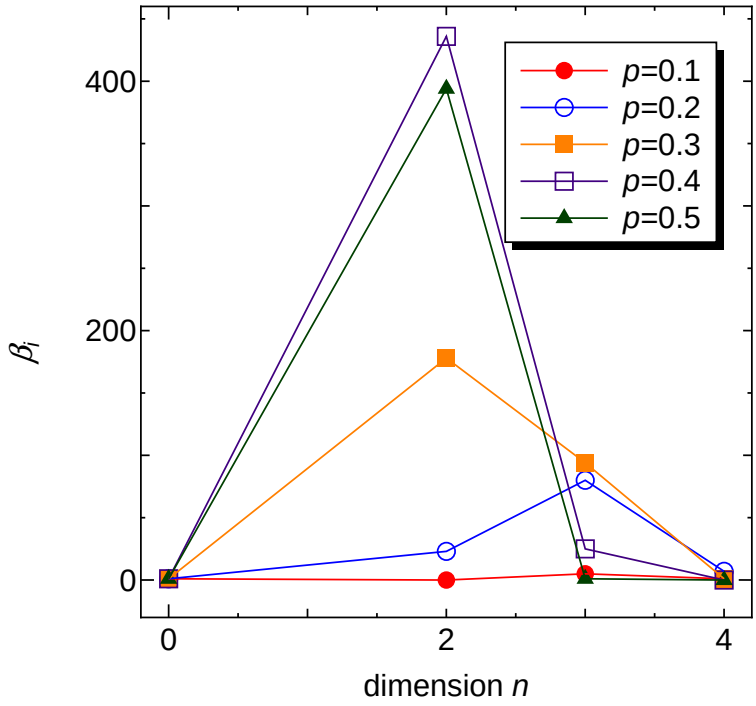


図 4.36: WS モデルにより生成されたネットワークのベッチスペクトル (ノード数 1000、リンク数 12000、1 次元ベッチ数を除外)

に有効な解析手法になると期待できる。

# 第5章　総括

## 5.1　まとめ

ベッチスペクトルからは、ネットワークが分断されているか、またどの程度ノード及びリンクが密集しているか知ることが出来た。まだ不明な部分もあり、不明な部分については今後の研究でさらに明らかになると思われる。

目的である非常に複雑なネットワークについて有効な解析値を得ることができた。しかし、ネットワーク形状によっては有効な解析値が得られない事が有ることも分かった。

改善方法として、ネットワークに対して前処理を行う事で有効な解析値を得るといった手法等が思いつく。また他にも様々な手法が考えられるだろう。

ベッチスペクトルは現状では汎用的な解析手法には至っていない。しかし、いくつかのネットワークについて調査した感触では、十分な有効性を秘めていると考えられる。

## 5.2　今後の課題

さらにベッチスペクトルについて研究を進めていく必要がある。具体的には

- ネットワークに対し前処理を行い、有効なベッチスペクトルを得る
- 今回想定していないネットワーク構造について、どのようなベッチスペクトルが得られるか

について調査及び研究する必要がある。

# 付 録A　ホモロジー計算

## A.1　例題

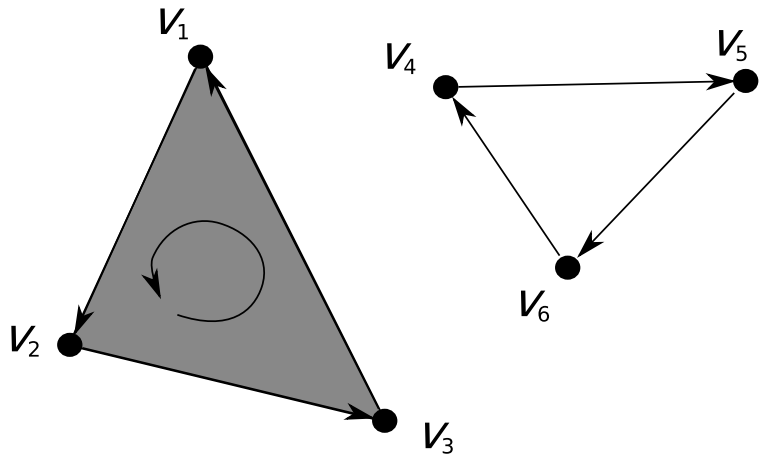


図 A.1: ホモロジー例題

問題の形状として図 A.1 を示す。これは一つの二次元単体、六つの一次元単体、六つの 0 次元単体からなる複体である。

## A.2　解

図 A.1 の最大次元は 2 なので、鎖複体は

$$\begin{array}{ccccccccc} & & \partial_3 & & \partial_2 & & \partial_1 & & \partial_0 & \\ 0 & \rightarrow & C_2(K) & \rightarrow & C_1(K) & \rightarrow & C_0(K) & \rightarrow & 0 \end{array} \tag{A.1}$$

で与えられる。

さらに、各次元の鎖群は次の式で与えられる。

$$C_2(K) = \{n(v_1v_2v_3)|n \in Z\} \tag{A.2}$$

$$\begin{aligned} C_1(K) = \{&m_0(v_1v_2), m_1(v_2v_3), m_2(v_3v_1), m_3(v_4v_5), m_4(v_5v_6), m_5(v_6v_4) \\ &|m_0, m_1, m_2, m_3, m_4, m_5 \in Z\} \end{aligned} \tag{A.3}$$

$$\begin{aligned} C_0(K) = \{&l_0(v_1), l_1(v_2), l_2(v_3), l_3(v_4), l_4(v_5), l_5(v_6) \\ &|l_0, l_1, l_2, l_3, l_4, l_5 \in Z\} \end{aligned} \tag{A.4}$$

$C_3(K) = 0$ より、明らかに

$$\mathrm{Im}\partial_3 = 0 \tag{A.5}$$

である (Im は像を表す)。

次に、$\partial_2 (n(v_1v_2v_3)) = n\{(v_1v_2) + (v_2v_3) + (v_3v_1)\} = 0$ を考えると、$n = 0$ となる。よって、

$$\mathrm{Ker}\partial_3 = 0 \tag{A.6}$$

となる (Ker は核を表す)。また、

$$\mathrm{Im}\partial_2 = \{(\mathrm{v}_1\mathrm{v}_2) + (\mathrm{v}_2\mathrm{v}_3) + (\mathrm{v}_3\mathrm{v}_1)\} \tag{A.7}$$

となる。

ここで、$\mathrm{Z}_1 = m_0(v_1v_2) + m_1(v_2v_3) + m_2(v_3v_1) + m_3(v_4v_5) + m_4(v_5v_6) + m_5(v_6v_4)$ とおくと、

$$\begin{aligned}
\partial_1(\mathrm{Z}_1) &= M_0((v_2) - (v_1)) + M_1((v_3) - (v_2)) + M_2((v_1) - (v_3)) \\
&\quad + M_3((v_5) - (v_4)) + M_4((v_6) - (v_5)) + M_5((v_4) - (v_6)) \\
&= (m_2 - m_0)(v_1) + (m_0 - m_1)(v_2) + (m_1 - m_2)(v_3) + \\
&\quad + (m_5 - m_3)(v_4) + (m_3 - m_4)(v_5) + (m_4 - m_5)(v_6) \\
&= 0
\end{aligned}$$

を満たす時、$m_1 = m_2 = m_3, m_4 = m_5 = m_6$ となるので、

$$\mathrm{Ker}\partial_1 = \{\mathrm{m}_1((\mathrm{v}_1\mathrm{v}_2)+(\mathrm{v}_2\mathrm{v}_3)+(\mathrm{v}_3\mathrm{v}_1))+\mathrm{m}_4((\mathrm{v}_4\mathrm{v}_5)+(\mathrm{v}_5\mathrm{v}_6)+(\mathrm{v}_6\mathrm{v}_4))|\mathrm{m}_1, \mathrm{m}_4 \in \mathrm{Z}\} \tag{A.8}$$

となる。また、

$$\begin{aligned}
\mathrm{Im}\partial_1 &= M_0(v_2 - v_1) + M_1(v_3 - v_2) + M_2(v_1 - v_3) \\
&\quad + M_3(v_5 - v_4) + M_4(v_6 - v_5) + M_5(v_4 - v_6)
\end{aligned} \tag{A.9}$$

である。次に、$\partial_0$ はすべてが 0 になるので、

$$\mathrm{Ker}\partial_0 = \mathrm{l}_0(\mathrm{v}_1) + \mathrm{l}_1(\mathrm{v}_2) + \mathrm{l}_2(\mathrm{v}_3) + \mathrm{l}_3(\mathrm{v}_4) + \mathrm{l}_4(\mathrm{v}_5) + \mathrm{l}_5(\mathrm{v}_6) \tag{A.10}$$

となる。

以上を用いてホモロジー群を計算すると、

$$\mathrm{H}_2(K) = \mathrm{Z}_2(K)/\mathrm{B}_2(K) = \mathrm{Ker}\partial_2/\mathrm{Im}\partial_3 = 0 \tag{A.11}$$

$$\mathrm{H}_1(K) = \mathrm{Z}_1(K)/\mathrm{B}_1(K) = \mathrm{Ker}\partial_1/\mathrm{Im}\partial_2 \tag{A.12}$$

ここで、$m_1((v_1v_2) + (v_2v_3) + (v_3v_1)) \in \mathrm{Im}\partial_2$ なので、

$$\mathrm{H}_1(K) = \{m_4((v_4v_5) + (v_5v_6) + (v_6v_4))|m_4 \in Z\} \simeq Z \tag{A.13}$$

$$\mathrm{H}_0(K) = \mathrm{Z}_0(K)/\mathrm{B}_0(K) = \mathrm{Ker}\partial_0/\mathrm{Im}\partial_1 \tag{A.14}$$

ここで、$(v_1) \simeq (v_2) \simeq (v_3), (v_4) \simeq (v_5) \simeq (v_6)$ となり、

$$\mathrm{H}_0(K) = \{(l_0{+}l_1{+}l_3)(v_1){+}(l_3{+}l_4{+}l_5)(v_4) | l_0, l_1, l_2, l_3, l_4, l_5 \in Z \quad \} \simeq Z{\oplus}Z \tag{A.15}$$

以上より、$n$ 次元ベッチ数を $\beta_n$ と表すと、

$$\beta_2 = \dim(\mathrm{H}_2(K)) = 0 \tag{A.16}$$

$$\beta_1 = \dim(\mathrm{H}_1(K)) = 1 \tag{A.17}$$

$$\beta_0 = \dim(\mathrm{H}_0(K)) = 2 \tag{A.18}$$

となる。

## A.3 解2

実際にプログラムで解いている計算式 (1.19) を用いて計算する。まず、各境界準同型写像の行列表記を以下に示す。

$$\partial_3 = 0 \tag{A.19}$$

$$\partial_2 = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \tag{A.20}$$

$$\partial_1 = \begin{pmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{A.21}$$

$$\partial_0 = 0 \tag{A.22}$$

これら行列の階数を求めると、次のようになる。

$$\mathrm{rank}(\partial_3) = 0 \tag{A.23}$$

$$\mathrm{rank}(\partial_2) = 1 \tag{A.24}$$

$$\mathrm{rank}(\partial_1) = 4 \tag{A.25}$$

$$\mathrm{rank}(\partial_0) = 0 \tag{A.26}$$

となる。最後に式 (1.19) から各次元のベッチ数を求めると以下の通りになる。

$$\beta_2 = \dim C_2 - \mathrm{rank}(\partial_2) - \mathrm{rank}(\partial_3) = 1 - 1 - 0 = 0 \tag{A.27}$$

$$\beta_1 = \dim C_1 - \mathrm{rank}(\partial_1) - \mathrm{rank}(\partial_2) = 6 - 4 - 1 = 1 \tag{A.28}$$

$$\beta_0 = \dim C_0 - \mathrm{rank}(\partial_0) - \mathrm{rank}(\partial_1) = 6 - 0 - 4 = 2 \tag{A.29}$$

# 付 録B　ソースコード

## B.1　概要

ベッチ数解析プログラムは以下から構成される。

- error.h
- graph.h
- betti.h
- main.c プログラムの実行を制御する
- error.c エラーが起きた際の処理を行う
- graph.c グラフ構造に関する処理（主にI/O）を行う
- betti.c ベッチ数を解析する

以下に各コードの詳細を示す。

## B.2　error.h

```
#ifndef _ERROR_H
#define _ERROR_H

void error(char *_message);

#endif
```

## B.3　graph.h

```
/*
  グラフ構造を扱う
*/
```

```
#ifndef _GRAPH_H
#define _GRAPH_H

#include <stdio.h>

struct _graph_link
{
  unsigned int node1;
  unsigned int node2;
};

struct _graph
{
  unsigned int node_count;
  unsigned int link_count;

  struct _graph_link *links;
};

typedef struct _graph graph;

/* グラフを生成する（リンクは未定義の状態で）*/
graph* graph_create(unsigned int _node_count, unsigned int _link_count);

/* グラフをネットファイルから生成する */
graph* graph_createFromFile(char *_filename);

/* グラフを標準入力から生成する */
graph* graph_createFromInput(FILE* _fp);

/* グラフを解放する */
void graph_dispose(graph *_instance);

/* グラフの複製を生成する */
graph* graph_copy(graph *_src);

/* ノードが接続されているか調べる */
int graph_check_connection(graph *_graph, unsigned int _id1, unsigned int _id2);

/* グラフをファイルに出力する */
void graph_outputToFile(graph *_graph, char *_filename);
```

```
/* グラフを出力する */
void graph_output(graph *_graph, FILE *_fp);

#endif
```

## B.4 betti.h

```
#ifndef _BETTI_H
#define _BETTI_H

#include "graph.h"

struct _betti_result
{
  unsigned int count;
  unsigned int *clique;
  unsigned int *betti;
};

typedef struct _betti_result betti_result;

/* ベッチ数を計算する */
extern betti_result* calculate_betti(graph *_graph);

/* ベッチ数の計算結果を破棄する */
extern void betti_result_dispose(betti_result *_instance);

#endif
```

## B.5 main.c

```
#include "graph.h"
#include "betti.h"
#include "error.h"
#include <stdio.h>
#include <stdlib.h>
```

```
#pragma warning(push)
#pragma warning(disable: 4616)
#pragma warning(disable: 4996)
#pragma warning(disable: 1478)
#pragma warning(disable: 4100)

/*
  ./betti
*/
int main(int argc, char *argv[])
{
  unsigned int j;
  graph *g;
  betti_result *be;

  g = graph_createFromInput(stdin);
  if( g == NULL ) error("Can't create network.\n");

  be =  calculate_betti(g);

  for(j=0; j<be->count; ++j)
  {
    printf("%u, %u, %u\n", j, be->clique[j], be->betti[j]);
  }

  betti_result_dispose(be);
  graph_dispose(g);

  return 0;
}

#pragma warning(pop)
```

## B.6 error.c

```
#include "error.h"
#include <stdio.h>
#include <stdlib.h>
```

```
#pragma warning(push)
#pragma warning(disable: 4616)
#pragma warning(disable: 4996)
#pragma warning(disable: 1478)

void error(char *_message)
{
  int i;
  if( _message ) fprintf(stderr, "Error:%s\n", _message);
  i = 0;
  scanf("%d", &i);
  exit(1);
}

#pragma warning(pop)
```

## B.7 graph.c

```
/*
  グラフ構造を扱う
*/

#include "graph.h"
#include "error.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#pragma warning(push)
#pragma warning(disable: 4616)
#pragma warning(disable: 4996)
#pragma warning(disable: 1478)

#define GRAPH_CHAR_BUFFER_SIZE 512

graph* graph_create(unsigned int _node_count, unsigned int _link_count)
{
  graph *instance;
  unsigned int i;
```

```
  instance = (graph*)malloc(sizeof(graph));
  if( instance == NULL ) return NULL;

  instance->node_count = _node_count;
  instance->link_count = _link_count;
  instance->links = (struct _graph_link*)calloc(instance->link_count,
                    sizeof(struct _graph_link));
  if( instance->links == NULL ) error("Memory error. @ graph creating.");

  for(i=0; i<_link_count; ++i)
  {
    instance->links[i].node1 = 0;
    instance->links[i].node1 = 1;
  }

  return instance;
}

graph* graph_createFromFile(char *_filename)
{
  FILE *fp;
  graph *instance;

  /* ファイルを読み込む */
  fp = fopen(_filename, "r");
  if( fp == NULL ) return NULL;

  instance = graph_createFromInput(fp);

  fclose(fp);

  return instance;
}

graph* graph_createFromInput(FILE* _fp)
{
  char sbuf[GRAPH_CHAR_BUFFER_SIZE];
  graph *instance;
  unsigned int linki;

  instance = (graph*)malloc(sizeof(graph));
```

```
if( instance == NULL ) error("Memory error. @ graph creating.");

instance->node_count = 0;
instance->link_count = 0;
instance->links = NULL;

/* リンクの先頭まで読み込む */
while( fgets(sbuf, GRAPH_CHAR_BUFFER_SIZE, _fp) != NULL)
{
  char sbuf2[GRAPH_CHAR_BUFFER_SIZE];
  int ibuf;

  if( sscanf(sbuf, "%s %d", sbuf2, &ibuf) != 2) continue;

  if( strcmp(sbuf2, "*Vertices") == 0 )
  {
    instance->node_count = ibuf;
  }
  else if(strcmp(sbuf2, "*Edges") == 0)
  {
    instance->link_count = ibuf;
    break;
  }
}

instance->links = (struct _graph_link*)calloc(instance->link_count,
                  sizeof(struct _graph_link));
if( instance->links == NULL ) error("Memory error2. @ graph creating.");

/* リンクを読み込み */
linki = 0;
while( fgets(sbuf, GRAPH_CHAR_BUFFER_SIZE, _fp) != NULL)
{
  int ibuf1, ibuf2;

  if( sscanf(sbuf, "%d %d", &ibuf1, &ibuf2) != 2 ) continue;

  if( ibuf1 < ibuf2)
  {
    instance->links[linki].node1 = ibuf1 - 1;
    instance->links[linki].node2 = ibuf2 - 1;
```

```
    }
    else
    {
      instance->links[linki].node1 = ibuf2 - 1;
      instance->links[linki].node2 = ibuf1 - 1;
    }
    ++linki;
  }

  return instance;
}

void graph_dispose(graph *_instance)
{
  if( _instance )
  {
    if( _instance->links ) free(_instance->links);
    free(_instance);
  }
}

graph* graph_copy(graph *_src)
{
  graph *instance;

  if( _src == NULL ) error("Src is NULL. @ graph copying.");

  instance = (graph*)malloc(sizeof(graph));
  if( instance == NULL ) return NULL;

  instance->node_count = _src->node_count;
  instance->link_count = _src->link_count;

  instance->links = (struct _graph_link*)malloc(
                    instance->link_count * sizeof(struct _graph_link) );
  if( instance->links == NULL ) error("Memory error. @ graph copying.");

  memcpy(instance->links, _src->links,
         instance->link_count * sizeof(struct _graph_link));

  return instance;
```

```
}

int graph_check_connection(graph *_graph, unsigned int _id1, unsigned int _id2)
{
  unsigned int i;
  if( _id1 == _id2 ) return 1;

  for(i=0; i<_graph->link_count; ++i)
  {
    if( (_graph->links[i].node1 == _id1 && _graph->links[i].node2 == _id2) ||
      (_graph->links[i].node2 == _id1 && _graph->links[i].node1 == _id2) )
    {
      return 1;
    }
  }

  return 0;
}

void graph_outputToFile(graph *_graph, char *_filename)
{
  FILE *fp;

  fp = fopen(_filename, "w");
  if( fp == NULL ) error("File open error. @ graph saving.");

  graph_output(_graph, fp);

  fclose(fp);
}

/* グラフを出力する */
void graph_output(graph *_graph, FILE *_fp)
{
  unsigned int i;

  fprintf(_fp, "*Vertices %u\n", _graph->node_count);
  for(i=0; i<_graph->node_count; ++i)
  {
    fprintf(_fp, "%u\n", i + 1);
  }
```

```
  fprintf(_fp, "*Edges %u\n", _graph->link_count);
  for(i=0; i<_graph->link_count; ++i)
  {
    fprintf(_fp, "%u %u\n", _graph->links[i].node1 + 1,
            _graph->links[i].node2 + 1);
  }
}

#pragma warning(pop)
```

## B.8 betti.c

```
/*
  ベッチ数を計算する。
*/
#include "betti.h"
#include "error.h"
#include <stdlib.h>
#include <string.h>
#include <math.h>

#pragma warning(push)
#pragma warning(disable: 4616)
#pragma warning(disable: 4996)
#pragma warning(disable: 1478)

#define THRESHOLD 0.0000001

/*
  構造体宣言
*/

struct ret_list_node
{
  unsigned int clique;
  unsigned int betti;
  struct ret_list_node *next;
};
```

```
struct ret_list
{
  unsigned int count;
  struct ret_list_node *top;
  struct ret_list_node *last;
};

struct es_list_node
{
  unsigned int *ids;
  struct es_list_node *next;
};

struct es_list
{
  unsigned int count;
  unsigned int dimension;
  struct es_list_node *top;
  struct es_list_node *last;
};

struct tmatrix_row_node
{
  unsigned int row_index;
  double value;
  struct tmatrix_row_node *next;
  struct tmatrix_row_node *back;
};

struct tmatrix_colmun_node
{
  struct tmatrix_row_node *top;
};

struct tmatrix
{
  unsigned int colmun_count;
  struct tmatrix_colmun_node *colmuns;
};

/*
```

```
  スタティック関数　プロトタイプ宣言
*/

/* 結果リスト */
static struct ret_list* ret_list_create(void);
static void ret_list_dispose(struct ret_list *_instance);
static void ret_list_add(struct ret_list *_instance,
                         unsigned int _clique, unsigned int _betti);

/* 単体リスト */
static struct es_list* es_list_create(int _dimension);
static void es_list_dispose(struct es_list* _instance);
static void es_list_add(struct es_list *_instance, unsigned int *_ids);
static int es_node_comparer(const void *_v1, const void *_v2);

/* 推移行列 */
static struct tmatrix* tmatrix_create(unsigned int _col_count);
static void tmatrix_dispose(struct tmatrix *_instance);
static void tmatrix_add(struct tmatrix *_instance, unsigned int _col,
                        unsigned int _row, double _value);
static void tmatrix_add_row(struct tmatrix *_instance, unsigned int _dest_row,
                            unsigned int _src_row, double _factor);
static struct tmatrix_row_node* tmatrix_get(struct tmatrix *_instance,
                                        unsigned int _col, unsigned int _row);
static void tmatrix_delete(struct tmatrix *_instance, unsigned int _col,
                           struct tmatrix_row_node *_node);
static void tmatrix_delete_row(struct tmatrix *_instance, unsigned _row);

/* 単体を検出する */
static struct es_list* element_search(graph *_graph, unsigned int _dimension);
static unsigned int* nextNodes(graph *_graph, unsigned int _id,
                               unsigned int *_count);
static void recursion_check(struct es_list *_result, graph *_graph,
              unsigned int _id,
              unsigned int *_tbuffer, unsigned int _tbuffer_size,
              unsigned int *_nbuffer, unsigned int _nbuffer_size,
              unsigned int _now_index, unsigned int _rest_count
              );

/* d → d-1 の推移行列を作成する */
static struct tmatrix* create_translate_matrix(graph* _graph,
```

```
                                                    unsigned int _dimension);
static int element_comparer(unsigned int *_elm1,
                            unsigned int *_elm2,
                            unsigned int _size);

/*
 階数を求める
 元の行列は破壊される
*/
static unsigned int calculate_rank(struct tmatrix *_matrix);

/*
  グローバル関数
*/
betti_result* calculate_betti(graph *_graph)
{
  betti_result *result;
  struct ret_list *rets;
  unsigned int dim;
  unsigned int prev_clique;
  unsigned int prev_bi;

  result = (betti_result*)malloc(sizeof(betti_result));
  if( result == NULL ) error("Memory error. @ betti calculating.");
  result->clique = NULL;
  result->betti = NULL;

  /* 結果リスト作成 */
  rets = ret_list_create();
  dim = 0;

  prev_clique = _graph->node_count;

  prev_bi = 0;
  while( prev_clique )
  {
    unsigned int betti, clique;
    unsigned int bi;
    struct tmatrix *mat;

    betti = 0;
```

```
  clique = prev_clique;

  /* 遷移行列生成 */
  mat = create_translate_matrix(_graph, dim + 1);

  prev_clique = mat->colmun_count;

  bi = calculate_rank(mat);

  tmatrix_dispose(mat);

  betti = clique - prev_bi - bi;

  /* 結果追加 */
   ret_list_add(rets, clique, betti);

  prev_bi = bi;
  ++dim;
}

/* 結果リストから結果を生成。*/
result->count = rets->count;
if( result->count > 0 )
{
  struct ret_list_node *node;
  unsigned int index;

  result->betti = (unsigned int*)malloc( rets->count * sizeof(unsigned int) );
  if( result->betti == NULL ) error("Memory error. @ betti calculating.");
  result->clique = (unsigned int*)malloc( rets->count * sizeof(unsigned int) );
  if( result->clique == NULL ) error("Memory error. @ betti calculating.");

  node = rets->top;
  index = 0;
  while(node)
  {
    result->clique[index] = node->clique;
    result->betti[index] = node->betti;

    node = node->next;
    ++index;
```

```
    }
  }
  ret_list_dispose(rets);

  return result;
}

void betti_result_dispose(betti_result *_instance)
{
  if( _instance )
  {
    if( _instance->clique ) free(_instance->clique);
    if( _instance->betti ) free(_instance->betti);
    free(_instance);
  }
}

/*
  スタティック関数
*/

static struct ret_list* ret_list_create()
{
  struct ret_list *ins;

  ins = (struct ret_list*)malloc(sizeof(struct ret_list));
  if( ins == NULL ) error("Memory error. @ ret list creating.");

  ins->count = 0;
  ins->top = NULL;
  ins->last = NULL;

  return ins;
}

static void ret_list_dispose(struct ret_list *_instance)
{
  struct ret_list_node *node;

  node = _instance->top;
  while(node)
```

```
  {
    struct ret_list_node *bnode;

    bnode = node->next;
    free(node);
    node = bnode;
  }

  free(_instance);
}

static void ret_list_add(struct ret_list *_instance,
                         unsigned int _clique, unsigned int _betti)
{
  struct ret_list_node *node;

  node = (struct ret_list_node*)malloc(sizeof(struct ret_list_node));
  if( node == NULL ) error("Memory error. @ ret list adding.");
  node->clique = _clique;
  node->betti = _betti;
  node->next = NULL;

  if( _instance->top )
  {
    _instance->last->next = node;
    _instance->last = node;
  }
  else
  {
    _instance->top = node;
    _instance->last = node;
  }

  ++_instance->count;
}

static struct es_list* es_list_create(int _dimension)
{
  struct es_list* instance;

  instance = (struct es_list*)malloc(sizeof(struct es_list));
```

```
  if( instance == NULL ) error("Memory error. @ es_list creating.");

  instance->count = 0;
  instance->top = NULL;
  instance->last = NULL;
  instance->dimension = _dimension;

  return instance;
}

static void es_list_add(struct es_list *_instance, unsigned int *_ids)
{
  struct es_list_node* node;

  if( _instance == NULL ) error("Instance = NULL. @ es_list adding.");

  node = (struct es_list_node*)malloc(sizeof(struct es_list_node));
  if( node == NULL ) error("Memory error. @ es_list adding.");

  node->next = NULL;
  node->ids = (unsigned int*)
              malloc((_instance->dimension + 1) * sizeof(unsigned int) );
  if( node->ids == NULL ) error("Memory error2. @ es_list adding.");

  memcpy(node->ids, _ids, (_instance->dimension + 1) * sizeof(unsigned int));

  qsort(node->ids, (_instance->dimension + 1), sizeof(unsigned int),
        es_node_comparer);

  #pragma omp critical
  {
    ++_instance->count;
    if( _instance->top == NULL )
    {
      _instance->top = node;
      _instance->last = node;
    }
    else
    {
      _instance->last->next = node;
      _instance->last = node;
```

```
    }
  }
}

static void es_list_dispose(struct es_list *_instance)
{
  if( _instance )
  {
    struct es_list_node *node, *node2;
    node = _instance->top;
    while( node )
    {
      node2 = node->next;

      if( node->ids ) free(node->ids);
      free(node);

      node = node2;
    }

    free(_instance);
  }
}

static int es_node_comparer(const void *_v1, const void *_v2)
{
  int v1 = (int)(*((unsigned int*)_v1));
  int v2 = (int)(*((unsigned int*)_v2));
  return v1 - v2;
}

static struct tmatrix* tmatrix_create(unsigned int _col_count)
{
  struct tmatrix* ret;
  unsigned int i;

  ret = (struct tmatrix*)malloc(sizeof(struct tmatrix));
  if( ret == NULL ) error("Memory error. @ tmatrix creating.");

  ret->colmun_count = _col_count;
  ret->colmuns = (struct tmatrix_colmun_node*)
```

```
                malloc(_col_count * sizeof(struct tmatrix_colmun_node));
  if( ret->colmuns == NULL ) error("Memory error2. @ tmatrix creating.");

  for(i=0; i<_col_count; ++i)
  {
    ret->colmuns[i].top = NULL;
  }

  return ret;
}

void tmatrix_dispose(struct tmatrix *_instance)
{
  unsigned int i;
  if( _instance == NULL ) return;

  for(i=0; i<_instance->colmun_count; ++i)
  {
    struct tmatrix_row_node *node, *bnode;

    node = _instance->colmuns[i].top;
    while( node )
    {
      bnode = node->next;
      free(node);
      node = bnode;
    }
  }

  free(_instance->colmuns);
  free(_instance);
}

static void tmatrix_add(struct tmatrix *_instance,
                        unsigned int _col, unsigned int _row, double _value)
{
  struct tmatrix_row_node *rnode;
  struct tmatrix_row_node *new_node;

  if( fabs(_value) <= THRESHOLD ) return;
```

```
new_node = (struct tmatrix_row_node*)malloc(sizeof(struct tmatrix_row_node));
if( new_node == NULL ) error("Memory error. @ tmatrix adding.");

new_node->back = NULL;
new_node->next = NULL;
new_node->value = _value;
new_node->row_index = _row;

rnode = _instance->colmuns[_col].top;
if( rnode == NULL )
{
  _instance->colmuns[_col].top = new_node;
}
else
{
  while( rnode )
  {
    if( rnode->row_index == _row )
    {
      free(new_node);
      rnode->value += _value;

      if( fabs(rnode->value) <= THRESHOLD )
      {
        tmatrix_delete(_instance, _col, rnode);
      }
      break;
    }
    else if( rnode->row_index > _row )
    {
      if( rnode->back )
      {
        new_node->back = rnode->back;
        new_node->back->next = new_node;
      }
      else
      {
        _instance->colmuns[_col].top = new_node;
      }

      new_node->next = rnode;
```

```
        rnode->back = new_node;
        break;
      }
      else
      {
        if( rnode->next == NULL )
        {
          rnode->next = new_node;
          new_node->back = rnode;
          break;
        }
      }

      rnode = rnode->next;
    }
  }
}

static void tmatrix_add_row(struct tmatrix *_instance,
                            unsigned int _dest_row,
                            unsigned int _src_row,
                            double _factor)
{
  int i;
  #pragma omp parallel for
  for(i=0; i< _instance->colmun_count; ++i)
  {
    struct tmatrix_row_node *node;
    node = tmatrix_get(_instance, i, _src_row);
    if( node )
    {
      tmatrix_add(_instance, i, _dest_row, _factor * node->value);
    }
  }
}

static struct tmatrix_row_node* tmatrix_get(struct tmatrix *_instance,
                                            unsigned int _col, unsigned int _row)
{
  struct tmatrix_row_node *node;
```

```
  node = _instance->colmuns[_col].top;
  while( node )
  {
    if( node->row_index == _row ) return node;
    if( node->row_index > _row ) return NULL;

    node = node->next;
  }

  return NULL;
}

static void tmatrix_delete(struct tmatrix *_instance,
                           unsigned int _col, struct tmatrix_row_node *_node)
{
  if( _node->back )
  {
    if( _node->next)
    {
      _node->back->next = _node->next;
      _node->next->back = _node->back;
    }
    else
    {
      _node->back->next = NULL;
    }
  }
  else
  {
    if( _node->next)
    {
      _instance->colmuns[_col].top = _node->next;
      _instance->colmuns[_col].top->back = NULL;
    }
    else
    {
      _instance->colmuns[_col].top = NULL;
    }
  }

  free(_node);
```

```
}

static void tmatrix_delete_row(struct tmatrix *_instance, unsigned _row)
{
  int i;
  #pragma omp parallel for
  for(i=0; i<_instance->colmun_count; ++i)
  {
    struct tmatrix_row_node *node;

    node = tmatrix_get(_instance, i, _row);
    if( node ) tmatrix_delete(_instance, i, node);
  }
}

static struct es_list* element_search(graph *_graph, unsigned int _dimension)
{
  struct es_list *result;
  int i, icount;
  result = NULL;

  if( _graph == NULL ) error("_graph = NULL. @ element searching.");

  result = es_list_create(_dimension);

  icount = (int)_graph->node_count;
  #pragma omp parallel for
  for(i=0; i<icount; ++i)
  {
    unsigned int *nexts, ncount;

    if( _dimension == 0 )
    {
      unsigned int tmp[1];
      tmp[0] = i;
      es_list_add(result, tmp);
    }
    else
    {
      /*
       接続先のノードリストを取得
```

```
       i 点以下のノードについては考慮しない
      */
      nexts = nextNodes(_graph, i, &ncount);
      if( nexts != NULL )
      {
        if( ncount >= _dimension )
        {
          unsigned int *targets;
          targets = (unsigned int *)
                    malloc( (_dimension) * sizeof(unsigned int));
          if( targets == NULL )
                    error("Can't get targets nodes. @ element searching.");

          /*
           i 点と dimension 個の点からなる単体を検出したい
           再帰的に調べる
          */
          recursion_check(result, _graph, i, targets,
                          _dimension, nexts, ncount, 0, _dimension);
          free(targets);
        }

        /* 後処理 */
        free(nexts);
      }
    }
  }

  return result;
}

static unsigned int* nextNodes(graph *_graph,
                               unsigned int _id, unsigned int *_count)
{
  unsigned int i, icount;
  unsigned int n, ncount;
  unsigned int *nexts;
  nexts = NULL;

  /* ここで単体の検出を行う */
```

```
  /* リンクを捜査し、接続されているノード数を取得する。*/
  icount = _graph->link_count;
  ncount = 0;
  for(i=0; i<icount; ++i)
  {
    if( (_graph->links[i].node1 == _id || _graph->links[i].node2 == _id) &&
      _graph->links[i].node1 >= _id && _graph->links[i].node2 >= _id)
    {
      ++ncount;
    }
  }

  /* 一つも無ければ終了 */
  *_count = 0;
  if( ncount <= 0) return NULL;

  /* 接続先配列を作成 */
  nexts = (unsigned int*)malloc( ncount * sizeof(unsigned int));
  if( nexts == NULL ) error("Memory error. @ next nodes searching.");

  n = 0;
  for(i=0; i<icount; ++i)
  {
    if( _graph->links[i].node1 == _id && _graph->links[i].node2 > _id)
    {
      nexts[n++] = _graph->links[i].node2;
    }
    else if( _graph->links[i].node2 == _id && _graph->links[i].node1 > _id)
    {
      nexts[n++] = _graph->links[i].node1;
    }

    if( n >= ncount ) break;
  }

  *_count = ncount;
  return nexts;
}

static void recursion_check(struct es_list *_result,
              graph *_graph, unsigned int _id,
```

```
                unsigned int *_tbuffer, unsigned int _tbuffer_size,
                unsigned int *_nbuffer, unsigned int _nbuffer_size,
                unsigned int _now_index, unsigned int _rest_count
                )
{
  if( _rest_count <= 0 )
  {
    unsigned int *elm_buffer;
    /* 単体発見 */
    elm_buffer = (unsigned int*)
                 malloc((_tbuffer_size + 1) * sizeof(unsigned int));
    if( elm_buffer == NULL ) error("Memory error. @ recursion_check.");

    elm_buffer[0] = _id;
    memcpy(elm_buffer + 1, _tbuffer, _tbuffer_size * sizeof(unsigned int));

    es_list_add(_result, elm_buffer);

    free(elm_buffer);
  }
  else
  {
    unsigned int j, jcount;

    if( _rest_count > _nbuffer_size - _now_index ) return;

    recursion_check(_result, _graph, _id,
                    _tbuffer, _tbuffer_size,
                    _nbuffer, _nbuffer_size, _now_index + 1, _rest_count);

    /* クリーク状になっているか。*/
    jcount = _tbuffer_size - _rest_count;
    for(j=0; j<jcount; ++j)
    {
      if( !graph_check_connection(_graph, _tbuffer[j], _nbuffer[_now_index]))
      {
        return;
      }
    }

    /* 接続先はすべてクリーク */
```

```
    _tbuffer[ _tbuffer_size - _rest_count ] = _nbuffer[_now_index];
    recursion_check(_result, _graph, _id,
                    _tbuffer, _tbuffer_size,
                    _nbuffer, _nbuffer_size, _now_index + 1, _rest_count - 1);
  }
}

struct tmatrix* create_translate_matrix(graph* _graph, unsigned int _dimension)
{
  struct es_list *elms;
  struct es_list *elms_tmp;
  struct es_list_node *node;

  unsigned int *ibuf;
  unsigned int col;

  struct tmatrix *result;

  elms = NULL;
  elms_tmp = NULL;
  node = NULL;
  ibuf = NULL;
  result = NULL;

  if( _dimension <= 0 ) error("Illigal dimension. @ translate matrix creating.");

  elms_tmp = es_list_create(_dimension - 1);
  if( elms_tmp == NULL ) error("Memory error1. @ translate matrix creating.");

  elms = element_search(_graph, _dimension);
  result = tmatrix_create(elms->count);

  ibuf = (unsigned int*)malloc(_dimension * sizeof(unsigned int));

  col = 0;
  node = elms->top;
  while( node )
  {
    unsigned int i, j;

    for(i=0; i<elms->dimension+1; ++i)
```

```
  {
    struct es_list_node *node_tmp;
    int iflag;

    unsigned int index;
    char sig;
    sig = (char)((i & 0x1 ) ? -1 : 1);
    index = 0;
    for(j=0; j<elms->dimension+1; ++j)
    {
      if( i==j ) continue;
      ibuf[index++] = node->ids[j];
    }

    /* ibuf に仮の単体が入る */
    /* elms_tmp と比較 */
    node_tmp = elms_tmp->top;
    index = 0;
    iflag = 1;
    while( node_tmp )
    {
      if( element_comparer(node_tmp->ids, ibuf, _dimension) )
      {
        /* 同じ組み合わせ発見 */
        iflag = 0;
        break;
      }

      node_tmp = node_tmp->next;
      ++index;
    }
    if( iflag ) es_list_add(elms_tmp, ibuf);

    tmatrix_add(result, col, index, sig);
  }

  node = node->next;
  ++col;
}

free(ibuf);
```

```
  es_list_dispose(elms_tmp);
  es_list_dispose(elms);

  return result;
}

static int element_comparer(unsigned int *_elm1,
                            unsigned int *_elm2,
                            unsigned int _size)
{
  unsigned int i;
  for(i=0; i<(int)_size; ++i)
  {
    if( _elm1[i] != _elm2[i] ) return 0;
  }

  return 1;
}

static unsigned int calculate_rank(struct tmatrix *_matrix)
{
  unsigned int col;
  unsigned int rank;

  if( _matrix->colmun_count <= 0 ) return 0;

  rank = 0;
  for(col = 0; col<_matrix->colmun_count; ++col)
  {
    struct tmatrix_row_node *rnode;
    unsigned int row;
    double row_value;

    /* Col 列の調査 */
    rnode = _matrix->colmuns[col].top;
    if( rnode == NULL ) continue;

    ++rank;

    row = rnode->row_index;
    row_value = rnode->value;
```

```
    rnode = rnode->next;
    while(rnode)
    {
      struct tmatrix_row_node *tmp;
      tmp = rnode->next;

      tmatrix_add_row(_matrix, rnode->row_index, row,
                      -rnode->value / row_value);
      rnode = tmp;
    }

    /* row行は消しておく */
    tmatrix_delete_row(_matrix, row);
  }

  return rank;
}


#pragma warning(pop)
```

# 参考文献


[1] A. Barabàsi：『新ネットワーク思考 - 世界の仕組みを読み解く -』，NHK 出版 (2002) 368pp.

[2] 増田 直紀，今野 紀雄：『複雑ネットワークの科学』産業図書 (2007) 182pp.

[3] 増田 誠也：複雑ネットワークの生成と解析，九州工業大学情報工学部電子情報工学科卒業論文 (2007).

[4] 瀬山 士郎：『トポロジー：柔らかい幾何学 [増補版]』，日本評論社 (2005) 221pp.

[5] 田中 利史，村上 斉：『トポロジー入門』，サイエンス社 (2005) 200pp.

[6] 廣田 祐二：位相不変量を用いたネットワークの解析，九州工業大学情報工学部電子情報工学科卒業論文 (2005).

[7] 大沢 慶祐：種々のスケールフリーネットワークの位相不変量を用いた解析，九州工業大学情報工学部電子情報工学科卒業論文 (2006).

[8] Y. Saad：Iterative Methods for Sparse Linear Systems，PWS, Boston, 1996.

[9] 「 Networks / Pajek 」，http://vlado.fmf.uni-lj.si/pub/networks/pajek/，(2008/2/26 アクセス)

# 謝辞

本研究を行うにあたり、多大な御指導及び御助言を賜りました小田部荘司準教授、松下照男教授、木内優助教、また有明高専電子情報工学科 松野哲也助教授に深く感謝を致します。

また、様々な支援を頂いた小田部研究室、松下研究室の皆様に感謝の意を表します。