修士論文

自己修復で頑健なネットワークに再構築する分散アルゴリズムの提案

KIM JAEHO

主指導教員 林 幸雄

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和３年３月

Abstract

Many infrastructures in real world have been known to be represented by Scale-free(SF) networks. However, in a SF network, malicious attacks to a few hubs break down the connectivity easily. Furthermore, these networks are also threatened by frequently occurred natural large disasters. To overcome these problems, we propose a self-healing method based on enhancing loops for improving robustness. Enhancing loops is adopted because of the following recent works. Network dismantling and decycling problems are asymptotically equivalent. In addition, for finding the minimal set of nodes for spreading information to the whole network, the optimal influencers are defined as the set whose removals cause the fragmentation into many local clusters. These facts indicate that removing all loops make the network into tree structure which lead to be easily fragmented by removing articulation nodes. Therefore, it is important for making robust networks to maintain as many loops as possible. In fact, enhancing loops is strongly related to make onion-like structure with the optimal robustness of connectivity. For example, enhancing loops by intermediate attachments with selecting low degree nodes is effective to make the onion-like structure. Unlike SF network, onion-like network with positive degree-degree correlation is robust against malicious attacks as well as random failures.

In recent years, several healing methods have been proposed. In a distributed local healing, the most damaged node has new connection with randomly selected node in the neighbors of attacked nodes. In this healing, the number of healing links are controllable according to the damage. However, the selection of healing nodes is heuristic in order of the most damaged ones. Another is the healing method on interdependent two lattices. When adding healing links, the candidate of new connections is expanded beyond nearest neighbors to maintain whole connectivity except failed components.

Such expanding of connection range is important for our method, in which healing links are added to the network after removal of high degree nodes. The neighbors of attacked nodes lose a part of their links, while the unbroken links of removed nodes can be reused. Therefore, some links emanated from removed nodes are reused as healing links between the extended neighbors. We consider that the number of reusable links depends on situations of damage.

To expand the connection range of healing link, each node transfers information to other nodes in three hops. If a node can communicate within two hops, new connections are created only between nearest neighbors. However, by communicating other nodes in three hops, candidates of healing links could be extended over the nearest neighbors in distributed manner.

We emphasize enhancing loops in the following self-healing method. First, a

ring as the simplest loop is created within extended neighbors to maintain the larger connectivity. In case of incomplete rings without enough healing links, the order of linking is decreasing order of its connected component size to obtain larger connectivity after healing. Next, if healing links are residual after making rings, loops on the rings are enhanced between low degree nodes.

We numerically evaluate our proposed method in comparison with the conventional self-healing method in implementing sequential algorithms. For several infrastructure networks, such as flight routes, power grid, and Internet, we consider the three measures: the connectivity, the robustness of connectivity and efficiency of paths. In addition, we discuss the number of additional ports as the resource of healing.

Through numerical simulations, we obtain the following results.

- The reconstructed networks by our proposed method maintains the higher connectivity than ones by the conventional method.
- In case of sufficiently using healing links, it obtains both higher robustness and efficiency than the conventional method.
- However, a larger number of additional ports is required than that in the conventional method.

The sequential algorithm is not always practicable because of time delay for transferring and cost of communication paths between center and other nodes, etc. Therefore, we describe a distributed algorithm for applying our self-healing on real systems. On the asynchronous system, our distributed algorithm consists of five phases. The outline of algorithm is as follows.

1. After the initiation at a node (healing node) that detect the malfunction nodes, each healing node sends messages of notification to other healing nodes in three hops directly. Through interacting with other healing nodes in three hops by sending the messages, each healing node knows the healing node's IDs over the three hops and makes the temporary routing table to communicate further healing nodes.
2. To make a ring, the order of connecting ring is determined by the size of the connected components. Therefore, each healing node uses a delivery tree to know the size of its component. After knowing the size, in order to reduce the number of required messages such as in floodings, a leader node is decided. A role of leader is determining the order of ring and requesting to other healing nodes for new connections to make a ring.

3. Each healing node sends the size of component to a leader. A leader decides the order of connecting ring by received information of component sizes. For making a ring, a leader sends requesting message to others healing nodes. The healing nodes received the message make new connections.

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究背景

航空、電力、通信、道路などのインフラは、点 (ノード) と線 (リンク) の集合であるネットワークに単純化して表現でき、現実の多くのネットワークは驚くほど共通した繋がり方としてスケールフリー構造を持っていることが今世紀初頭頃に明らかになった [1]。スケールフリーネットワークには非常に多くのリンクを持つハブノードが少数ながら存在する。ところが、ハブを狙って攻撃するとネットワークの連結性はすぐに崩壊してしまう大きな欠点を持っている。一方、インフラネットワークは台風やテロのような種々の災害からも被害を受けやすい。例えば、電力網で、2 千万人以上の被害があった停電事故は 2000 年から総計 16 件であり、航空網では夏に頻繁な台風による運航遅延率と欠航率が高い。これらの被害は全て金銭的かつ人的損害に繋がる。

こうした問題を克服するために、レジリエンスに基づいたシステム設計が近年注目を集めている。レジリエンスとは危機を含めた変化に対して基本的な機能を維持するための適応力あるいは復活力である [2]。例えば、レジリエンスがあるシステム設計として、元の状態へ速く戻ることや危機が来る前に予め対処するためのバッファーを備えることが考えられる。しかしながら、現実のネットワークを復元しても構造的に弱いスケールフリーのままであり、バックアップ切り替えを持つ設計では冗長なシステムとして複雑になってしまい、別の問題が発生する。また、今回の Covid19 のような予測不可能な危機が来ると、それに合わせてシステムも変化する必要がある。このように危機を変化のチャンスと見なして、これまで暗黙に前提されてきた復元から脱却して、より頑健なネットワークに再構築すべきことに本研究では特に注目する。ここで、頑健性とは攻撃に対する結合耐性である。攻撃はネットワークから重要なノードを狙って削除することである。スケールフリーネットワークで連結性はハブに強く依存して、ハブ攻撃でノードが素早く孤立する。ネットワーク機能を保つ連結性を維持することは非常に重要であるので、上記の脆弱性の問題を克服するために、攻撃を受けたネットワークが頑健な構造を持つように再構築することを考える。

一方、ネットワークの頑健性はループ構造と強い関連性があることが近年徐々に明らかになりつつある。例えば、ネットワークを破壊する最適の方法がループを削除することと漸近的に同等であり [3]、ネットワークを作るための逐次成長法の中でループを強化することで、攻撃に最適耐性を持つ玉葱状ネットワークが構

築できている [4]。

1.2 研究目的

本研究では、ループ強化と頑健性増加の関連性に着目して、頑健なネットワークに再構築するための自己修復法を提案する。提案した手法の有効性を確認するために、従来提案された自己修復法 [5] と数値シミュレーションを通じて比較を行う。その際、いくつのインフラネットワークの実データに適用して、修復したネットワークの連結性、頑健性、経路効率を測る。また、上記の提案法を自律分散アルゴリズムで実装できるよう記述する。

1.3 本論の構成

本論文の構成を以下に示す。

第二章 ネットワークの基本的な用語、頑健性とループの関連性、本研究と関係がある従来研究、修復法の有効性の確認に用いる分析指標 (連結性、頑健性、経路効率) について述べる。

第三章 ループ強化に基づいたネットワーク修復法の概要について述べる。

第四章 5 つのインフラネットワークの実データに対する、提案法と従来法を適用した結果を示す。連結性、頑健性、経路効率を比べて、提案法の有効性について述べる。また、修復に必要な追加ポート数を述べる。

第五章 本研究で提案する修復法に対する、自律分散アルゴリズムとしての記述を紹介する。

第六章 本研究の結論をまとめる。

第2章　関連研究

ネットワークは、ノード (点) 集合 $V = \{v_1, v_2, ..., v_N\}$ とそれらの間を繋ぐリンク (線) 集合 $E = \{e_{ij}; 1 \leq i, j \leq N, i \neq j\}$ で構成するグラフ $G \equiv G(V, E)$ で表現される。ここで、リンクの両端が同じ自己ループと、ノード間に複数のリンクがある多重辺を許さない。また、リンクに重みと方向がない単純グラフを本研究で扱うネットワークとして考える。以下、主な分析指標と提案方法の基本となるループ強化について説明する。

2.1　ネットワークの主な分析指標

ノード総数を N 個として、ネットワークの連結関係を $N \times N$ の行列で表す。この行列は隣接行列 A と呼ばれ、各成分は次のように定義される。

$$A_{ij} = \begin{cases} 0(\text{ノード } i, j \text{ 間にリンク無し}) \\ 1(\text{ノード } i, j \text{ 間にリンク有り}) \end{cases} \tag{2.1}$$

従って、ノード i が持つリンク数 (次数 k_i) は次のようになる。

$$k_i = \sum_{j=1}^{N} A_{ij} \tag{2.2}$$

スケールフリーネットワークではノードの次数分布がべき乗則 $P(k) \sim k^{-\gamma}$ に従う。ゆえに、ごく少数だが多くのリンクを持つノードが存在する。このノードはハブと呼ばれ、ハブはネットワーク全体の連結性に強く関わる。

2.1.1　最大連結成分

ハブノードを集中的に削除すると、スケールフリーネットワークはすぐに多数の連結成分に分断される。連結成分は全てのペアノード間に経路が存在する部分グラフである。その中で、一番大きい連結成分を最大連結成分という。本研究では修復したネットワークの連結性を測るために、最大連結成分の大きさの比率 $S_q(q)$ を用いる。

$$S_q(q) = \frac{S(q)}{(1-q)N} \tag{2.3}$$

ここで、N はノード攻撃により機能停止となって削除される前の元々のノード数、q は削除したノード数の割合 (攻撃率)、$S(q)$ は攻撃率 q における最大連結成分の大きさ (そこに含まれるノード数) である。q の範囲は $[0, 1]$ である。

2.1.2 頑健性

修復したネットワークの更なる攻撃に対する耐性を測るために、頑健性指標 R を用いる [6]。R はネットワークでノードを一つずつ削除するたびに、連結性が維持している程度を観る。

$$R = \frac{1}{N} \sum_{q=\frac{1}{N}}^{1} S_q(q) \tag{2.4}$$

ここで、$\sum$ は全ての攻撃率 q に対する $S_q(q)$ の和を意味しており、R 値の範囲は $\frac{1}{N} \leq R \leq 0.5$ になる。この値が大きい程、ネットワーク上でノードを削除しても、残りのノードがお互い連結されて頑健であると言える。また、本研究でノードを削除する攻撃は、次数が一番大きいノードを順に選び、ノードの次数を再計算しながら削除する次数順攻撃とする。

2.1.3 ネットワーク経路の効率

ネットワークにおけるノード同士はお互い物や情報などを交換すると考えられる。その際、ノード間の経路の長さが短いほど、コストや時間が掛からない効率的な交換ができる。本研究では、修復したネットワークの経路効率を指標 E を用いて測る。

$$E = \frac{1}{N(N-1)} \sum_{i,j=1;i\neq j}^{N} \frac{1}{d_{ij}} \tag{2.5}$$

ここで、d_{ij} はノード i と j 間の距離 (最小のホップ数) である。E 値の範囲は $0 \leq E \leq 1$ であり、この値が大きい程、ノード間の平均距離が短いことを意味する。

2.2 ループと頑健性

本研究では、頑健なネットワークを再構築するためにループ強化に基づいた自己修復法を提案する。ループと頑健性の関連性について以下に述べる。まず、最大連結成分の除去とループ削除が漸近的に同価であることを示した研究がる [3]。この研究では、ネットワーク G が持つ最大連結成分の大きさを、定数 C より小さくするために削除する必要があるノード数の最小比率 $\theta_{dis}(G, C)$ とネットワーク G から全てのループ構造を除去するために削除する必要があるノード数の最小比率

$\theta_{dec}(G)$ を定義した。このパラメーターは次数分布 $q = \{q_k\}_{k\geq 0}$ を持つランダムグラフで、式 (2.6)(2.7) で定義される。

$$\theta_{dec}(q_k) = \lim_{N\to\infty} E[\theta_{dec}(G)] \tag{2.6}$$

$$\theta_{dis}(q_k) = \lim_{C\to\infty} \lim_{N\to\infty} E[\theta_{dis}(G, C)] \tag{2.7}$$

ここで、$E[\sim]$ は次数分布が $\{q_k\}$ であるランダムグラフ集団における平均値である。両パラメーターは任意の次数分布では、$\theta_{dis}(q_k) \leq \theta_{dec}(q_k)$ であるが、$\langle k^2 \rangle < \infty$ の条件を満たす時、$\theta_{dis}(q_k) = \theta_{dec}(q_k)$ であることを示した。

また、拡散の要となるインフルエンサーを探す研究 [7] では、インフルエンサーをネットワークの最大連結成分を除去するために必要なノードだと定義して、最適のインフルエンサー集合を求めた。大きさが N であるネットワークのノードを集合 $(n_1, n_2, ..., n_N)$ における。$n_i = 0$ であるノード i は削除されたノードで、$n_i = 1$ は生きているノードとする。また、ベクトル $\vec{v} = (v_1, v_2, ..., v_N)$ の、$v_i = 0$ であるノード i は最大連結成分に属しないノードで、$v_i = 1$ は最大連結成分に属するノードとする。最大連結成分の大きさは $\vec{v}$ の要素を全部足した値である。$\vec{n}$ と $\vec{v}$ の関係をメッセージ伝搬式を用いて表すと、次のようになる。

$$v_{i\to j} = n_i[1 - \prod_{k\in\partial i\setminus j} (1 - v_{k\to j})] \tag{2.8}$$

ここで、$\partial i \setminus j$ はノード i の最近接からノード j を外したノード集合である。上記の解の安定生は $v_{i\to j}$ を線形変換に近似したヤコビ行列の最大固有値 $\lambda(\vec{n}; q) < 1$ であることから、最大固有値 $\lambda(\vec{n}; q)$ が最小化すること。q はネットワークから削除したノード数の比率である。すなわち、最適のインフルエンサーはループを構成するノード集合となる。

上記の研究では、ネットワークからループを除去すると、効果的にネットワークが崩壊されることを示した。ネットワークからループを全部削除すると木構造になるが、木構造は少数の節ノードを削除することで連結成分がすぐにバラバラになる構造であるので (図 2.1)、ループを消すことはネットワークの連結性に致命的な攻撃である。ゆえに、ネットワークが高い頑健性を保つためには、ループ構造が重要である。実際、頑健なネットワークを作るために、ループ強化に着目した研究が進められてきた。ネットワークを作るための逐次成長法の中でループを強化することで、攻撃に最適耐性を持つ玉葱状ネットワークが構築できる [4]。ここで、玉葱状ネットワークは中心部は次数が高いノードがお互い連結され、その中心を次数が低いノードが幾層にも囲むことで可視化される。各層は次数が同じノード同士が繋がっていて、いわゆる正の次数相関を持つ。正の次数相関は高い頑健性と関係がある [8]。また、ループを作るために必要不可欠なノードを計算して、そのノード間を繋ぐようにリンクを張り替えてループを強化することで、頑健なネットワークにする研究もある [9]。

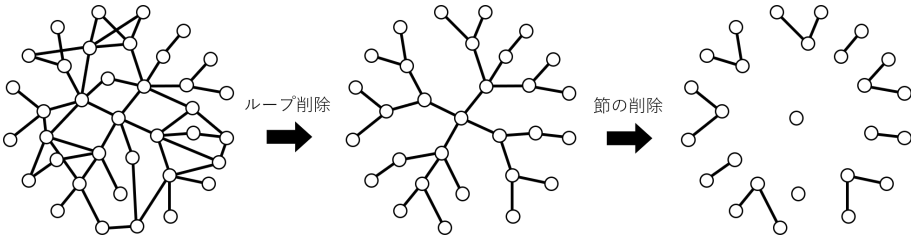

図 2.1: ループ削除によるネットワーク構造変化。ネットワークでループを全部削除すると木構造になる。また、節ノードを削除すると木構造はバラバラになる

2.3 従来のネットワーク修復研究

災害や攻撃で壊れたネットワークの機能を回復するために、これまで種々の修復法が提案されてきた。本節では、本研究と関連性がある修復法を紹介する。

まずは被害が大きいノードを修復するよう、局所的な修復法が考えられた [5]。この方法では、ノードが受けた被害を攻撃後の次数 $k_{damaged}$ と攻撃前の次数 $k_{original}$ の比 $\frac{k_{damaged}}{k_{original}}$ で表して、この比が低いノードは攻撃で多くのリンクを失ったノードであるので、被害が大きいノードに修復リンクを付与して、攻撃以前のネットワーク上で 2 ホップ離れていたノードをランダムに選択して、新しい連結を作る。すなわち、修復リンクは削除されたノードの最近接ノード間に生成される。修復に使われるリンク数は閾値 q_c を決めて、$\frac{k_{damaged}}{k_{original}} \leq q_c$ であるノードのみ修復することで、リンク数を調節する。本論文で、提案する修復法が、このヒューリスティックな手法より頑健なネットワークを構築できることを後述する。

また、正方格子上での修復法として、削除したノードの拡張近接間をランダムに連結する方法がある [10]。ノードを一つずつ削除するたびに、削除ノードの拡張近接間を確率 w で連結させ、連結成分が分断しないまで拡張が繰り返される。この方法では、一部の孤立したノードを除いた全ての連結性を維持できる。これは修復を進めながら、修復で追加されるリンクの範囲が広くなり、遠く離れていたノード間を連結するからである。このような修復範囲の拡張は正方格子だけではなく、一般的なネットワークに適用できる余地があり、後述する提案方法に取り入れている。

本研究で提案する方法と同じく、拡張した修復範囲に輪形成して、輪上のループを強化する研究がある [11]。この方法ではまず、NP 困難である、ループに必要不可欠なノード集合を統計物理に基づいた近似解法で求める。この近似計算で、ネットワークにあるノード i がその集合に属する確率 (q_i^0) を得る。q_i^0 を用いて修復を行う。ループ強化するために、q_i^0 が小さいノード間を連結する。つまり、$q_i^0 + q_j^0$ が一番小さいノード i と j を選んで繋ぐ。修復対象になるノードは、機能停止になって削除されたノードの最近接の中で選ぶ。このような修復で、q_i^0 が小さいノード間から新しいループが形成することで、ループに必要不可欠なノード集合の数を

増える。また、輪を形成する順は、最初に削除されたノードの最近接間から連結を始めて、次に削除されたノードの最近接に修復範囲を拡張する。最後に削除されたノードの最近接まで修復範囲を拡張して、拡張した範囲にある修復対象を一つの輪で連結することで、高い連結性を持つようにする。輪を作った後、修復リンク数に余分があれば、余分の修復リンク数分だけ輪上に $q_i^0 + q_j^0$ が一番小さいノード間を連結してループを強化する。本研究で提案する方法が修復範囲の拡張と輪形成に着目したことは、この方法と同じであるが、範囲を拡張する方法と、ループ強化のためにノードを選択する基準が相当に異なる。それを第三章で後述する。

第3章　提案手法

本章では頑健なネットワークに再構築する自己修復法の概要を述べる。

ネットワークの修復法は、基本的に攻撃や故障の後、リンクを失って機能不全となったノードを修復対象にして、それの間に修復リンクを追加する手法である。修復の際には、攻撃で機能停止したノードが持っていたリンクの一部を再利用する。被害状況によって、そのリンクの状態も異なるから、一部だけを使えると想定する。ゆえに、再利用できるリンク数 (r) を以下で定義する。

$$r = \alpha \cdot \overset{\sim}{\sum_{i}} k_i \tag{3.1}$$

ここで、k_i は機能停止したノード i の次数である。しかしながら、その故障ノードと他の故障ノード間にあるリンクを重複して数えないように計算する。ゆえに、$\sum_i^{\sim} k_i$ の $\sim$ は重複を外した計算を意味する。また、α は使えるリンクの割合を表すパラメーターで、α の範囲は $(0, 1]$ とする。

また、局所分散アルゴリズムによる実現を考えて、各修復対象は予め 3 ホップまで情報を送ることが出来るとして、集めた情報で修復リンクの範囲を拡張できるようにする。ゆえに、修復は各拡張した修復範囲ごとにループを加えて強化するように局所的に行い、r も各修復範囲ごとに計算する。ゆえにその値は、同じ修復範囲にあるノードが検知した故障ノードの次数の和になり、詳しい計算方法は以降に述べる。修復法は 3 ステップで構成され、以下のように行う。

Step1 各修復対象は 3 ホップ内まで情報を交換するようにして、修復リンクは追加される範囲を拡張する。例えば図 3.1(左) のように、2 ホップまで直接通信できると、修復リンクが追加される範囲が各機能停止したノードの最近接間である。しかしながら、ネットワークの被害状況によって、その範囲が重ねないことで分断されたネットワークを一つに連結できない場合がある。実際、図 3.1(左) で修復リンクは各灰色円中に生成されて、ネットワークが一つにならない。ゆえに、3 ホップの通信なら第 2 隣接との連結ができ、第 2 隣接を中継して第 3 や第 4 隣接へ修復リンクの連結範囲を拡張できる。例えば、図 3.1(右) でノード 4 はノード 2 を通じてノード 3,6,7 と通信できるから、遠く離れていたノードと連結できるようになる。こうして拡張した修復リンクの範囲内で、同じ拡張隣接にあるノードはお互い通信できる。

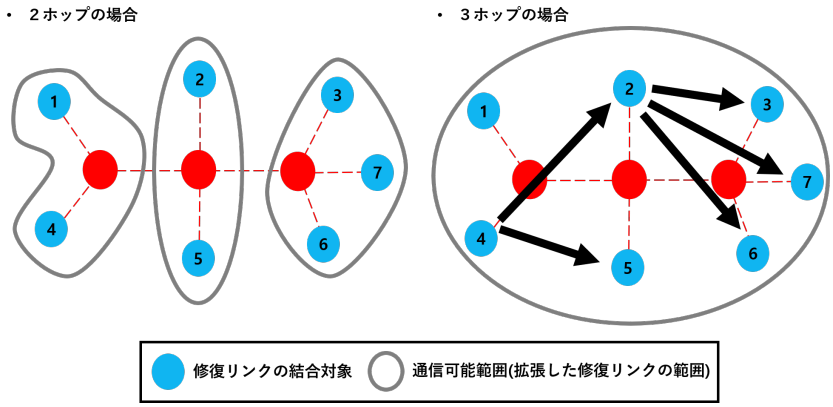

図 3.1: 通信可能な距離による差。(左) 2 ホップが修復リンクの範囲だと、ネットワークを一つに連結できない。(右) 3 ホップなら、他のノードを中継することで修復リンクの範囲を広げる。遠くにあるノード間を連結することで、高い連結性を維持できる

Step2 図 3.2 の緑線のように、拡張した修復リンクの追加範囲 (拡張隣接) のノードに輪を形成する。一般に、ループに必要不可欠な最小のノードを探し出すことは NP 困難な問題であるが、最も少ないリンク数で形成できるループとしての輪を考える。輪形成は各拡張隣接ごとに行い、ネットワークには複数の局所輪が作られる。

例えば、図 3.2 には 2 つの拡張隣接がある。拡張隣接 1 はノード 1 ～ 7 のノード集合であり、拡張隣接 2 はノード 8 ～ 12 のノード集合である。同じ拡張隣接にあるノード同士は互いを中継して通信できる。各拡張隣接ごとの修復リンク数は、拡張隣接のノードの最近接にある故障ノードの次数に基づく。つまり、拡張隣接 1 で使える修復リンク数 r_1 は、ノード 1 ～ 7 の最近接にある故障ノード A,B,C の次数和を割合である。割合 α が 1.0 なら、重複リンクを除いて計算した修復リンク数は 9 本になる。また、拡張隣接 2 で使える修復リンク数 r_2 は故障ノード D,E の次数和の割合であり、割合 α が 1.0 なら、$r_2 = 6$ になる。これを用いて各拡張隣接で輪を作る。但し、割合 α が小さくて再利用できるリンク数 r が少ない場合、拡張隣接にある全てのノードを輪で繋げない場合がある。その時できるだけ高い連結性を得るために、輪を形成する順序をノードが属している連結成分サイズの大きさに従うことにする。

Step3 Step2 で輪を形成した後、再利用できるリンク数 r の残り分を使って各輪上の次数が低いノード間にショートカットを追加して、ループを強化する (図 3.2 の黄線)。このショートカットでネットワークが正の次数相関を持つようにする。

例えば、図 3.2 で割合 α が 1.0 の時、拡張隣接 1 に輪を作るために 7 本使った

から、残った修復リンク数は 2 本である。ゆえに、拡張隣接 1 の中にショートカット 2 つを追加する。また、拡張隣接 2 では輪を作るために 5 本使ったので、残り 1 本を用いてショートカットを追加する。

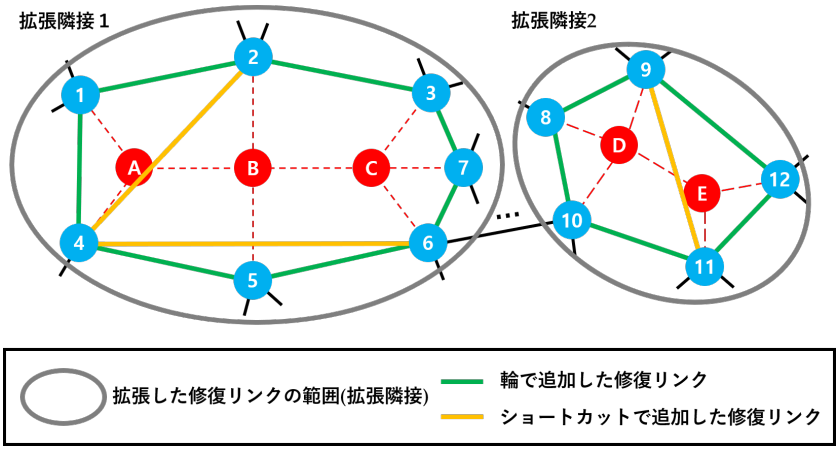

図 3.2: 提案法で修復したネットワークの結果図。Step2 では各拡張隣接のノード間に輪を形成する。Step3 では各輪上にショートカットを追加する。修復リンクの結合対象: 青ノード、故障ノード: 赤ノード、故障ノードのリンク: 赤点線、輪: 緑線、ショートカット: 黄線

第4章　実データに対する自己修復法の実験と評価

以下に示すインフラネットワークの実データに対して、提案した修復法と既存の自己修復法 [5] について、2.2 節の指標を用いて比較する。対象になるネットワークは公開されている現実のインフラネットワークである。4.1 節は対象ネットワークの詳細、4.2 節は各ネットワークに対する指標の比較結果を示す。4.3 節は修復に必要なポート数について議論する。提案法の各数値は順次的なアルゴリズムで実装したものである。

4.1　実ネットワークの詳細

- AirTraffic: アメリカ連邦航空局 (FAA) 所属 NFDC(National Flight Data Center) の航空ルート情報で作られたネットワーク。
  (http://konect.cc/networks/maayan-faa/)
- ASOregon: アメリカオレゴン大学 (University of Oregon) で進めたプロジェクト (Route Views Project) で集めた AS(autonomous systems) レベルでのインターネットの構造。
  (http://konect.cc/networks/dimacs10-as-22july06/)
- OpenFlight: 全世界の空港とそれら間の流動量を現している Openflights.org から出したネットワーク。
  (http://konect.cc/networks/opsahl-openflights/)
- PowerGrid: アメリカ西部の電力発電所のネットワーク。
  (http://konect.cc/networks/opsahl-powergrid/)
- USAirport: アメリカで流動量が多い順で上位 500 個の空港のネットワーク。
  (https://toreopsahl.com/datasets/usairports)

表 4.1 は無向グラフにした実ネットワークの基本特性を示す。

ネットワーク	総ノード数	総リンク数	平均次数	最大次数	最小次数	直経
AirTraffic	1226	2408	3.9	34	1	17
ASOregon	6474	12572	3.9	1458	1	9
OpenFlight	2905	15645	10.8	242	1	14
PowerGrid	4941	6594	2.7	19	1	46
USAirport	500	2980	11.9	145	1	7

表 4.1: 実ネットワークの基本特性

4.2 提案法と従来法を比較するための各指標

2.2 節の式 (2.3∼5) を用いて、修復したネットワークの特性を調べる。図 4.1∼4.5 で、横軸はネットワークから次数順攻撃で削除したノード数の攻撃率 q で、縦軸はそのネットワークを修復した連結性 S_q(左)、頑健性 R(中)、経路効率 E(右) を測定したものである。

また、使える修復リンクの割合 α の値を変えながら、従来法と修復結果を比較した。線の色はその割合 α(5%:オレンジ色、10%:青色、20%:緑色、50%:黄色、100%:紫色) を示して、提案法はダイア印、従来法は丸印で表す。黒色点線は攻撃前の元々のネットワークに対する数値である。

図 4.1 は AirTraffic ネットワークに対する結果である。図 4.1(左) において、提案法は割合 α が 0.5 以上であると、ほとんどの連結性を維持できるが、高い q 値では少し減少する (図 4.1(左) の黄、紫線)。しかしながら、従来法は、攻撃率 q が 0.5 以上で、連結性が急激に減少した。割合 α が 0.2 以下でも、提案法は従来法より、高い連結性を示す (図 4.1(左) の赤、青、緑線)。

また、図 4.1(中)(右) において、割合 α が 0.5 以上で、提案法は高い頑健性と経路効率を示した (図 4.1(中)(右) の黄、紫線)。0.2 以下の割合でも、高い q 値では頑健性と経路効率が増加した (図 4.1(中)(右) の青、緑線)。これは、高い q 値での修復したネットワークが正則グラフになるからである。従来法は割合が 0.2 以下で、頑健性と経路効率が 0 に近く、0.5 以上でも減少しつつあり、攻撃以前のネットワークに対する数値 (黒点線) よりも低い値となっている。

図 4.2、4.3、4.5 の (左) から、ASOregon,OpenFlight,USAirport ネットワークに対する連結性も AirTraffic の結果と同じく、割合 α が 0.5 以上で提案法はほとんどの連結性を維持できるが、従来法は減少する (図 4.2,4.3,4.5(左) の黄、紫線)。また、ネットワークによっては (OpenFlight,USAirport) 割合 α が 0.2 でもほとんどの連結性を維持する (図 4.3,4.5(左) の緑線)。

また、図 4.2、4.3、4.5 の (中) から、ASOregon,OpenFlight,USAirport ネットワークに対するで頑健性を比較すると、提案法は割合 α が 0.5 以上で従来法より非常に高い数値を示した (図 4.2,4.3,4.5(中) の黄、紫線)。また、OpenFlight,USAirport ネットワークでは α が 0.2 でも、黒色点線の攻撃以前より高い数値を示す (図 4.3,4.5(中)

の緑線)。これは両ネットワークが他より平均次数が高くて、ネットワークの大きさに対して修復に使えるリンク数が多いからと考えられる。

図 4.2、4.3、4.5 の (右) における経路効率も頑健性と同じく、割合 α が 0.5 以上で提案法の結果が従来法より高い (図 4.2,4.3,4.5(右) の黄、紫線)。しかしながら、ASOregon ネットワークでは割合 α が 1.0 でも $q = 0.6$ 以下では黒色点線の攻撃以前の数値を超えてない。一方、平均次数がほぼ同じ AirTraffic ネットワークでは黒色点線の攻撃以前より高い数値を示したが、ASOregon ネットワークではそうではない理由は攻撃以前の ASOregon は総次数の弱 10%を占めるハブが存在して (表 4.1)、元々高い経路効率を持っていたと考えられる。

図 4.1、4.2、4.3、4.5 の (左) に示す、AirTraffic,ASOregon,OpenFlight,USAirport ネットワークに対する連結性の結果は、$q = 0.8 \sim 0.9$ で小幅減少する。しかしながら、PowerGrid ネットワークでは、急激に減少して、低い連結性を示す (図 4.4(左))。ゆえに、頑健性と経路効率も急激に減少する (図 4.4(中)(右))。これは、PowerGrid ネットワークが他のネットワークより高い直経を持つゆえに、3 ホップの通信では遠くにあるノードと連結できず、連結性を維持できないことに起因すると考えられる。

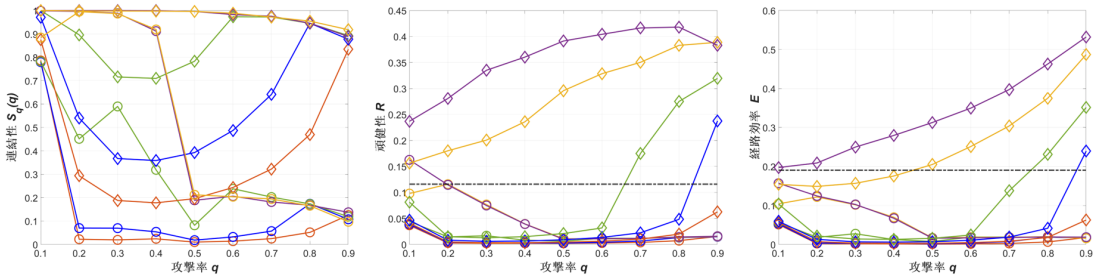

図 4.1: AirTraffic での修復結果

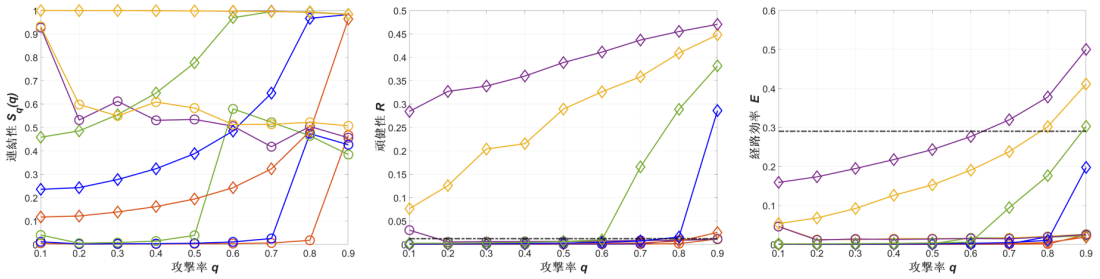

図 4.2: ASOregon での修復結果

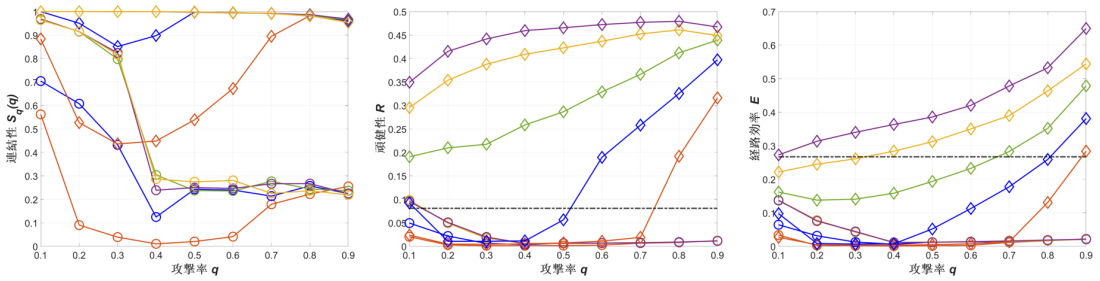

図 4.3: OpenFlight での修復結果

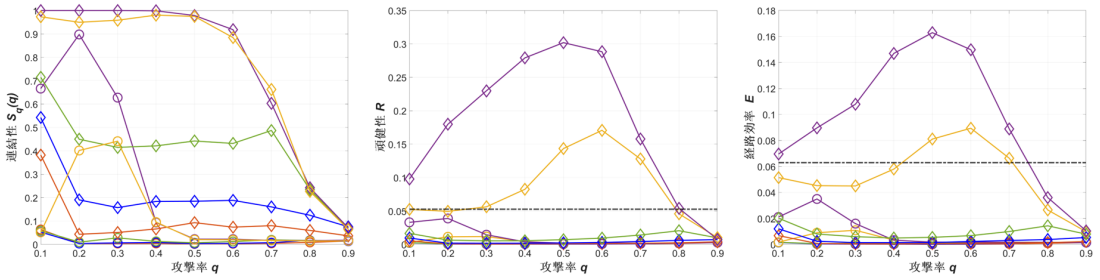

図 4.4: PowerGrid での修復結果

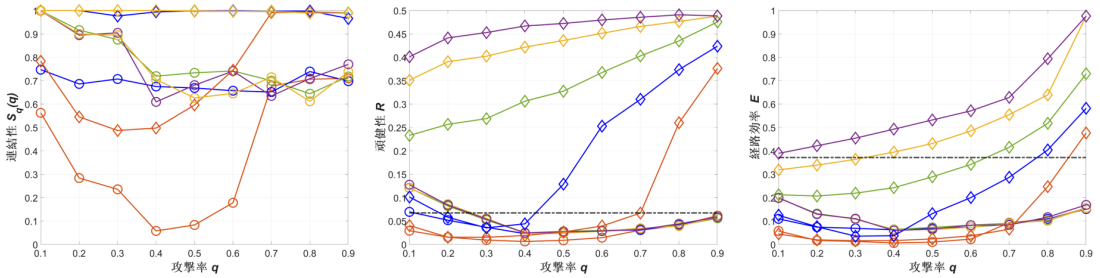

図 4.5: USAirport での修復結果

4.3 修復に必要な追加ポート数

本節では、修復したネットワークの特性とは異なる、修復法に必要な追加ポート数を比較する。リンクをノードに追加するとき、コンセントのように、接続する部分が必要である。この接続部はポートと呼ばれる。攻撃以前のネットワークでのノード次数はノードが使えるポート数を表し、攻撃でリンクを失ってもそれのポートは再利用できると考えるのが自然である。例えば、攻撃以前に次数 $k_i = 1$ だったノード i が修復後に $k_i = 5$ になると、修復に必要な追加ポート数は 4 である。以下に示す表 4.2~4.11 では、各ネットワークで提案法と従来法で修復した場合、必要な追加ポート数の最大値と平均値を示す (括弧中が平均値)。表の横はネットワーク攻撃率 q で、縦は式 (3.1) における割合 α を示す。

表 4.2 は、AirTraffic ネットワークを提案法で修復する場合、必要な追加ポート

数の最大値と平均値である。攻撃率 q と割合 α が高いほど、最大値と平均値が急激に増加する。高い q での修復ネットワークは正則グラフになるので、追加ポート数の最大値と平均値が高い。また、AirTraffic ネットワークでの最大次数 k_{max} が 34 であることを考えると、追加ポート数の最大値の上限は $1.15k_{max}$ を超えない。これは、次数が低いノード間にショートカットを繰り返して追加することで、修復ネットワークが正則グラフになるように、修復リンクが均等に分配されることに起因すると考えられる。

AirTraffic ネットワークに対する従来法の結果 (表 4.3) で、追加ポート数の最大値は低い q 値、高い α 値で大きい数値を示した。これは図 4.1 で、従来法が割合 α が 0.5 以上で、ほぼ全ての連結性を維持できたことと関係がある。即ち、従来法の修復範囲は前述のように 2 ホップであるが、低い攻撃率 q では 2 ホップの修復範囲でも多くの修復リンクを追加できることである。また、全ての q と α に対して、ほぼ同じ追加ポート数の平均値を示す。

表 4.4 は、ASOregon ネットワークを提案法で修復した時、必要な追加ポート数の最大値と平均値である。この表でも AirTraffic に対する結果と同じく、q と α が高いほど、最大値と平均値が増加する。しかしながら、ASOregon ネットワークの最大次数 $k_{max} = 1458$ と比較して最大値の上限は $0.03k_{max}$ 以下である。ASOregon は AirTraffic に比べて、弱 5 倍の総ノード数を持つが、平均次数は 3.9 で同じである。また、提案法の修復で割合 $\alpha \geq 0.5$ ではノード削除から生き残ったノードがほぼ全部一つの連結成分で繋がている (図 4.2(左) の黄線、紫線)。ゆえに、ASOregon と AirTraffic での追加ポート数の最大値と平均値が同価である理由はノード 1 個当たりに追加される修復リンク数が同じであるからと考えられる。ここで、ASOregon が次数が 1458 であるハブを持つことで、最大次数と比べて追加ポート数の最大値上限は非常に低い値 ($0.03k_{max}$) を持つ。

ASOregon ネットワークに対する従来法の結果 (表 4.5) でも、低い q 値と高い割合 α 値で大きい追加ポート数の最大値を示すが、その値が $1.15k_{max}$ を超えない。また、全ての q、α の範囲で追加ポート数の平均値は一定な数値を示す。

表 4.6 は、提案法を用いる OpenFlight ネットワークの修復に必要な追加ポート数の最大値と平均値である。攻撃率 q と割合 α が増加するとともに、追加ポート数の最大値も増加する。OpenFlight での最大次数 k_{max} は 242 で、最大値の上限は $0.45k_{max}$ である。しかしながら、OpenFlight での最大値の上限は AirTraffic、ASOregon での上限より高い数値を示す。これは OpenFlight の平均次数 ($\langle k \rangle = 10.8$) が他ネットワークの平均次数より高いことに起因すると考えられる。

OpenFlight に対する従来法に必要な追加ポート数の最大値と平均値 (表 4.7) は、AirTraffic、ASOregon での結果 (表 4.3、4.5) と同じく、全ての q、α の範囲で一定な追加ポート数の平均値を示す。また、提案法の結果 (表 4.6) で、最大値の上限は他ネットワークより高い数値を持つが、従来法の結果 (表 4.7) で、最大値の上限は他ネットワークと近接する数値を持つ。

PowerGrid ネットワークは他ネットワークより、高い直径と低い平均次数、最大

次数を持つ。しかしながら、PowerGrid ネットワークを提案法で修復する場合、必要な追加ポート数の最大値と平均値結果表 (4.8) は、他ネットワークの結果と同じく、q と α が高いほど、最大値と平均値も増加する。また、PowerGrid の最大次数 (k_{max}) はで、最大値の上限は $1.21k_{max}$ になる。また、他のネットワーク結果との差は、例えば $q = 0.9$、$\alpha = 1.0$ の時、追加ポート数の最大値は平均値より、AirTraffic で 1.09 倍、ASoregon で 1.02 倍、OpenFlight で 1.02 倍大きい。PowerGrid だけが 3.07 倍である。図 4.4(左) で示すように、$q = 0.9$、$\alpha = 1.0$ の場合、提案法で修復したネットワークは弱 0.1 の低い連結性を持つ。この時の修復ネットワークは、多数の拡張隣接と、その数より多い孤立ノードで構成されているから、修復リンクを持たない孤立ノードが追加ポート数の低い平均値に寄与することが原因だと考えられる。

表 4.9 は、従来法を用いる PowerGrid ネットワークの修復に必要な追加ポート数の最大値と平均値である。最大値は低い q 値、高い α 値で大きい数値を示し、全ての q、α の範囲で一定な平均値を示す。

最後に USAirport ネットワークでの結果として、表 4.10 は提案法で修復する場合、必要な追加ポート数の最大値と平均値である。最大値は q と α が高いほど増加し、最大値の上限は $0.41k_{max}(k_{max} = 145)$ である。$q = 0.9$、$\alpha = 1.0$ での最大値と平均値は $q = 0.8$、$\alpha = 1.0$ での数値より小さい。それは提案法で修復したネットワークは $q = 0.9$、$\alpha = 1.0$ の場合、最大連結成分大きさの平均値は 49 である。この最大連結成分は完全グラフであるが、$q = 0.8$、$\alpha = 1.0$ の修復ネットワークはその構造を持ていない。即ち、ノード攻撃から生き残ったノード数に対して、使える修復リンク数が多いことで、最大値と平均値が $q = 0.9$ で減少することはこれに基づくと考えられる。

表 4.11 は、USAirport ネットワークを従来法で修復する時に必要な追加ポート数の最大値と平均値である。他ネットワークでの従来法結果 (表 4.3、4.5、4.7、4.9) と同じく、低い q 値、高い α 値で大きい最大値を表し、平均値は全ての q、α の範囲で一定な値を示す。

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.1 (1.0)
0.1	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.2 (1.0)	3.0 (1.9)
0.2	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	2.0 (1.1)	3.0 (2.0)	7.0 (5.4)
0.5	1.7 (1.0)	3.0 (1.3)	3.0 (1.5)	3.0 (1.6)	3.0 (1.9)	4.0 (2.8)	6.0 (4.3)	9.0 (7.4)	19.0 (16.7)
1.0	4.0 (2.4)	4.0 (2.6)	5.0 (3.5)	6.0 (4.3)	7.0 (5.5)	9.0 (7.5)	13.0 (10.8)	19.0 (17.2)	39.0 (35.8)

表 4.2: AirTraffic で提案法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.4 (1.1)	1.2 (1.1)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.4 (1.1)	1.4 (1.1)	1.5 (1.1)
0.1	2.3 (1.2)	2.1 (1.1)	1.9 (1.1)	1.7 (1.1)	1.3 (1.1)	2.1 (1.1)	2.6 (1.2)	2.5 (1.2)	1.5 (1.1)
0.2	2.3 (1.2)	2.8 (1.2)	3.4 (1.2)	3.0 (1.2)	2.8 (1.2)	3.6 (1.3)	3.2 (1.3)	2.8 (1.3)	1.7 (1.2)
0.5	4.4 (1.3)	4.8 (1.4)	3.7 (1.3)	3.6 (1.3)	3.5 (1.3)	3.2 (1.3)	2.7 (1.2)	2.5 (1.2)	1.8 (1.1)
1.0	5.5 (1.6)	4.0 (1.4)	4.7 (1.3)	3.9 (1.3)	3.6 (1.3)	3.3 (1.2)	2.9 (1.2)	2.5 (1.2)	1.6 (1.1)

表 4.3: AirTraffic で従来法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)
0.1	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	3.0 (2.3)
0.2	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	2.0 (1.4)	3.0 (2.3)	7.0 (6.0)
0.5	2.0 (1.1)	2.0 (1.3)	2.0 (1.5)	3.0 (1.9)	3.0 (2.3)	4.0 (3.1)	6.0 (4.7)	9.0 (7.9)	19.0 (17.7)
1.0	4.0 (2.7)	4.0 (3.2)	5.0 (3.8)	6.0 (4.7)	7.0 (6.0)	9.0 (7.9)	12.0 (11.1)	19.0 (17.6)	38.0 (37.1)

表 4.4: ASOregon で提案法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.4 (1.0)	1.5 (1.0)	1.8 (1.0)	1.9 (1.1)	2.1 (1.1)	2.5 (1.1)	2.5 (1.1)	2.9 (1.2)	4.2 (1.4)
0.1	2.6 (1.1)	2.1 (1.1)	2.6 (1.1)	2.8 (1.1)	2.8 (1.1)	3.6 (1.2)	3.5 (1.3)	4.6 (1.4)	4.2 (1.4)
0.2	3.6 (1.2)	3.8 (1.2)	4.0 (1.2)	4.0 (1.2)	4.3 (1.3)	5.3 (1.4)	5.0 (1.4)	4.9 (1.4)	4.2 (1.4)
0.5	5.8 (1.4)	6.0 (1.4)	5.7 (1.4)	5.5 (1.4)	5.7 (1.4)	5.3 (1.4)	5.0 (1.4)	5.1 (1.4)	5.0 (1.4)
1.0	5.8 (1.4)	5.8 (1.4)	5.4 (1.4)	5.3 (1.4)	5.1 (1.4)	5.6 (1.4)	4.9 (1.4)	4.6 (1.4)	4.2 (1.4)

表 4.5: ASOregon で従来法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	2.0 (1.5)	5.0 (3.5)
0.1	1.6 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	2.0 (1.1)	2.0 (1.5)	3.0 (2.1)	5.0 (3.6)	10.0 (8.3)
0.2	3.0 (1.6)	3.0 (1.7)	3.0 (1.8)	3.0 (2.1)	4.0 (2.7)	5.0 (3.5)	7.0 (5.1)	10.0 (8.3)	21.0 (18.7)
0.5	6.0 (4.3)	7.0 (5.0)	7.0 (5.5)	8.0 (6.6)	10.0 (8.2)	13.0 (10.8)	17.0 (15.0)	26.0 (23.7)	53.0 (51.0)
1.0	11.0 (8.5)	13.0 (10.9)	15.0 (12.7)	17.0 (15.0)	21.0 (18.5)	26.0 (23.7)	35.0 (32.6)	53.0 (50.4)	107.0 (104.4)

表 4.6: OpenFlight で提案法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	2.7 (1.1)	2.4 (1.1)	2.3 (1.0)	2.4 (1.1)	3.1 (1.0)	3.2 (1.0)	3.5 (1.1)	3.5 (1.3)	2.8 (1.0)
0.1	3.3 (1.1)	3.5 (1.2)	3.8 (1.1)	4.3 (1.1)	3.9 (1.1)	4.0 (1.2)	3.6 (1.2)	3.4 (1.3)	2.8 (1.1)
0.2	6.3 (1.3)	5.1 (1.2)	5.0 (1.1)	4.4 (1.1)	4.3 (1.2)	3.7 (1.3)	3.9 (1.3)	3.2 (1.2)	2.6 (1.1)
0.5	5.7 (1.5)	4.6 (1.4)	4.5 (1.3)	4.4 (1.3)	3.9 (1.3)	4.1 (1.3)	3.4 (1.2)	3.1 (1.2)	3.0 (1.1)
1.0	5.8 (1.5)	4.8 (1.4)	4.4 (1.3)	5.0 (1.2)	4.2 (1.3)	4.1 (1.3)	3.6 (1.3)	3.5 (1.2)	2.8 (1.2)

表 4.7: OpenFlight で従来法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.2 (1.0)	1.5 (1.0)	2.0 (1.0)	2.0 (1.0)	1.8 (1.0)	1.9 (1.0)	1.9 (1.0)	1.6 (1.0)	1.4 (1.0)
0.1	1.9 (1.0)	2.0 (1.0)	2.1 (1.0)	2.0 (1.0)	2.0 (1.0)	2.1 (1.0)	2.2 (1.0)	2.0 (1.0)	2.0 (1.2)
0.2	2.0 (1.0)	2.2 (1.0)	2.5 (1.0)	2.2 (1.0)	2.3 (1.0)	2.3 (1.0)	2.5 (1.1)	2.0 (1.3)	5.0 (2.2)
0.5	2.3 (1.0)	2.7 (1.0)	2.4 (1.1)	2.6 (1.2)	2.5 (1.4)	3.0 (1.9)	4.0 (2.6)	6.0 (3.8)	12.7 (4.9)
1.0	2.8 (1.2)	3.0 (1.8)	3.0 (2.0)	4.0 (2.8)	5.0 (3.5)	6.0 (4.5)	8.0 (6.1)	13.0 (8.5)	23.0 (7.5)

表 4.8: PowerGrid で提案法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.3 (1.0)	1.4 (1.0)	1.2 (1.0)	1.4 (1.0)	1.2 (1.0)	1.1 (1.0)	1.6 (1.0)	1.9 (1.1)	1.9 (1.1)
0.1	2.1 (1.1)	1.6 (1.0)	2.0 (1.0)	1.8 (1.0)	1.8 (1.0)	1.9 (1.0)	2.5 (1.1)	2.7 (1.1)	1.4 (1.0)
0.2	3.0 (1.1)	2.4 (1.1)	2.3 (1.1)	2.5 (1.1)	2.8 (1.1)	2.9 (1.1)	3.3 (1.1)	2.2 (1.1)	1.4 (1.0)
0.5	4.2 (1.2)	4.0 (1.2)	4.3 (1.2)	4.2 (1.2)	3.9 (1.2)	3.4 (1.2)	3.0 (1.1)	2.7 (1.1)	1.6 (1.1)
1.0	5.3 (1.3)	5.2 (1.3)	4.7 (1.2)	4.3 (1.2)	3.8 (1.2)	3.6 (1.2)	3.1 (1.1)	2.6 (1.1)	1.5 (1.1)

表 4.9: PowerGrid で従来法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	2.0 (1.5)	5.0 (3.7)
0.1	1.4 (1.0)	1.1 (1.0)	1.0 (1.0)	1.0 (1.0)	2.0 (1.3)	2.0 (1.5)	3.0 (2.2)	5.0 (3.7)	11.0 (8.9)
0.2	3.0 (1.6)	3.0 (1.8)	3.0 (1.9)	3.0 (2.2)	4.0 (2.8)	5.0 (3.8)	7.0 (5.3)	11.0 (8.7)	23.0 (19.7)
0.5	7.0 (4.9)	7.0 (5.1)	8.0 (5.9)	9.0 (7.0)	11.0 (8.8)	14.0 (11.6)	19.0 (16.0)	29.0 (25.4)	46.5 (42.7)
1.0	13.0 (10.2)	14.0 (11.4)	17.0 (13.4)	19.0 (15.9)	23.0 (19.7)	29.0 (25.6)	39.0 (35.4)	59.0 (55.0)	46.4 (43.0)

表 4.10: USAirport で提案法に必要な追加ポート数の最大値と平均値

$\alpha\backslash q$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.05	1.4 (1.0)	1.7 (1.1)	1.5 (1.1)	1.6 (1.1)	1.8 (1.2)	2.3 (1.1)	2.6 (1.3)	2.4 (1.2)	1.5 (1.2)
0.1	2.8 (1.3)	2.7 (1.2)	2.3 (1.2)	2.5 (1.2)	2.9 (1.2)	2.8 (1.3)	2.4 (1.2)	2.2 (1.3)	1.8 (1.2)
0.2	5.0 (1.4)	3.5 (1.3)	3.1 (1.3)	2.6 (1.3)	2.7 (1.2)	2.3 (1.2)	2.7 (1.3)	2.1 (1.2)	1.4 (1.2)
0.5	4.7 (1.4)	3.8 (1.3)	2.7 (1.2)	3.1 (1.3)	2.8 (1.2)	2.9 (1.3)	2.4 (1.2)	2.3 (1.3)	1.9 (1.2)
1.0	4.8 (1.4)	3.9 (1.3)	3.0 (1.3)	3.4 (1.3)	2.9 (1.3)	3.0 (1.3)	2.2 (1.2)	2.4 (1.2)	2.0 (1.3)

表 4.11: USAirport で従来法に必要な追加ポート数の最大値と平均値

第5章　提案修復法の自律分散アルゴリズムとしての記述

4.2 節と 4.3 節の結果は、逐次的なコンピューターシミュレーションのプログラムとして、ネットワーク全体の情報を前提して、リンクを失ったノードに修復リンクを与えている。しかしながら、このような中央集権の処理は、実際のシステム上の実装を考えると、情報を集める時間と中央から命令する時間遅れや、収集する経路が破壊される可能性などが生じ得る。ゆえに、リンクを失った各ノードが壊れたノードを検知することから自律的に開始して、お互いのノードやリンクの局所情報だけをやり取りしながら処理を進める分散アルゴリズムを考える。このアルゴリズムは、必要な変数を宣言するイニシエーションを除いて、5 つのフェーズで構成されている。まずは、それら全体的な概要と、このアルゴリズムのために前提にしている条件を説明する。その後、イニシエーションを含めて、各フェーズにおける目標や処理の概要を説明して、アルゴリズムの動作について述べる。

5.1　全体的な概要

ネットワーク上の各ノードは、外部からの攻撃や内部での故障で機能が停止する場合がある。隣接ノードが機能停止して応答がなく、それとの結合が途絶えてダメージを受けたノードは修復リンクの結合対象となる。ここで、最初に連絡できる範囲は元のネットワーク上で 3 ホップ離れている結合対象までだと仮定する。なぜなら、第三章で述べたように、２ホップでは遠くのノードと連結できず、高い連結性を維持できないからである。各結合対象はいろんな制御情報をメッセージできる範囲内でやり取りすることで、範囲外にあるノードの制御情報を収集する。ネットワーク連結性や頑健性を考慮して修復を行う。

5.2　前提条件

従来の研究 [12][13] を参照して、本節で前提する非同期システムが満たす条件を以下に述べる。記述する分散アルゴリズムは、このシステム上で行われる。

- 制御信号のやり取りが終わるまで、追加的なノードやリンクの破壊はないものとする。もし、途中状態に破壊があれば検出から改めて再開する。

- 各ノードでの処理速度と送受信によるタイムラグは無視できるものとする。
- 各フェーズでは、そのフェーズ以外の動作はしない。
- ほぼ同時に届いたメッセージはランダム順に処理する。
- 待機列に入ったメッセージは即座に処理し、処理順は到着した順にする (FIFO)。
- ネットワークの各ノードはグローバルな共有メモリーをシェアせず、局所的な情報を受けてメッセージを交換することのみで、記述されたプロセスに従い次の動作を行う。
- 初めに故障を検知したノードは修復対象になり、自分から 3 ホップ離れているノードまで制御信号のやり取りが直接できるものとする。
- 各ノードでの処理は必ず有限時間内に行われ、処理後の送信も即座に行う。
- ノード同士の局所時計の時刻は違っていてもかまわない。なぜなら、各ノードにおけるメッセージの到着順のみを考慮するからである。

5.3 イニシエーション

故障を検知したノード (修復リンクの結合対象) は自発的にイニシエーションを実行する。各結合対象は元々の 3 ホップ内にある全てのノードまで直接通信できると仮定している。そのノード集合を ECN_i(Emergency Contactable Nodes) とする (図 5.1 の 1 行)。また、あるノードの最近接ノード ID 集合を $neighbors_i$ とする。ネットワークにある全てのノードは一般的な通信や物資輸送などの処理を行う通常モードにおいて既に自分の ECN_i と $neighbors_i$ を予め知っているものとする。

結合対象 i は ECN_i へ Alert メッセージをブロードキャストする (図 5.1 の 3~4 行)。Alert メッセージをもらった結合対象は送信先 ID を DC_i(Direct Contacts) にセーブすることで、各ノードは自分の ECN_i の中で、他の修復リンクの結合対象であるノードの ID を知る (図 5.1 の 6 行)。修復リンクは結合対象間に追加されるので、仮定した手段を使って ECN_i の中で対象ではないノードとは通信しない。ゆえに、ECN_i 全体へメッセージを伝搬せず、DC_i を特定してフェーズを進行して、フラッディングのような無駄なメッセージ処理を防ぐ。図 5.1 は故障を検知したノード i イニシエーションである。宣言する 2 つの変数は後述するフェーズ 1 から 5 まで使用するので予め指定する。また、修復が完了するまで追加的な故障はないと仮定する。もし、追加的な故障があれば再度検知からやり直す。

```
Initiation
1  Declare the variable ECN_i which is the set of NodeIDs within 3hops
2  DC_i ← ∅                                          % the initial set
3  For all k ∈ ECN_i
4      Send Alert (i) to k

5  Process of Receive Alert (j) from j to i
6      DC_i ← DC_i ∪ {j}
```

図 5.1: 故障を検知したノード i のイニシエーション

5.4 フェーズ 1: 修復リンクの範囲を拡張する

5.4.1 フェーズ 1 の説明

ノード i のイニシエーションにより ECN_i の各結合対象は、3 ホップにある他の結合対象 (DC_i) の ID を知る。修復リンクの範囲を拡張するために、3 ホップ以上にあるノードの ID が必要で、その ID を知ると DC_i のノードを中継して通信できる。このため、それらの ID を知るために、各結合対象は自分が知っている ID を Gathering メッセージに入れて DC_i のノードに送る。

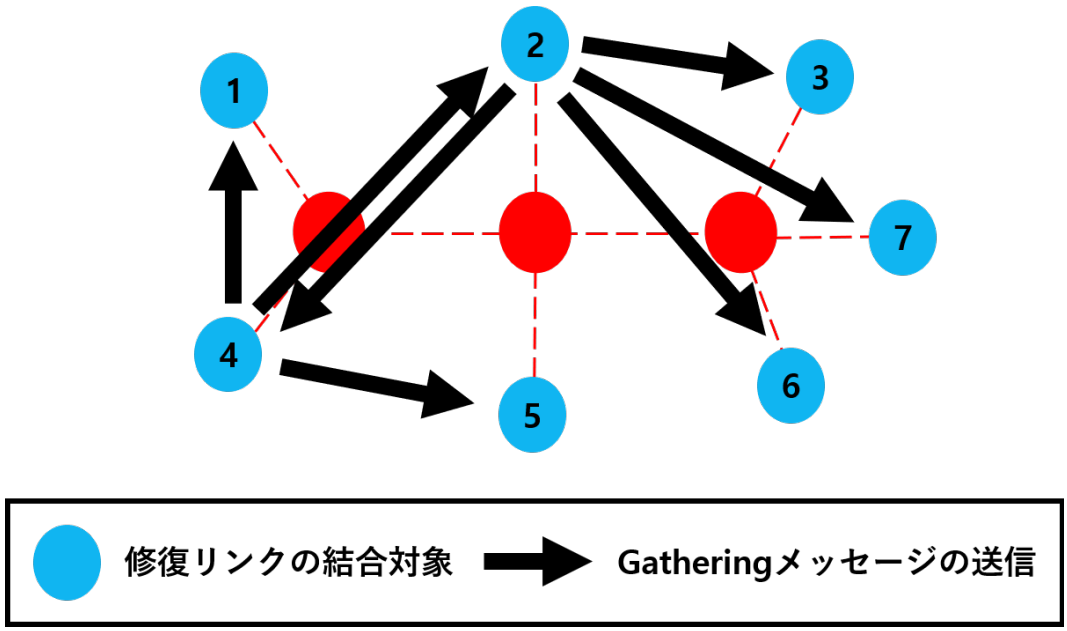

図 5.2: フェーズ 1 の概要図。 フェーズ 1 で、ノード 2 と 4 が送信する Gathering メッセージ。ノード 4 はノード 2 を通じてノード 3,6,7 とも通信できる。 故障ノード: 赤ノード

例えば、図 5.2 でノード 4 はノード 2 へ Gathering メッセージを送る。ノード 4 が最初に知っている ID は DC_4 のノード集合だけである。他の結合対象も同じ動

作をするので、ノード 2 から Gathering メッセージをもらったノード 4 は DC_2 にあったノード 3,6,7 を知る。こうしたメッセージを介して増加するノード ID 集合を $Expanded\_DC$ とする。この時、ノード 4 は表 5.1 のように臨時のルーティング表を作ることができる。なぜなら、ノード 4 が直接通信できるのはノード 2 であるが、そこからノード 3,6,7 の ID をもらうことにより、ノード 2 を介してノード 3,6,7 と通信できるから、ノード 2 をルーティング表の転送先とすれば良い。その後、ノード 4 は増加した ID 集合を Gathering メッセージに入れて DC_4 の全てのノードへ送る。この過程は自分の DC_i から来る Gathering メッセージに、新しく知れるノード ID がないまで繰り返し、その結果遠くのノードの ID を知る。

送信元	受信先	転送先
4	3	2
4	6	2
4	7	2

表 5.1: ノード 4 が持つ臨時のルーティング表。ノード 4(送信元) がノード 3,6,7(受信先) にメッセージを送るためには、ノード 2(転送先) を中継する

5.4.2 フェーズ１のアルゴリズムの動作

以下は、フェーズ１における変数の意味を説明したものである。

- ECN_i : ノード i から３ホップ内にあるノードの不変集合。集合にあるノードとは通常の送信先とは違い、他の手段を用いて通信する。ネットワークにある全てのノードは自分の ECN_i を通常から知っているものとする。

- DC_i : 故障を検知したノード i(修復リンクの結合対象) から３ホップ内にある、他の結合対象の ID の可変集合。結合対象 i は DC_i にあるノードとメッセージを交換することで、３ホップ外にあるノードの ID を知る。

- $Expanded\_DC_i$: 結合対象 i が知っている、他の結合対象 ID の可変集合。最初は DC_i だけを知るが、DC_i のノードと Gathering メッセージをやり取りすることで、この集合を拡張する。

このフェーズの目標は、修復リンクの結合対象が自分から３ホップ外にある他の結合対象の ID を収集することである。フェーズ１が始まった結合対象 i は DC_i に向かって、自分が知っている結合対象の ID 集合を表す $Expanded\_DC_i$ を Gathering メッセージに入れて送る (図 5.3 の 2,3 行)。

Gathering メッセージを受信する場合、メッセージにある $Expanded\_DC_i$ と自分の $Expanded\_DC_i$ を合わせて更新する (図 5.3 の 5 行)。更新した $Expanded\_DC_i$

に以前にはなかったノード ID があると、Gathering メッセージの送信元を中継先にして前述したルーティング表を作る (図 5.3 の 6~8 行)。

全ての DC_i から Gathering メッセージを受信した時 (図 5.3 の 9 行)、以前 DC_i に送った $Expanded\_DC_i$ より新しいノード ID があると、また DC_i に Gathering メッセージを送る (図 5.3 の 10~12 行)。しかしながら、全ての DC_i からメッセージをもらっても、$Expanded\_DC_i$ に新しい情報 (結合対象の ID) がないなら (図 5.3 の 13 行)、このノードでのフェーズ 1 は終了して、フェーズ 2 を始める (図 5.3 の 14 行)。この時、$Expanded\_DC_i$ にあるノードはお互い他の結合対象を中継して通信することができ、$Expanded\_DC_i$ にあるノードは全て同じ $Expanded\_DC_i$ を持つことになる。この $Expanded\_DC_i$ を拡張隣接と呼ぶ。

```
(Phase1)

1  Start()
2  For all k ∈ DC_i
3      Send Gathering (Expanded _DC_i ) to k

4  Process of Receive Gathering (Expanded _DC_j ) from j to i
5      Expanded _DC_i ← Expanded _DC_i ∪ Expanded _DC_j          % updating the node ID info
6      if Expanded _DC_i has a new info then
7          Make the temporary routing table
8      end if
9      if i has been received all of Gathering from DC_i then
10         if Expanded _DC_i has a new info then
11             For all k ∈ DC_i
12                 Send Gathering (Expanded _DC_i) to k
13         else
14             Terminate the phase 1 and start the phase 2
15         end if
16     end if
```

図 5.3: ノード i でのフェーズ 1 のアルゴリズム

5.5 フェーズ 2: 連結成分の大きさを知るため配信木を作る

5.5.1 フェーズ 2 の説明

フェーズ 1 の後、各結合対象は根ノードになって配信木を作り、自分が属している連結成分の大きさを調べる。以下のようにして、連結成分にあるノードが分散処理として最近接ノードとメッセージをやり取りすることだけで、根ノードは連

結成分の大きさを知り、故障事実を同じ連結成分にある他のノードに知らせることができる。故障を検知したノードが他のノードに故障事実を伝搬する従来のアルゴリズム [14] では、根ノードが幅優先探索による木構造を作る。しかしながら、そのアルゴリズムで各ノードはネットワーク全体の構造情報 (連結関係) を知っている。このフェーズではノードが構造情報を持たなくても、親ノードを指定することだけで木構造を作る長所がある。

プロセスは例えば、図 5.4(左) で、根ノードであるノード 1 は最近接 (ノード 2,3) へ Mode_change メッセージを送る。Mode_change メッセージを受信したノード 2,3 は自分のモードを通常から修復に変えて、ノード 1 を親ノードにする。また、Mode_change メッセージを受信すると、そのメッセージにある根ノード ID をセーブするが、それの理由は後述する。

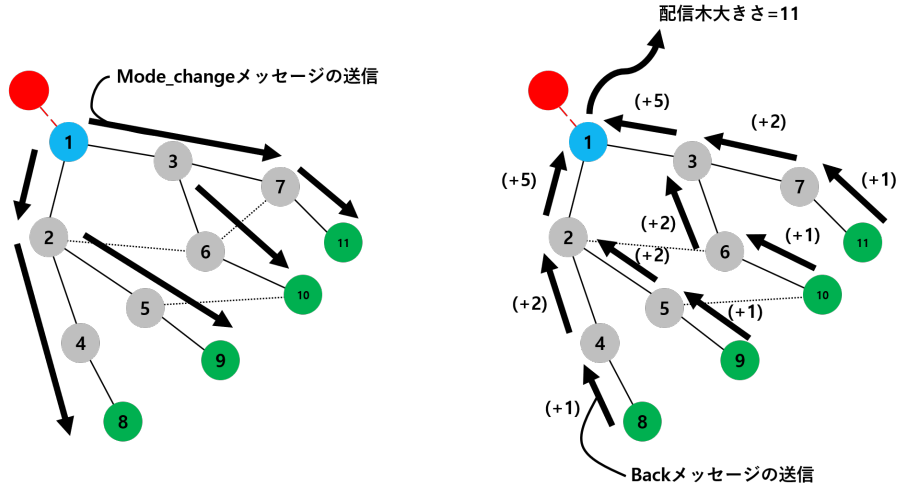

図 5.4: フェーズ 2 で配信木を作る過程の概要。まずは葉ノードまで Mode_change メッセージを送り、後は葉から Back メッセージを送る; 根ノード: 青ノード、葉ノード: 緑ノード、配信木に含まれないリンク: 黒点線、配信木: 黒実線

その後、ノード 2,3 は親ノードを除いた自分の最近接へ Mode_change メッセージを送る。この場合、ノード 6 はノード 2,3 から Mode_change メッセージか来る。その際、先に来る Mode_change メッセージを処理して親ノードにする。その後に来る Mode_change メッセージに対しては、No_children メッセージで返答する。図 5.4(左) ではノード 3 から先に Mode_change メッセージが来たとして、ノード 6 はノード 3 を親ノードにする。ゆえに、黒点線は連結成分では実際繋がっているが、作られる配信木では繋がっていないことを意味する。このプロセスは根ノードが属する連結成分全体に拡がるまで繰り返される。送信したノードを親とし、それ以外の隣接ノードを子にすると、この過程の最後には、子ノードがない葉ノードが見つかり、連結成分は一つの木構造 (配信木) になる。

葉ノードになったノード 8,9,10,11 は最近接が親ノードしかないから、図 5.4(右)

のように、自分の親ノードへ大きさの情報を入れて Back メッセージを送る。各ノードが持つ大きさ情報の初期値は 1 とする。親ノードを除いた全ての最近接から No_children メッセージや Back メッセージをもらったノードはもらった大きさ情報と自分のを足して、また自分の親ノードへ送る。例えば、図 5.4(右) のノード 4 はノード 8 から 1 の大きさ情報をもらって、自分の大きさを足して、親ノードであるノード 2 に 2 の大きさ情報がある Back メッセージを送る。また、ノード 2 は親ノードではない隣接の子ノードになるノード 4,5 からは Back メッセージをもらい、以前 No_children メッセージをもらったノード 6 は子ノードではない。これでノード 2 は全ての最近接との親子関係を明らかにしたので、自分の親ノードに大きさ情報を送ることができる。このように、各ノードは自分の親ノードのみ変数で指定して、子ノードを識別する別の集合を持たずに、もらうメッセージの種類だけで判別する。この過程を繰り返すと根ノードは自分が属する連結成分の大きさを知る。

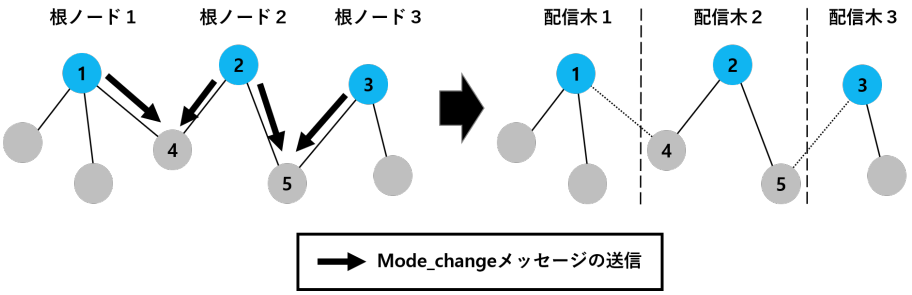

図 5.5: 複数の配信木が生じる過程。(左) 連結成分に多数の根ノードがあると、ノード 4,5 のように、異なる配信木から Mode_change メッセージをもらえる。(右) 最初に来たメッセージの送信元を親にして、後のメッセージを断ることで、各配信木構造は固定される

注意すべきことは、一度 Mode_change メッセージをもらって親ノードを決めると、二度 Mode_change メッセージをもらっても親ノードを変えないことである。図 5.4 では連結成分に根ノードが一つだったが、図 5.5 ように連結成分に複数の根ノードがあると、配信木は根ノードごとに生じる。この時、ノード 4,5 は各自 2 つの根ノード 1 と 2 や 2 と 3 から Mode_change メッセージをもらう (図 5.5(左))。Mode_change メッセージをもらうたびに、親ノードを変えると大きさ情報を重複に送って無駄や混乱が生じるので、図 5.5(右) のように、親ノードを固定して配信木を分離する。若しノード 4 がノード 2 を親ノードにして、その後ノード 1 から Mode_change メッセージをもらうと、ノード 4 はノード 1 へ No_children メッセージを送る。No_children メッセージを送ることで、自分はノード 1 の子ノードではないことを知らせると同時に配信木 2 の存在も知らせられる。このように、異なる配信木から Mode_change メッセージをもらうノードは、自分が属しない配信木に自分が属する配信木の存在を知らせる。これは後述するフェーズ 3 のためで、フェー

ズ 3 では複数の配信木を合わせて連結成分全体の大きさを導き出す。

5.5.2 フェーズ 2 のアルゴリズムの動作

以下は、フェーズ 2 における変数の意味を説明したものである。

- $Parent_i$: ノード i の親ノードの ID。最初にもらって処理した Mode_change メッセージの送信元が親ノード (根ノードは自分自身が親ノード)
- $Root_i$: ノード i が知っている配信木の根ノード ID の可変集合。異なる配信木から Mode_change メッセージをもらうノードは複数の根ノード ID を知る。
- $Num\_of\_candidates_i$: ノード i が親子関係を明らかにする必要がある最近接ノードの数。最初は最近接ノード数から親ノードを引いた値だが、No_children メッセージや Back メッセージを受けて親子関係を明らかにするたびに減少する。
- $Size_i$: ノード i が持っている配信木大きさの変数。根ノードを含めて全てのノードが持つ初期値は 1 であり、子ノードから来た $Size_i$ を全部足すことで、配信木上で自分から葉ノードまでの連結成分の大きさを知る。

フェーズ 1 が終わった結合対象 i は上記の変数を指定して (図 5.6 の 2~5 行)、最近接 ($neighbors_i$) へ Mode_change メッセージを送る (図 5.6 の 6,7 行)。

Mode_change メッセージを初めて受信したノードは (図 5.6 の 10 行)、通常のプロセスを止めて、修復処理の準備をする。そして、受信元 j を親ノードにし、根ノード ID を受け取る (図 5.6 の 11,12 行)。また、$Size_i$ と $Num\_of\_candidates_i$ 変数の指定する (図 5.6 の 13,14 行)。$Size_i$ は大きさの情報で、初期値は 1 であり、$Num\_of\_candidates_i$ はノード i が親子関係が設立する候補のノード数で、初期値は親ノードを除いた最近接の個数である。その後、親ノードを除いた最近接へ Mode_change メッセージを送る (図 5.6 の 18,19 行)。

親ノードを指定してから Mode_change メッセージを受信すると (図 5.6 の 17 行)、そのメッセージにある根ノード ID だけを受け取って、No_children メッセージで送信元に返答する (図 5.6 の 22,23 行)。これは、概要で述べたように、異なる配信木から来た Mode_change メッセージには、自分が持っている根ノード ID とは違う根ノード ID がある。ゆえに、これを受け取った後、Back メッセージに入れて自分の根ノードに送ることで、根ノードは自分が属する連結成分にある他の根ノードの存在を知る。この動作はフェーズ 3 のためであり、フェーズ 3 については後述する。

あるノードでは Mode_change メッセージをもらって指定する $Num\_of\_candidates_i$ が 0 になることがある (図 5.6 の 15 行)。このノードは配信木上の葉ノードである。葉ノードは親ノード以外の最近接が存在しない。ゆえに、Back メッセージを自分

親ノードへ送る (図 5.6 の 16 行)。Back メッセージには大きさの情報 ($Size_i$) と根ノード ID 集合 ($Root_i$) があり、これの根ノード ID 集合を自分の根ノードへ送る。

No_children メッセージを受信すると、$Num\_of\_Candidates_i$ から 1 を引く (図 5.6 の 26 行)。送信元は子ノードにならないから、大きさの情報は得ない。

Back メッセージを受信する時も、$Num\_of\_Candidates_i$ から 1 を引く (図 5.6 の 30 行)。また、メッセージにある $Root_j$ を自分の $Root_i$ に加える (図 5.6 の 31 行)。これで、他のノードが収集した異なる配信木の根ノード ID を自分の根ノードに伝える。メッセージにある $Size_i$ も、自分の $Size_i$ と足す (図 5.6 の 32 行)。

親ノードを除いた全ての最近接から No_children メッセージや Back メッセージをもらうと、$Num\_of\_candidates_i$ は 0 になる (図 5.6 の 33 行)。こうすると、親ノードに自分が持っている $Root_i$ と $Size_i$ 入れて Back メッセージを送る (図 5.6 の 35 行)。葉ノードから始めて Back メッセージを親ノードに送信することを繰り返すことで、根ノードは自分の配信木にある全ての情報 ($Root_i$、$Size_i$) をもらえる。根ノードは親ノードが自分自身であるから、Back メッセージを送れるノードがない。ゆえに、$Same\_comp_i$ と $Size\_list_i$ と呼ばれる集合変数を生成して、フェーズ 2 を終わらせる (図 5.6 の 37~39 行)。$Same\_comp_i$ の初期値は $Root_i$、$Size\_list_i$ の初期値は $Size_i$ で、これらの変数についてはフェーズ 3 にて後述する。

(Phase2)

```
1  % Only the root node finished the phase 1 %
2  Parent_i ← i
3  Root_i ← {i}                                          % Initial set
4  Size_i ← 1
5  Num_of_Candidates_i ← |neighbors_i| − 1;
6  For all k ∈ neighbors_i
7      Send Mode_change(i, Root_i) to k
8  % End %

9  Process of Receive Mode_change(j, Root_j) from j to i
10     if Parent_i =⊥ then                               % ⊥ means nul at the normal mode.
11         Parent_i ← j
12         Root_i ← Root_j                               % Initial set
13         Size_i ← 1
14         Num_of_Candidates_i ← |neighbors_i| − 1;
15         if Num_of_Candidates_i = 0 then               % if a node becomes leaf
16             Send Back(i, Root_i, Size_i) to j         % send Back message
17         else
18             for all k ∈ neighbors_i \ j
19                 Send Mode_change(i, Root_i) to k
20         end if
21     else
22         Root_i ← Root_i ∪ Root_j                      % updating the root ID info
23         Send No_children(i) to j
24     end if

25 Process of Receive No_children(j) from j to i
26     Num_of_Candidates_i ← Num_of_Candidates_i − 1
27     if Num_of_Candidates_i = 0 then
28         Send Back(i, Root_i, Size_i) to Parent_i

29 Process of Receive Back(j, Root_j, Size_j) from j to i
30     Num_of_Candidates_i ← Num_of_Candidates_i − 1
31     Root_i ← Root_i ∪ Root_j
32     Size_i ← Size_i + Size_j
33     if Num_of_Candidates_i = 0 then
34         if parent_i ≠ i then
35             Send Back(i, Root_i, Size_i) to Parent_i
36         else                                          % Only root node permits this condition
37             Same_comp_i ← Same_comp_i ∪ Root_i
38             Size_list_i ← Size_list_i ∪ {Size_i}
39             Terminate phase 2 and caculate the component size(Comp_size_info_i)
40         end if
41     end if
```

図 5.6: ノード i でのフェーズ 2 のアルゴリズム

5.6　フェーズ 3: 連結成分全体の大きさを求める

5.6.1　フェーズ 3 の説明

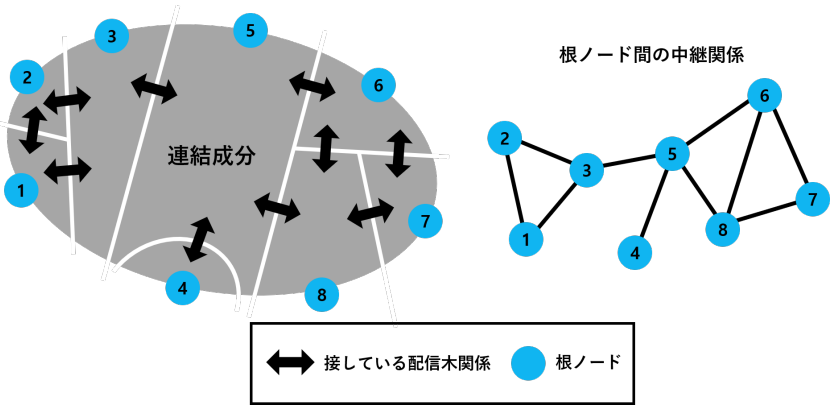

図 5.7: フェーズ 3 の概要図。(左) 連結成分にある多数の配信木。8 つの根ノードがあり、8 つの配信木が存在する。(右) 各配信木が隣接する関係

フェーズ 2 が終わると、図 5.7(左) のように、連結成分に複数の配信木が生じることがある。連結成分全体の大きさを知るためには、これらの全ての配信木の大きさが必要である。このために、フェーズ 2 で根ノードに他の根ノード ID を伝えている。各根ノードの $Root_i$ 変数には多数の根ノード ID があり、フェーズ 1 で作った臨時ルーティング表を用いて、他の根ノードと通信できる。しかしながら、図 5.7(右) のように、ある根ノードは隣接する配信木の根ノードだけを知る。つまりフェーズ 2 を進めても、図 5.7 での根ノード 2 は遠くの根ノード 7 を知らない。ゆえに、最初に知っている他の根ノード ($Root_i$) とメッセージを交換して、根ノード ID 集合を増加させる。その時、連結成分上で、図 5.7(左) のように配信木が生成されていると、根ノード間のメッセージ中継関係は図 5.7(右) のようになる。

フェーズ 3 を始める根ノードは、知っている他の根ノードに Disseminate メッセージを送る。Disseminate メッセージには根ノード ID 集合 ($Same\_comp_i$) と配信木の大きさ集合 ($Size\_list_i$) がある。例えば図 5.7 のノード 2 はノード 3 から Disseminate メッセージをもらってノード 5 の ID を知り、ノード 3 はノード 5 から Disseminate メッセージをもらってノード 6 の ID を知る。隣接する根ノードとの Disseminate メッセージ交換を繰り返すと、各根ノードは連結成分にある全ての根ノード ID を知ることができる。しかしながら、ノード 7 がまだフェーズ 2 を進めているとすると、全ての根ノード ID は知っているが、ノード 7 が Disseminate メッセージを送らないと、ノード 7 がある配信木の大きさは分からない。そこで、他の根ノードはフェーズ 3 を終わらせずに待機する。このアルゴリズムで、各ノードでの処理は必ず有限時間内に行い、処理後の送信も即座に行うものと仮定しているので、ノード 7 は必ずフェーズ 2 を終わらせる。ゆえに、ノード 7 がフェーズ 3 を始めて自分

の配信木大きさを伝搬することで、他の根ノードがそれを知り、フェーズ 3 を終了する。フェーズ 3 で全ての根ノードは自分が属する連結成分の大きさ ($Size\_list_i$ の和) とそれに属する根ノード ID($Same\_comp_i$) を知る。

5.6.2 フェーズ 3 のアルゴリズムの動作

以下は、フェーズ 3 における変数の意味を説明したものである。

- $Same\_comp_i$: ノード i が知っている、同じ連結成分に属する他の根ノード ID の可変集合。初期集合は $Root_i$ である。Disseminate メッセージで、他の根ノードが知る ID 集合を受け取ることで、この集合は増加する。

- $Size\_list_i$: ノード i が知っている配信木大きさの可変集合。初期集合は $Size_i$ である。Disseminate メッセージで、他の根ノードが知る ID 集合を受け取ることで、この集合は増加する。

フェーズ 3 を始めるノード i は $Root_i$ に Disseminate メッセージを送る (図 5.8 の 1~4 行)。Disseminate メッセージには今知っている根ノード ID 集合 ($Samp\_comp_i$) と配信木サイズ集合 ($Size\_list_i$) がある。

Disseminate メッセージを受信した時、メッセージにある情報を自分の変数と合わせて更新する (図 5.8 の 6,7 行)。更新した変数に、以前の変数にはなかった新しい情報があると、また $Root_i$ に Disseminate メッセージを送る (図 5.8 の 8~11 行)。この過程を繰り返すと、まだフェーズ 2 である根ノード ID は知るが、それが属する配信木の大きさは知らない。連結成分全体の大きさを知るためには、連結成分にある全ての配信木の大きさが必要である。各ノードでの処理は必ず有限時間内に行い、処理後の送信も即座に行うものと仮定しているので、根ノードのフェーズ 2 はいつか必ず終わる。ゆえに、各根ノードは自分の $Samp\_comp_i$ と $Size\_list_i$ の大きさが同じくなるまで待機する (図 5.8 の 12 行)。即ち、知っている根ノード数と配信木の大きさの数が異なるから、まだ誰かがフェーズ 2 を進めていることを分かる。連結成分にある全ての根ノードがフェーズ 2 を終わらせると、各根ノードは連結成分全体の大きさを計算できるようになり (図 5.8 の 13 行)、計算した後フェーズ 4 を始める。

```
(Phase3)
1  % Only the root node finished the phase 2 %
2  For all k ∈ Root_i \ i
3      Send Disseminate (Same_comp_i, Size_list_i) to k
4  % End %

5  Process of Receive Disseminate (Same_comp_j, Size_list_j) from j to i
6      Same_comp_i ← Same_comp_i ∪ Same_comp_j                    % updating the info
7      Size_list_i ← Size_list_i ∪ Size_list_j                    % updating the info
8      if Same_comp_i or Size_list_i has a new info then
9          For all k ∈ Root_i \ i
10             Send Disseminate (Same_comp_i, Size_list_i) to k
11     end if
12     if Same_comp_i has the same set size of Size_list_i then
13         Calculate the component size(Comp_size_info_i)
14         Terminate the phase 3 and start the phase 4
15     end if
```

図 5.8: ノード i でのフェーズ３のアルゴリズム

5.7 フェーズ 4: 拡張した修復リンクの範囲の情報を集める

5.7.1 フェーズ４の説明

フェーズ４では各 $Expanded\_DC_i$ のノード中に輪とショートカットを追加するための情報を集める。第三章で述べたように、$Expanded\_DC_i$ の中で輪を形成する順序は、ノードが属する連結成分の大きさに従い、ショートカットは小さい次数のノード間に追加される。また、使う修復リンク数は式 (3.1) で定義され、機能停止したノード次数和から重複リンク数を除いた数値である。ゆえに、以下の 4 つの情報を集めることを目指す。

1. $Expanded\_DC_i$ の各ノードが属している連結成分のサイズ (輪のため)
2. $Expanded\_DC_i$ の各ノードの次数 (ショートカットのため)
3. $Expanded\_DC_i$ の各ノードが検知した故障ノードの ID と次数 (再利用するリンク数を把握するため)

リーダーノードを決めて各ノードが上記の 4 つの情報をリーダーに送って集めることが、各ノードがフラッディングして情報を集めることより必要な情報量とメッセージ数が少ない。即ち、$Expanded\_DC_i$ のノード数が N 個だと、フラッディングでのメッセージ複雑度は $O(N^2)$ であるが、リーダーを決めると複雑度は $O(N)$ になる。また、後述するフェーズ 5 で輪とショートカットを連結する時、集めた情報を基にしてリーダーが各ノードに連結を依頼することが、分散処理として連結依頼

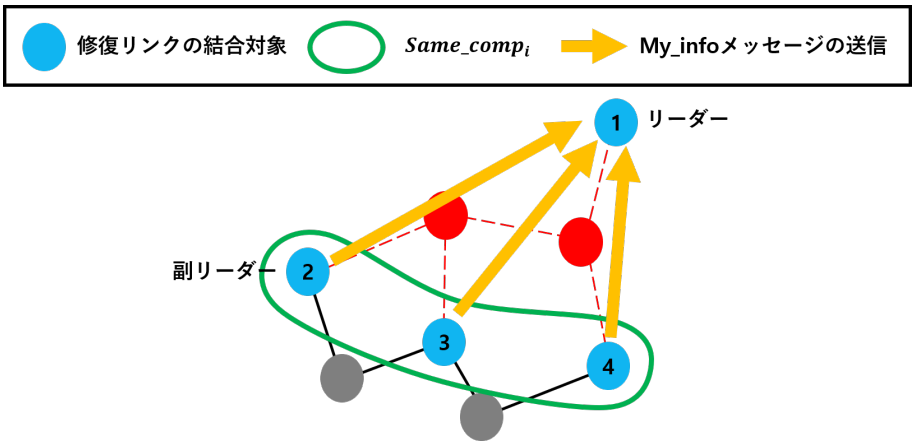

図 5.9: フェーズ 4 の概要図。$Expanded\_DC_i$ の各ノードはリーダーに My_info メッセージを送る。副リーダーではないノードは 0 の $Comp\_size\_info_i$ を送る。リーダー: ノード 1、副リーダー: ノード 2、故障ノード: 赤ノード、一般ノード: 灰色ノード

を受けた各ノードが自ら自分の処理資源を使って依頼を処理することでフェーズが進める長所がある。リーダーを選ぶ既存のアルゴリズム [13] は、ネットワークに生成木を形成して、各ノードがその生成木情報を保持することで、メッセージ交換によりリーダーを決めることができる。しかしながら、このフェーズで各ノードは既にリーダーになる基準 (e.g.ID の高さ) を知っているので、自分の $Expanded\_DC_i$ のノード ID と自分の ID を比べることでリーダーノードの ID を知ることで、メッセージが使わずにリーダーを決める。リーダーとの通信はフェーズ 1 で作ったルーティング表を用いてメッセージを送る。

$Expanded\_DC_i$ の各ノードは上記の情報を一つのリーダーノードに送る。図 5.9 では、$Expanded\_DC_i = 1, 2, 3, 4$ でノード 2,3,4 は同じ連結成分にある。ここで、ノード 1 がリーダーノードである。本研究でリーダーの基準はノード ID のノードの低さとしたが、他の特性 (e.g. ノードのキャパシティー等) に代えることもできる。フェーズ 4 を始めるノードは自分の $Expanded\_DC_i$ で一番低い ID のノードをリーダーにする。ゆえに、ノード 2,3,4 はノード 1 に上記の 4 つの情報 (自分が属している連結成分サイズ情報 ($Comp\_size\_info_i$)、自分の次数 (Deg_info_i)、自分が検知した故障ノード ID(Err_info_i) と次数 (Res_info_i)) を入れて My_info メッセージを送る。これでノード 1 は $Expanded\_DC_i$ にある全ての情報を持つ。これらを用いてリーダーは輪形成の順序を決める。若しノード 2 の連結成分がノード 1 のより大きいと、輪形成の順は $[2, 3, 4, 1]$ になる。しかしながら、使える修復リンクが 1 本しかない場合はノード 2 と 3 を繋ぐよりも、ノード 2 とノード 1 を繋ぐ方が、異なる連結成分を一つすることができる。同じ連結成分にあるノード間を繋ぐことより、異なる連結成分間を繋ぐことが重要であると考える。このために、同じ連結成分にあるノード ($Same\_comp_i$) の中で一つ (副リーダー) だけが連結成分

の大きさを送り、それ以外のノードは 0 の大きさをリーダーに送る。副リーダーは $Same\_comp_i$ で ID が一番低いノードにする。ゆえに、フェーズ 4 を始めるノードはまた自分の $Same\_comp_i$ で一番低い ID のノードを副リーダーにする。リーダーのみならず、副リーダーを考える理由は異なる連結成分間の連結を輪形成で優先するためである。これでノード 1(リーダー) が決める輪形成の順序は、異なる連結成分間の連結を先にするために、一番大きい連結成分にあり副リーダーであるノード 2 が順の先に来て、次に他の連結成分にあるノード 1 が来る。そして、0 の大きさを送ったノード 3 と 4 が来ることで輪形成の順序は $[2, 1, 3, 4]$ のようになる。

リーダーノードは故障ノードの次数を全部足すが、その値から重複リンク数を引く必要がある。故障ノード間にあるリンクを重複リンクで定義する。重複リンクを探すために、リーダーは自分の $Expanded\_DC_i$ の全てのノードが検知した故障ノードの情報 (ID) を活用する必要がある。以前、修復リンクの結合対象は 3 ホップまで直接通信できると前提したから、結合対象は自分から 3 ホップまでのネットワーク構造を知っている。しかしながら、拡張した修復リンクの範囲が 3 ホップより広いと、リーダーは 3 ホップ外にある重複リンクを探索することができない。ゆえに、リーダーは自分以外の $Expanded\_DC_i$ のノードに、集めた故障ノードの情報を入れて Multiple_searching メッセージを送る。

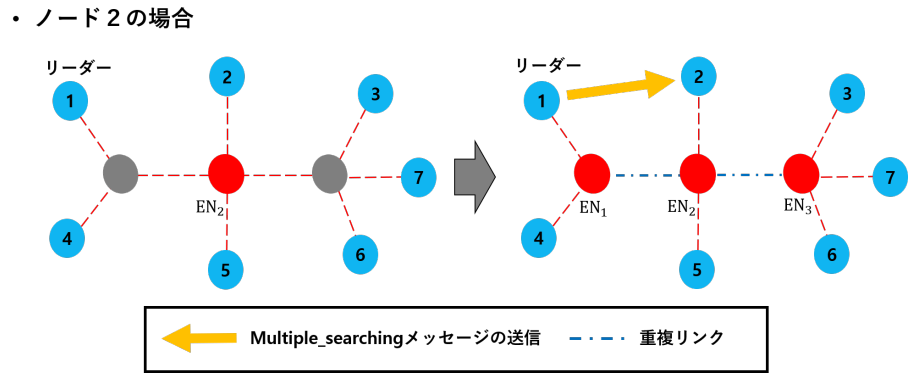

図 5.10: 重複リンク探索の過程。(左) ノード 2 が最初に知っている故障情報。最近接の故障だけを検知したから、他の故障を知らない。(右) リーダーから Multiple_searching メッセージをもらったノード 2。自分の 3 ホップ内にある他の故障情報を知ることで、重複リンクを定義できる

例えば、図 5.10(左) のノード 2 は最初に最近接にあるノード EN_2 の故障だけを知る。リーダーから Multiple_searching メッセージをもらうと、EN_1 と EN_3 の故障も知ることができる。各ノードは自分から 3 ホップまでの局所的なネットワーク構造 (各ノードの ID、次数と連結関係) は分かると仮定した。例えば、ノード 2 は EN_1 の ID と次数、EN_2 の ID と次数及びに、EN_1 と EN_2 が繋がっていることも知っている。EN_1 の故障を知ることから、EN_1 と EN_2 間に重複リンクがあると判断する。故障ノード間にあるリンクは、故障ノードの次数を単純に足した時に重複で数えられるリンクであるから、ノード 2 はそのリンクの ID を Multiple_info

メッセージに入れてリーダーに送る。リンクの ID は両端にあるノード ID のペアで表す。ゆえに、ノード 2 が送る重複リンク ID は (EN_1, EN_2) と (EN_2, EN_3) である。重複リンクの情報を ID で送る理由は、図 5.10 のように、ノード 5 が探索する重複リンクが、ノード 2 が探索した重複リンク ((EN_1, EN_2)、(EN_2, EN_3)) と同じになる場合がある。この時、リーダーは同じリンク ID をもらうことで、重複する情報を判別する。自分以外の $Expanded\_DC_i$ のノードから全てのを Multiple_info メッセージをもらったリーダーは故障ノードの次数和から、持っている重複リンク ID 数を引くことで再利用できるリンク数を求める。実際に使える修復リンク数は、その数値に外部要因からの影響を示す割合 α(式 3.1) を掛けたものであるが、α はこのアルゴリズムでは決められないパラメーターである。このフェーズ 4 を通じてリーダーは輪形成を順序行い、修復に使えるリンク数と $Degree\_info_i$ を通じて次数が低いノードの ID を知る。

5.7.2 フェーズ 4 のアルゴリズムの動作

以下は、フェーズ 4 における変数の意味を説明したものである。

- $Comp\_size\_info_i$: ノード i が持っている $Expanded\_DC_i$ の各ノードが属している連結成分大きさの可変集合。ノード i はフェーズ 3 で既にこれを得ている。
- Deg_info_i : ノード i が持っている $Expanded\_DC_i$ のノード次数の可変集合。ノード i の次数 k_i が要素である。
- Err_info_i : ノード i が持っている、$Expanded\_DC_i$ のノードが検知した故障ノード ID の可変集合。故障ノード ID が要素である。各ノード i は最近接の故障だけを検知できる。
- Res_info_i : ノード i が持っている、$Expanded\_DC_i$ のノードが検知した故障ノード次数の可変集合。故障ノード次数 k_{EN_i} が要素である。集めた Res_info_i を再利用できるリンク数の計算に用いる。
- $Same\_comp_i$: ノード i と同じ連結成分にある、他の修復リンクの結合対象 ID の集合。フェーズ 3 で、この集合のノードとメッセージを交換することで、自分が属する連結成分大きさを計算した。副リーダーは各 $Same\_comp_i$ 中で ID が一番低いノードに定義する。
- $Ring\_seq_i$: リーダーが $Comp\_size\_info_i$ を用いて、定めた輪形成の順序。$Expanded\_DC_i$ のノード ID で構成されている集合である。
- $Multiple\_edge\_info_i$: ノード i が重複リンク定義したリンク ID の可変集合。ノード i はリーダーからもらった Err_info_i で 3 ホップ内で重複リンクの ID を知る。重複リンクの ID は両端にある故障ノード ID で表す。

- $Reusable_resource$: 修復で再利用できるリンク数。Res_info_i にある故障ノード次数を全部足した値から $Multiple\_edge\_info_i$ にある重複リンクの数を引いた値の割合 α 倍である。この変数はリーダーが計算する。

フェーズ 4 を始めるノード i は自分の $Expanded\_DC_i$ で ID が一番低いノードをリーダーに定義して (図 5.11 の 2 行)、自分の $Same\_comp_i$ で ID が一番低いノードを副リーダーに定義する (図 5.11 の 3 行)。自分の ID がリーダーではないノードはリーダーにフェーズ 3 で求めた $Comp\_size\_info_i$、自分の次数 (Deg_info_i)、検知した故障ノードの ID(Err_info_i) と次数 (Res_info_i) を My_info メッセージに入れて送る (図 5.11 の 4~6,10 行)。この場合、副リーダー以外のノードは $Comp\_size\_info_i$ を 0 にして送る (図 5.11 の 7~9 行)。

リーダーは自分のフェーズに構わず、My_info メッセージを受信するたびに、上記の 4 つの $info_i$ を集める (図 5.11 の 13~16 行)。自分の $Expanded\_DC_i$ から全ての My_info メッセージをもらって、フェーズ 3 も終わらせたら、リーダーは集めた $Comp\_size\_info_i$ を用いて輪形成の順序 ($Ring\_seq_i$) を決める (図 5.11 の 17,18 行)。輪形成の順序は $Comp\_size\_info_i$ の大きさに従い、同じサイズのノードについてはランダムに決める。これで $Ring\_seq_i$ は副リーダーの ID が前に、連結成分大きさを 0 で送った他のノードの ID が後ろ順のノード ID 集合になる。その後、自分以外の $Expanded\_DC_i$ のノードに、集めた Err_info_i を入れて Multiple_searching メッセージを送る (図 5.11 の 19,20 行)。

Multiple_searching メッセージを受信したノードは、自分から 3 ホップまでのノードの ID と次数情報を基にして、もらった故障ノード ID(EN) 間にあるリンクを重複リンクで定義する (図 5.11 の 23 行)。重複リンクの ID は両端の故障ノード ID ペア (EN_i, EN_j) にして、その変数名を $Multiple\_edge\_info_i$ とおく (図 5.11 の 24 行)。これを Multiple_info メッセージに入れて LN へ送る (図 5.11 の 25 行)。

自分の $Expanded\_DC_i$ から Multiple_info メッセージを全部もらうと (図 5.11 の 28 行)、集めた $Multiple\_edge\_info_i$ と Res_info_i を利用して再利用できるリンク数 ($Reusable\_resource$) を計算する。それの数値は故障ノードの次数 (Res_info_i) を全部足して、全ての重複リンク ($Multiple\_edge\_info_i$) 数を引いたものに $\alpha(0 < \alpha \leq 1)$ を掛けた値である (図 5.11 の 29 行)。α は外部要因 (e.g. リンクの破壊、追加的なリンク投入) からの影響を示すもので、アルゴリズムでは決められない。このフェーズ 4 が終わって、リーダーは次の情報を知る。

- 再利用できるリンク数 ($Reusable\_resource$)
- 輪を作る順序 ($Ring\_seq_i$)
- ショートカットを追加する、次数が小さいノードの ID(Deg_info_i)

これで、輪とショートカットを連結できる準備は整った。

(Phase4)

1 % Only the node finished the phase 3 %
2 *Determine the leader node ID in* $Expanded\_DC_i$
3 *Determine the subleader node ID in* $Same\_comp_i$
4 $Deg\_info_i \leftarrow \{k_i\}$
5 $Err\_info_i \leftarrow \{EN_i\}$
6 $Res\_info_i \leftarrow \{k_{EN_i}\}$
7 **if** *i is not subleader* **then**
8 $Comp\_size\_info_i = 0$
9 **end if**
10 **Send** $My\_info(Comp\_size\_info_i, Deg\_info_i, Err\_info_i, Res\_info_i)$ **to LN**
11 % End %

12 Process of **Receive** $My\_info(Comp\_size\_info_j, Deg\_info_j, Err\_info_j, Res\_info_j)$ **from j to i**
13 $Comp\_size\_info_i \leftarrow Comp\_size\_info_i \cup Comp\_size\_info_j$ % *updating the info*
14 $Deg\_info_i \leftarrow Deg\_info_i \cup Deg\_info_j$ % *updating the info*
15 $Err\_info_i \leftarrow Err\_info_i \cup Err\_info_j$ % *updating the info*
16 $Res\_info_i \leftarrow Res\_info_i \cup Res\_info_j$ % *updating the info*
17 **if** *i has been received all of My_info from* $Expanded\_DC_i$ **then**
18 *Determine the sequence of connecting ring* $(Ring\_seq_i)$ *by decreasing order of component size*
19 **For all** $k \in Expaned\_DC_i \setminus i$
20 **Send** $Multiple\_searching(Err\_info_i)$ **to k**
21 **end if**

22 Process of **Receive** $Multiple\_searching(Err\_info_j)$ **from j to i**
23 *Within 3hops, finding the edges between error nodes*(EN) *in* Err_info_j
24 $Multiple\_edge\_info_i \leftarrow \{\{EN_i, EN_j\}\}$
25 **Send** $Multiple\_info(Multiple\_edge\_info_i)$ **to LN**

26 Process of **Receive** $Multiple\_info(Multiple\_edge\_info_j)$ **from j to i**
27 $Multiple\_edge\_info_i \leftarrow Multiple\_edge\_info_i \cup Multiple\_edge\_info_j$ % *updating the info*
28 **if** *i has been received all of Multiple_info from* $Expanded\_DC_i$ **then**
29 $Reusable\_resource \leftarrow \alpha \cdot$ (*adding all* k_{EM_i} *in* Err_info_i *and subtract the number of multiple edges*)
30 *Terminate the phase 4 and start the phase 5*
31 **end if**

図 5.11: ノード i でのフェーズ 4 のアルゴリズム

5.8 フェーズ 5: 輪とショートカットを形成する

5.8.1 フェーズ 5 の説明

フェーズ 5 では、輪とショートカットを作る。必要な情報はフェーズ 4 で集めているので、フェーズ 5 ではリーダーが他のノードにメッセージを送って連結を依頼する。リーダーノード i は自分の $Expanded\_DC_i$ の各ノードに Ringmaking メッセージを送る。Ringmaking メッセージには、受信元が連結すべきノード ID が 1 つある。例えば図 5.12 では、修復に使えるリンク総数は 9 本で (図 5.12 の赤点線)、$Expanded\_DC_i$ のノード数は 7 個 (図 5.12 の青ノード) である。輪形成のためには

7 本が必要で、輪形成の順序は [1,4,5,6,7,3,2] とする。まず、リーダーは $Ring\_seq_i$ を用いて、ノード 5 には連結すべきノード 6 の ID を、ノード 3 には連結すべきノード 2 の ID を入れて Ringmaking メッセージを送る。Ringmaking メッセージを受信したノードが、自らメッセージにあるノードと即座に連結をする。もし連結順にメッセージを巡回転送しながら輪を作ると、メッセージの処理対象ではないノードは無駄な待機をするようになる。しかしながら、このフェーズではリーダーが各ノードが連結すべきノードをメッセージに入れて送ることで、他のノードの処理を待たなくて輪形成ができる。

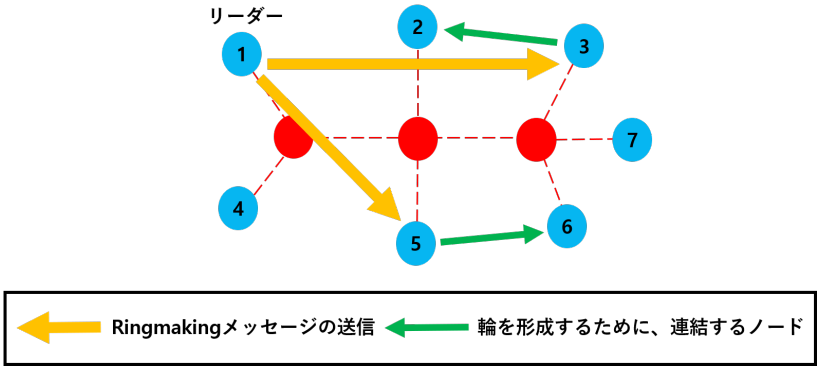

図 5.12: フェーズ 5 の概要図 1。再利用できるリンク数を 9、輪形成の順序が [1,4,5,6,7,3,2] の場合、全てのノードを輪で連結できる

しかしながら、外部要因で使えるリンク数が少ないと、$Expanded\_DC_i$ に部分的な輪を作るしかない。例えば図 5.13 では、使えるリンク数が 4 本だとすると、輪形成はノード 6 まではできる。ゆえに、リーダーはノード 4,5 については図 5.12 と同じく Ringmaking メッセージを送るが、ノード 6 には輪を閉じるためにノード 1 の ID を入れて送る。

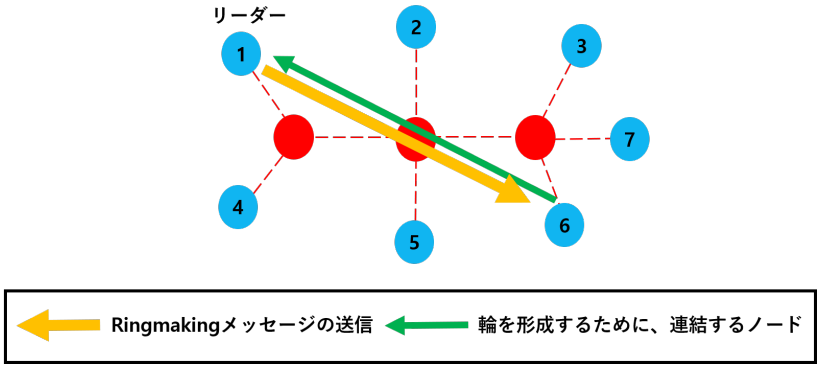

図 5.13: フェーズ 5 の概要図 2。再利用できるリンク数を 4、輪形成の順序が [1,4,5,6,7,3,2] の場合、ノード 6 まで繋いで、LN に戻ることで輪を形成する

リーダーが $Reusable\_resource$ 変数と $Expanded\_DC_i$ のノード数を比較して、輪を全部形成しても使える修復リンク数に余分があると判断すれば、次数が低いノード間にショートカットを更に追加する。次数が低いノード間の次数相関を増加して、ネットワークが正の次数相関を持つようにする。リーダーはフェーズ 4 で $Expanded\_DC_i$ のノードの次数 (Deg_info_i) も全部収集したことで、それを用いて次数が小さい順にノードを 2 つ ($Pair_1$ 、$Pair_2$) を決める。リーダーは片方 ($Pair_2$) に $Pair_1$ の ID を入れた Adding_SC メッセージを送る。Adding_SC メッセージをもらったノードは $Pair_1$ と新しい連結を作る。その後、リーダーは $Reusable\_resource$ 変数と Deg_info_i 集合を更新し、使える修復リンクがなくなるまで Adding_SC メッセージの送信を繰り返す。

5.8.2 フェーズ 5 のアルゴリズムの動作

以下は、フェーズ 5 における変数の意味を説明したものである。

- $Reusable\_resource$: フェーズ 4 で計算した、ある $Expanded\_DC_i$ の修復に使えるリンク数。輪やショートカットとしてリンクを作ると、変数 $Reusable\_resource$ の値を一ずつ減らす。

- $Next\_up$: 輪を形成するために、ノード i が連結すべきノードの ID。例えば、$Ring\_seq_i$ が $1, 2, ..., i, j, ...n$ であると、ノード i が連結するノード j はである。また、最後のノード n が連結するノードは最初のノード 1 である。

- $Pair_1, Pair_2$: ショートカットが追加される結合対象の ID。リーダーは Deg_info_i を用いて $Pair_1, Pair_2$ を決める。

リーダーノード i は、フェーズ 4 で $Reusable\_resource$、$Ring\_seq_i$、Deg_info_i を得ている。リーダーはまず、$Reusable\_resource$ と自分の $Expanded\_DC_i$ の大きさを比べる。$Reusable\_resource$ 変数が $Expanded\_DC_i$ の大きさより大きいと (図 5.14 の 7 行)、$Expanded\_DC_i$ のノードを全部輪で連結できるリンクがあることを意味する。ゆえに、リーダーは $Ring\_seq_i$ を用いて、$Expanded\_DC_i$ のノードに輪を形成するために連結すべきノードの ID を Ringmaking メッセージに入れて送る (図 5.14 の 10 行)。つまり、各ノードは $Ring\_seq_i$ 集合中で自分の次にあるノードの ID($Next\_up$) をもらうようになり (図 5.14 の 9 行)、Ringmaking メッセージを受信したノードがもらった $Next\_up$ ノードと連結することで輪を形成できる (図 5.14 の 21~22 行)。しかしながら、$Reusable\_resource$ が $Expanded\_DC_i$ の大きさより小さいと (図 5.14 の 2 行)、$Expanded\_DC_i$ の全てのノードを輪で連結できない。ゆえに、形成順序の途中で切れて部分的な輪を作る。$Reusable\_resource$ 変数の分だけ輪を作れば良く、リーダーは輪で連結できないノードには Ringmaking メッセージを送らない。つまり、リーダーは $Ring\_seq_i$ 集合中で $Reusable\_resource$ 変数番目のノードまで $Next\_up$ を Ringmaking メッセージに入れて送る (図 5.14 の 3~5 行)。

リーダー自身は自分にはRingmakingメッセージを送らず、$Next_up$ノードと連結する (図5.14の6,11行)。輪を作るために、最後の順にあるノードがRingmakingメッセージでもらう$Next_up$ノードはリーダーである。リーダーが$Expanded_DC_i$のノード数と$Reusable_resource$変数を比較してリンク数に余分があると判断する場合、次数が小さいノード間にショートカットを追加する。

リーダーはDeg_info_iを用いて、$Expanded_DC_i$中で次数が一番小さいノード($Pair_2$)にAdding_SCメッセージを送ってショートカットの生成を依頼する (図5.14の14,15行)。Adding_SCメッセージをもらったノードは二番目に次数が小さいノードのID($Pair_1$)と新しく連結する (図5.14の24行)。これで一つのショートカットを作ったので、リーダーは$Reusable_resource$変数で1を引き、次数情報(Deg_info_i)で$Pair_1$、$Pair_2$の数値に1を足す (図5.14の16,17行)。この過程を再利用できるリンクがなくなるまで繰り返す (図5.14の13~18行)。

(Phase 5)

1 % Only the leader node finished the phase 4 %
2 **if** *Reusable_resource is smaller than* $|Expanded\_DC_i|$ **then**
3 **For all** $k \in Ring\_seq_i\{1: Reusable\_resource\}$
4 $Next\_up \leftarrow$ *the next node ID of k in order of* $Ring\_seq_i$
5 **Send** *Ringmaking* $(Next\_up)$ **to k except LN**
6 *Making a new connection with node Next_up*
7 **else**
8 **For all** $k \in Ring\_seq_i$
9 $Next\_up \leftarrow$ *the next node ID of k in order of* $Ring\_seq_i$
10 **Send** *Ringmaking* $(Next\_up)$ **to k except LN**
11 *Making a new connection with node Next_up*
12 *Subtract* $|Expanded\_DC_i|$ *from Reusable_resource*
13 **Do**
14 $Pair_1, Pair_2 \leftarrow$ *the node having the smallest and the 2nd smallest degrees*
15 **Send** $Adding\_SC(Pair_1)$ **to** $\mathbf{Pair_2}$
16 *Add 1 to degrees of* $Pair_1$ *and* $Pair_2$ *in* $Degree\_info_i$
17 *Subtract 1 from Reusable_resource*
18 **Until** $Reusable\_resource = 0$
19 **end if**
20 % End %

21 Process of **Receive** *Ringmaking* $(Next\_up)$ **from j to i**
22 *Making a new connection with node Next_up*

23 Process of **Receive** $Adding\_SC(Pair_1)$ **from j to i**
24 *Make a new connection with* $Pair_1$

図 5.14: ノードiでのフェーズ5のアルゴリズム

これら図5.3、5.6、5.8、5.11、5.14で記述した、5段階のアルゴリズムを各ノードで実行することで、逐次的なアルゴリズムを用いて得た第4章の結果と同じ結果が得られると予想している。

第6章　おわりに

本論文では、被害を受けたノードに輪形成して、その輪上でループを強化する自己修復法を提案した。提案した修復法は、機能停止したノードのリンクを再利用する修復資源をパラメートリックに考えた。現実ネットワークのデータに対して、従来の自己修復方法と比較した結果を以下にまとめる。

- 次数順攻撃を受けたネットワークに提案する修復法を適用すると、従来の修復法より、高い連結性を維持できる。
- 修復に使えるリンク数が、機能停止したノード次数和の割合 α が 0.5 以上あると、提案法は次数順攻撃に対して高い頑健性と経路効率を持つネットワークを再構築できる。
- 従来法で修復すると、攻撃率 q や修復リンク割合 α の数値と余り関係なく、ほぼ一定な追加ポート数が必要である。提案法は修復リンクが十分なら正則グラフになるよう修復するから、多くの追加ポート数が必要である。
- 提案法を自律分散アルゴリズムに記述することで、中央集権的なアルゴリズムが含む問題の改善を図った。

また、提案した自律分散アルゴリズムを利用して、逐次的なアルゴリズムで実験した結果と同じ結果を得られると予想する。記述した分散アルゴリズムの有効性を数値シミュレーションで検証することは今後の課題となる。

発表論文·口頭発表

1. Jaeho Kim, and Yukio Hayashi, "Self-Healing Strategy for Improving Robustness in Limited Resource", International Conference on Complex Systems 2020, Book-of-Abstracts, pp.166, Online, Dec. 4-11, 2020.

参考文献

[1] R. Albert et al, "Error and attack tolerance of complex networks", Nature, 406(6794), 378-382, (2000).

[2] C. Folke, "Resilience: The emergence of a perspective for social–ecological systems analyses", Global environmental change, 16(3), 253-267, (2006).

[3] A. Braunstein et al, "Network dismantling", Proc. Natl. Acad. Sci. U.S.A. 113(44), 12368-12373, (2016).

[4] Y. Hayashi, N. Uchiyama, "Onion-like networks are both robust and resilient", Sci. Rep. 8(1), 1-13, (2018).

[5] L. K. Gallos, N. H. Fefferman, "Simple and efficient self-healing strategy for damaged complex networks", Phy. Rev. E, 92(5), 052806, (2015).

[6] C. M. Schneider et al, "Mitigation of malicious attacks on networks", Proc. Natl. Acad. Sci. U.S.A., 108(10), 3838-3841, (2011).

[7] F. Morone, H. A. Makse, "Influence maximization in complex networks through optimal percolation", Nature, 524(7563), 65-68, (2015).

[8] T. Tanizawa et al, "Robustness of onionlike correlated networks against targeted attacks", Phys. Rev. E 85(4), 046109, (2012).

[9] C. Masaki, Y. Hayashi, "A loop enhancement strategy for network robustness", Applied Network Science, 6, 3, (2021).

[10] M. Stippinger, J. Kertész, "Enhancing resilience of interdependent networks by healing", Physica A: Statistical Mechanics and its Applications, 416, 481-487, (2014).

[11] Y. Hayashi et al, "Effective Self-Healing Networks against Attacks or Disasters in Resource Allocation Control", Proc. of 12th Int. Conf. on Adaptive and Self-Adaptive Systems and Applications, 85-91, (2020)

[12] L. Nancy A. "Distributed algorithms", Elsevier, (1996).

[13] 亀田恒彦, 山下雅史, ”分散アルゴリズム”, 近代科学社, (1994).

[14] E. P. Duarte Jr, A. Weber, ”A distributed network connectivity algorithm”, ISADS 2003, 285-292, IEEE, (2003).