Vol.2022-OS-155 No.8
2022/5/27

輻輳制御アルゴリズムの自動生成における Structured Grammatical Evolution 適用の検討

広瀬 智之[1,a)]　阿部 洋丈[1,b)]　岡 瑞起[1,c)]

概要：ネットワークにおいては、パケットが集中し混雑する輻輳といった現象がしばしば観測される。この輻輳を制御するためのアルゴリズムに関する研究が続けられている。しかし、現在までに決定版とされる輻輳制御アルゴリズムは開発されていない。これは、インターネットが常に変化し続けており、ある時点では適切なアルゴリズムが将来にわたっても適切であり続けることが難しいことが原因として挙げられる。そこで、本研究では変化し続けるインターネットに対応した輻輳制御アルゴリズムの自動生成を目標とする。アルゴリズムの自動生成においては、環境とエージェントが共進化する強化学習アルゴリズムを用いる。エージェントの進化には Structured Grammatical Evolution を用いる。Structured Grammatical Evolution とは、出力される文字列の文法の正しさが保証される進化的アルゴリズムである。本論文では、生成したアルゴリズムの中で特徴的な個体とそれ以外を比較し、その変異がどのように他の個体との差を生み出したのかを考察する。

1. 背景

ネットワークではたびたび通信遅延が起こる。通信遅延の原因の一つとしては、通信路に対してパケット量が多すぎることによる輻輳が挙げられる。輻輳とは、ネットワークのある一箇所にパケットが集中し、通信バッファからあふれることによって、通信が途絶えてしまう現象である。

パケットの量を制御し、ネットワークの通信を円滑に進めるための手法として輻輳制御 [1] がある。輻輳制御では、まず各ノードが現在の通信の状況を観察し、送信パケット量を徐々に増やしていく。そして、通信経路上の混雑の兆候を検知したら、送信パケット量を調節する。以上のように輻輳制御は通信を正常に保ちつつ、通信量を最大化するようにはたらく。

輻輳制御において、送信パケット量の増やし方、異常検知時の送信パケット量の調整方法は、各輻輳制御アルゴリズムによって異なる。輻輳制御アルゴリズムは数多くあるが、現在では NewReno [2] や CUBIC [3] などが用いられることが多い。

インターネットはその誕生以来、その用途、規模は複雑多様化し続けている。それに対して既存の輻輳制御アルゴリズムも進化する必要がある。しかし、現在よく使われている輻輳制御アルゴリズムは人の手で作られたものである。人力でアルゴリズムを開発するのには限界があり、また現実のネットワークに追従するのも難しい。また、未だ人間が思いついていないアルゴリズムもある可能性がある。

そこで、本研究では輻輳制御アルゴリズムの自動生成を試みる。Endo らの研究 [4] では、輻輳制御アルゴリズムの自動生成を目的に、強化学習アルゴリズムの POET と進化的アルゴリズムの Grammatical Evolution を組み合わせたシステムを構築している。POET はエージェントと環境が共進化する強化学習アルゴリズムである。Grammatical Evolution とは、出力される文字列の文法の正しさが保証される進化的アルゴリズムである。文法の正しさが保証されていることで、プログラムとして意味のある文が生成できる。本研究では、Endo らの研究をベースに、Grammatical Evolution の改良版である Structured Grammatical Evolution を用いたシステムを構築した。これらを組み合わせた実装を用いて、生成した輻輳制御アルゴリズムの中で輻輳ウィンドウサイズの増減が特徴的なものとそうでないものを比較した。比較の結果、特徴的な個体は現在の輻輳ウィンドウサイズを元に次の輻輳ウィンドウサイズを決定していることが確認された。特徴的でない個体は、輻輳制御アルゴリズムに意味のある表現を確認できなかった。

[1] 筑波大学
University of Tsukuba, 1-1-1 Tennoudai, Tsukuba, Ibaraki 305–8573, Japan
[a)] hirose@osss.cs.tsukuba.ac.jp
[b)] habe@cs.tsukuba.ac.jp
[c)] mizuki@cs.tsukuba.ac.jp

2. 関連研究

本章では、輻輳制御アルゴリズムの研究と、その自動生成について関連する研究を述べる。また輻輳制御アルゴリズムの自動生成において用いられている手法も紹介する。

2.1 輻輳制御アルゴリズムの研究

輻輳制御アルゴリズムの研究は 1980 年代から行われており、数多くの輻輳制御アルゴリズムが考案されている。輻輳制御アルゴリズムには大きく分けて、Loss-based 方式と Delay-based 方式がある。どちらの方式も、輻輳ウインドウと呼ばれる一度に送信するデータ量を調節することで輻輳制御を試みるが、輻輳が起きたと判断するために用いる指標が異なっている。

Loss-based 方式は、ネットワークの輻輳状態を測る指標としてパケットロス数を用いる方式である。パケットロス数は、経路上で廃棄されたパケットの数のことである。ネットワークが混雑するほど、経路ノード上でのバッファオーバーフローなどで廃棄されるパケットが多くなると考えられる。この考え方をもとに、Loss-based 方式では、パケットロスが起きるまでは徐々に輻輳ウインドウを増加させ、パケットロスが検知されたら輻輳ウインドウを減少させる。Loss-based 方式の輻輳制御アルゴリズムには、New Reno [2] や CUBIC [3] などがある。

Delay-based 方式は、指標としてラウンドトリップタイムを用いる方式である。ラウンドトリップタイムとは、データを送信してから応答が返ってくるまでにかかる時間である。ネットワークが混雑している場合、経路上のノードのバッファにデータが多く蓄積されているので、新しく来たパケットを送信するのに時間がかかると予想される。この考え方をもとに、Delay-based 方式では、ラウンドトリップタイムが閾値を超えるまでは輻輳ウインドウを増加させ、ラウンドトリップタイムが閾値を超えたら輻輳ウインドウを減少させる。Delay-based 方式の輻輳制御アルゴリズムには、Vegas [5] などがある。

2.2 輻輳制御アルゴリズムの自動生成

輻輳制御アルゴリズムを自動生成する研究としては、TCP ex Machina [6] がある。この研究は人間が考えたアルゴリズムのパラメータを自動調整することによって、輻輳制御アルゴリズムを環境にフィットさせることを試みている。TCP ex Machina では、Remy という輻輳制御アルゴリズムを開発し、New Reno や Vegas といった既存の輻輳制御アルゴリズムに対して優位な性能を示している。しかし、Remy はあくまでベースとなる制御式のパラメータを自動調整することにとどまっており、輻輳制御アルゴリズム自体を生成しているわけではない。

⟨expr⟩::= ⟨expr⟩⟨op⟩⟨expr⟩
| (⟨expr⟩⟨op⟩⟨expr⟩)
| ⟨pre-op⟩(⟨expr⟩)
| ⟨var⟩
⟨op⟩::= + | - | × | /
⟨pre-op⟩::= sin
⟨var⟩::= x | 1

図 1: BNF の例

2.3 共進化する強化学習アルゴリズム

アルゴリズム自体を自動生成する研究もある。Endo らの研究 [4] では、輻輳制御アルゴリズムをエージェント、ネットワークを環境とみなし、強化学習の枠組みでアルゴリズムを進化させていく手法を取っている。Endo らの研究では、強化学習アルゴリズムに POET [7] を用いている。POET は従来のエージェントのみ進化させる強化学習アルゴリズムとは違い、環境を複雑化、多様化することによって、さらなるエージェントの進化を促す方式である。

POET は、まず簡単な環境をランダムで用意する。環境とエージェントのペアを作成し、エージェントを進化させていく。次に環境を複雑化させて次のペアを作成し、またエージェントを進化させる試行を繰り返す。このとき、環境の複雑化はエージェントにとって簡単すぎず難しすぎない難易度であった場合に行われる。

3. 文法に則った文字列を生成する進化的アルゴリズム

Endo らの研究では、エージェントとして輻輳制御アルゴリズムを、環境としてネットワークを適用し、エージェントの進化を Grammatical Evolution [8] という進化的アルゴリズムで行っている。

Grammatical Evolution は、定めた文法に対して、その文法に則った文字列を生成、進化させるアルゴリズムである。文法は Backus Naur 記法 (BNF) で表現されたものを用いる。

図 1 は、BNF で定義した文法の例である。Grammatical Evolution では、遺伝子型と呼ばれる整数列を一定の規則に従って BNF と対応付ける。BNF、つまり文法は表現型と呼ばれる。対応規則は

$$rule = i \bmod N_{rules}$$

で表される。ここで、i は整数値、N_{rules} は BNF の非終端記号のルール数である。例として**図 2** の整数列を扱う。

BNF の開始記号を <expr> とすると、 <expr> のルールの数は BNF の例から 4 である。図 2 の最初の整数に規則を用いると 220 mod 4 = 0 であるため、<expr> の 0 番目のルールを適用する。したがって、

220, 240, 220, 203, 101, 53, 202, 203, 102, 55, 220, 202, 241, 130, 37, 202, 203, 140, 39, 202, 202, 102

図 2: 整数列の例

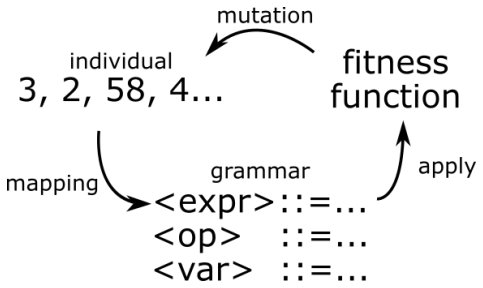

図 3: GE の処理フロー

<expr>→<expr><op><expr> と置き換えられる。次に <expr><op><expr> の一番左を置き換えることを試みる。図 2 の 2 番目の整数に規則を用いると 240 mod 4 = 0 であるため、<expr> の 0 番目のルールを適用する。したがって、<expr><op><expr>→<expr><op><expr><op><expr> と置き換えられる。この操作をこれ以上置き換えられなくなるまで行うと、最終的には

```
1-sin(x)× sin(x)-sin(x)× sin(x)
```

というプログラムコードが得られる。

整数列から文法に則ったプログラムコードへの変換を行った後、適応度関数と呼ばれる関数にかけ、環境に対してどれだけ適応しているかを測定する。適応度関数はその環境によって違ったものになる。

その適応度によって、整数列の変異を行う。変異とは、整数列の一部を変更することである。変異には、交配と突然変異の 2 種類がある。交配は、2 つの整数列の一部を交換する処理である。突然変異は、整数列の一部をランダムに変更する処理である。

Grammatical Evolution の処理フローを図に表すと、**図 3** のようになる。図 3 のサイクルを回していくことによって進化をさせていく。

4. Grammatical Evolution の発展形

Grammatical Evolution には発展形がある。Structured Grammatical Evolution [9] は、Grammatical Evolution にある種の構造を導入することによって、 [10] や [11] などで論じられている Grammatical Evolution の欠点をカバーし、性能を向上させている。

Grammatical Evolution の欠点として、冗長性の高さと局所性の低さが挙げられる。まず、冗長性の高さとは、遺伝子型の変異が表現型の変異に結びつかないことが多い、という意味である。例として、整数 17 と 29、$N_{rules} = 4$

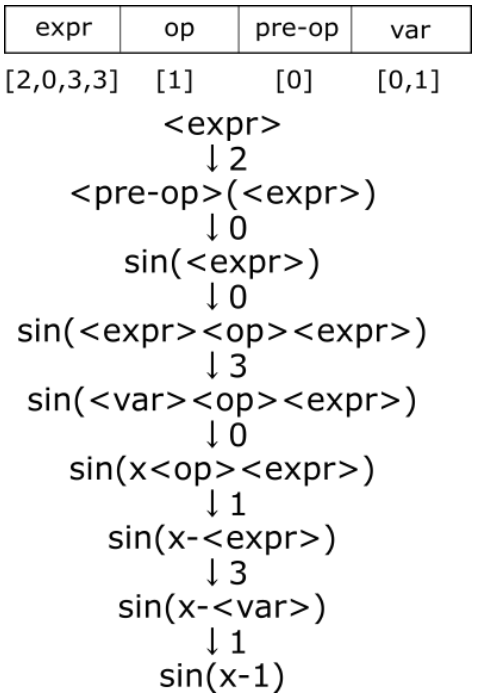

図 4: SGE のマッピングプロセスの例

の場合を考える。整数 17 と整数 29 の対応規則をそれぞれ $rule_{17}$ と $rule_{29}$ とすると、$rule_{17}$ と $rule_{29}$ はどちらも 1 であるため、整数値は違っていても対応する文法は同じ、ということになる。次に、局所性の低さとは、遺伝子型のある整数値の変異の影響が、周囲の整数値における表現型への変換へ伝播する、という意味である。 [11] では、遺伝子型においての最小単位の変異、つまり 1bit の変異が、表現型の最小単位の変異、つまり文法の 1 文字の変異に結びついている割合は、わずか 10% であることがわかっている。

このような欠点に対処するため、Structured Grammatical Evolution では遺伝子型において、Grammatical Evolution で用いられている整数列の代わりに、**図 4** のように整数の配列を用いて処理を行う。配列中の整数値は、0 から $(N_{rules} - 1)$ の間から生成される。このようにすることで、遺伝子型の整数値の種類が N_{rules} と対応するため、遺伝子型の変異が表現型の変異と一対一に結びつくことになり、冗長性の高さが改善される。また、整数値を各非終端記号毎の配列に格納することで、変異の影響がその非終端記号に閉じ込められる。そのため、局所性の低さが改善される。

5. 提案手法

本章では、輻輳制御アルゴリズムの自動生成の研究に対しての提案と、その実装について示す。

5.1 手法

Endo らの研究 [4] では、エージェントの進化に Grammatical Evolution を用いている。本論文では、Endo らの研究に対して、Grammatical Evolution の代わりに

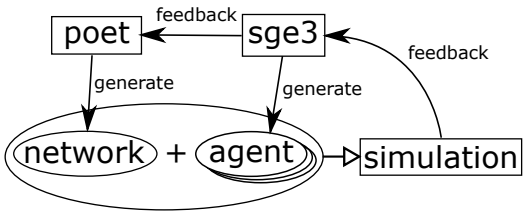

図 5: システムの概略図

tured Grammatical Evolution を用いて輻輳制御アルゴリズムの生成を行う手法を提案する。

5.2 実装

実装は、ns3-gym [12] をベースに行った。ns3-gym は、ネットワークシミュレータである ns-3 [13] をもとに OpenAI Gym [14] を組み込み、ネットワークシミュレーション上で強化学習を利用できるようにしたフレームワークである。

本研究では、ns3-gym 上に POET の実装である poet[*1] と Structured Grammatical Evolution の実装である sge3[*2] を組み込み、ns3-gym の機能を用いてネットワークシミュレーションを行った。

5.2.1 システムの概略

図 5 は実装したシステムの概略図である。まず、poet モジュールが環境となるネットワークを生成する。ネットワークに対して、sge3 モジュールがエージェントとなる輻輳制御アルゴリズムを複数生成する。ネットワークと輻輳制御アルゴリズムのペアを作成し、そのペアを用いてシミュレーションを行う。結果を sge3 モジュールに転送し、個体のリストをスループットでソートする。ソートした個体のうち上位 N_{top} 個体を保存し、変異させたのち次のループに使用する。この処理を繰り返し、全てのペアのシミュレーションが終了したら、poet モジュールにシミュレーションの結果を通知する。poet モジュールは結果を見て、難易度が適切だった場合再度ネットワークの生成を行う。

5.2.2 poet モジュール

図 6 は poet モジュールの概略図である。poet モジュールでは、sge3 モジュールから通知されたシミュレーションの結果を受け取り、エージェントにとってその環境が適切な難易度だったかどうかを判定する。簡単すぎず、難しすぎない難易度だった場合、環境を変異させ、新たにペアを作成する。難易度の調整に用いる指標としては、スループットを採用した。上限値 T_{max} と下限値 T_{min} の間にスループットが収まっている場合に環境を変異させる。ここで、エージェントは輻輳制御アルゴリズム、環境はネットワークである。

[*1] https://github.com/uber-research/poet
[*2] https://github.com/nunolourenco/sge3

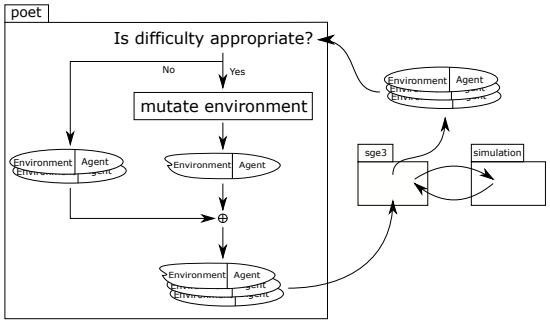

図 6: poet モジュールの概略図

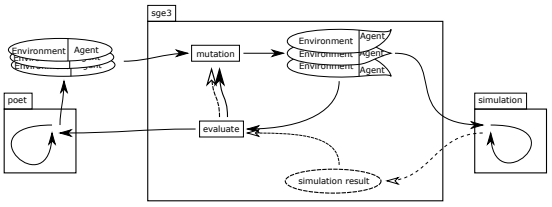

図 7: sge3 モジュールの概略図

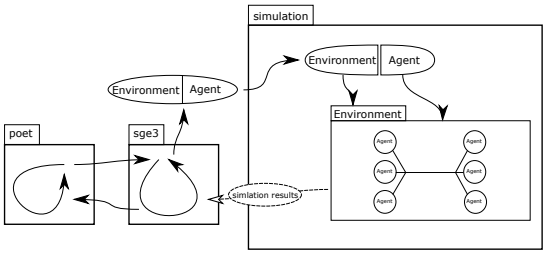

図 8: simulation モジュールの概略図

5.2.3 sge3 モジュール

図 7 は sge3 モジュールの概略図である。sge3 モジュールでは、poet モジュールから受け取った環境とエージェントのペアを変異させ、simulation モジュールに渡す。simulation モジュールでシミュレーションを行った後、シミュレーションの結果を受け取る。シミュレーションの結果と、環境とエージェントのペアを用いて、エージェントの評価を行う。評価結果を用いて、さらにエージェントの変異を行う。このサイクルを繰り返し、サイクル回数が規定を超えたら poet モジュールに環境とエージェントのペアを渡す。

5.2.4 simulation モジュール

図 8 は simulation モジュールの概略図である。simulation モジュールでは、ネットワークと輻輳制御アルゴリズムのシミュレーションを行う。ネットワークの構成に従ってノードを配置し、輻輳制御アルゴリズムを適用する。シミュレーションを行い、その結果を sge3 モジュールに通知する。

表 1: 自動生成の条件

条件	設定値
環境生成イテレーション	30
エージェント生成イテレーション	50
1 つの環境で生成する世代数	4
交配確率	0.9
突然変異確率	0.1
最小文法木サイズ	8
最大文法木サイズ	17
N_{top}	5
T_{max}	500
T_{min}	3000

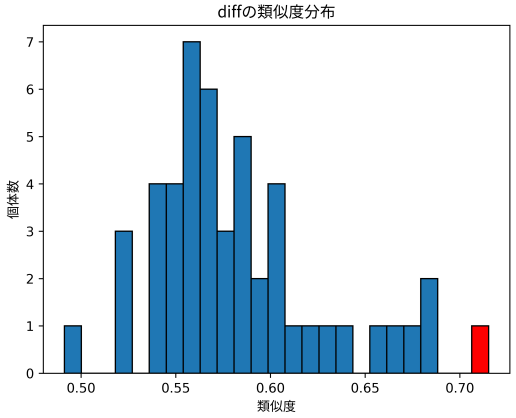

図 9: 個体 G とその同世代の各個体との類似度の分布。赤い部分が個体 B を表す。

6. 実験と考察

5 章で実装したシステムを用いて、輻輳制御アルゴリズムの自動生成を行う。また生成した輻輳制御アルゴリズムを用いたネットワークシミュレーションを行う。それらの結果を用いて、得られた輻輳制御アルゴリズムに対して、効果的な輻輳制御ができているか、輻輳ウィンドウサイズの変化グラフやプログラムの解読をすることで考察を行う。

6.1 輻輳制御アルゴリズムの自動生成

輻輳制御アルゴリズムの自動生成には、**表 1** のような条件で生成を行った。

環境生成イテレーション、エージェント生成イテレーション、世代数、交配確率、突然変異確率は先行研究を参考にした。最小文法木サイズ、最大文法木サイズは sge3 のデフォルト値を参考にした。BNF の文法定義はプログラム 6 のものを用いた。なお、プログラム 6 には適宜改行を入れてある。

生成の結果、1900 の個体が得られた。得られた個体は Python 言語で記述されている。その中で、特徴的な輻輳ウィンドウの増減の示す個体が 1 つ得られた。以降、この個体を**個体 G**、その個体のコードを**コード G** と呼ぶ。なお、その他の個体にはこのような特徴が見られなかった。

次に、個体 G との比較のため、同世代の個体の中で個体 G と類似度が最も高い個体である**個体 B** を選出した。類似度は Python の difflib ライブラリを用いて算出し、同世代の 50 個体の類似度の分布として**図 9** が得られた。図 9 において、赤くなっている部分が個体 B である。なお、コード G と**コード B**（個体 B のコード）の類似度は約 0.715 であった。

BNF から定義されるプログラムの構造をプログラム 1 に示す。`obs` はリストオブジェクトであり、様々な情報が格納されている。ここで、**表 2** に `obs` 中に含まれている情報の中で必要なもののリストを示す。`state` はネットワークの状態を表すものであり、各数値は**表 3** に記載されている意味である。

表 2: obs のパラメータの解説

obs	パラメータ	意味
`obs[4]`	輻輳ウィンドウサイズ	輻輳制御で調節するパケットのサイズ
`obs[5]`	スロースタート閾値	増加分を変動させる境界値
`obs[6]`	セグメントサイズ	一度に送信するデータサイズ
`obs[7]`	acked セグメントの数	返答があったセグメント
`obs[8]`	inflight バイト数	ネットワーク上に存在するデータの量
`obs[9]`	ラウンドトリップタイム	データを送信して返信が来るまでの時間
`obs[10]`	過去の RTT の最小値	

表 3: state の状態

state	状態	意味
`state == 0`	OPEN	正常な状態
`state == 1`	DISORDER	重複 ACK を受信した状態
`state == 2`	RECOVERY	3 度重複 ACK を受信した状態
`state == 3`	LOSS	ACK のタイムアウトを検知した状態

プログラム 1: 生成されたアルゴリズムの構造

```
class CongestionControlAlgorithm():
    def get_action():
        if state == OPEN:
            ...
        elif state == DISORDER:
            ...
        elif state == RECOVERY:
            ...
        elif state == LOSS:
            ...
```

6.2 シミュレーション

前節で生成した輻輳制御アルゴリズムを、ネットワークシミュレーションで検証する。ここで、検証環境は**表 4** の

表 4: シミュレーションの条件

条件	設定値
伝送路エラーレート	0.001
シミュレーション秒数	60
ネットワーク構成	図 10

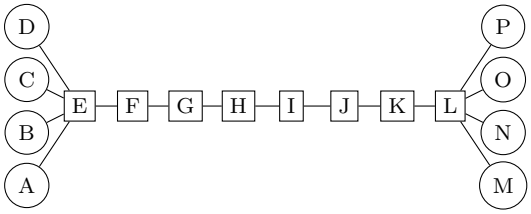

図 10: ネットワーク構成

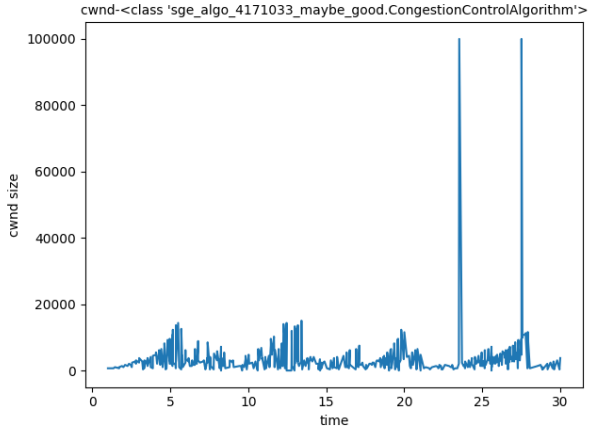

図 11: 生成した輻輳制御アルゴリズム 1 の輻輳ウィンドウの変化

ような環境である。**図 10** では、ダンベル型のネットワークを使用した。葉の数は 4 で、中継ノードが 8 のものを生成し、両端の葉のペア全てに生成した輻輳制御アルゴリズムを用いた通信を行わせた。個体 G を用いてシミュレーションを行った結果、**図 11** のような輻輳ウィンドウの変化が観測された。また。個体 B を用いてシミュレーションを行った結果、**図 12** のような輻輳ウィンドウの変化が観測された。なお、生成された 1900 の個体のうち、個体 G 以外の個体は図 12 のような、輻輳ウィンドウの変化が乱高下している挙動を示した。

6.3 実験結果の考察

本節では、ネットワークシミュレーションの結果や、生成されたプログラムの記述から、効果的な輻輳制御が行われているかを考察する。そのために、図 11 と図 12 を見比べ、輻輳ウインドウサイズの変化を比較する。また、コード G とコード B を比較し、生成されたアルゴリズムがどのような挙動をしているか考察する。

図 11 を見ると、輻輳ウインドウの大きさが上下に変化しながら、ある程度の大きさまで徐々に上昇しているのが

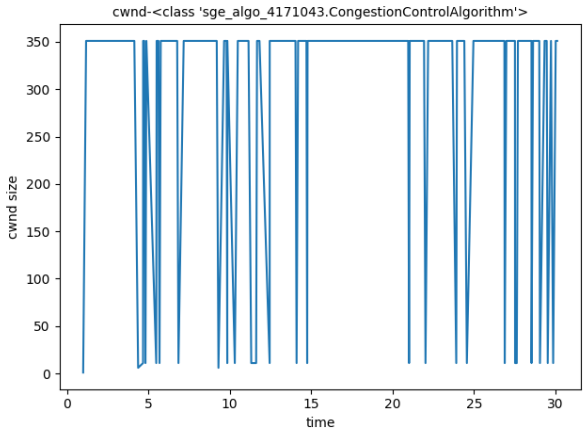

図 12: 生成した輻輳制御アルゴリズム 2 の輻輳ウィンドウの変化

確認できる。また、頻度は不明ではあるが、輻輳ウインドウサイズが上限値の 100000 まで上昇していることが確認できる。

コード G の抜粋であるプログラム 2[*3]を見ると、正常な状態、つまり状態が OPEN のときは、現在の輻輳ウインドウをもとに輻輳ウインドウを増加させている。また、プログラム 4 を見ると、ネットワークが異常な状態、つまり状態が DISORDER、RECOVERY、LOSS のときは、プログラムからは、輻輳ウィンドウサイズを一概に減少させているとは読み解くことはできない。

プログラム 2: コード G の正常状態の輻輳ウィンドウサイズの増加

```
if state == OPEN:
    new_cWnd = 2 + segment_size
        + cwnd
```

プログラム 3: コード B の正常状態の輻輳ウィンドウサイズの増加

```
if state == OPEN:
    new_cWnd = 19 + segment_siz
        e - bytes_inflight - 8 %
         segment_size
```

個体 B に関しては、図 12 を見ると、輻輳ウインドウの大きさが乱高下しており、効率的な輻輳制御ができていないことがわかる。

コード B の抜粋であるプログラム 2 を見ると、正常な状態のときは輻輳ウィンドウサイズを増加させているが、その増分が大きすぎるため、図 12 のように乱高下する結果となったと考えられる。また、ネットワークが異常な状態のときであるプログラム 4 を見てみると、ところどころ定数になっているのが確認される。これによって、輻輳ウィ

[*3] 読みやすさのため、`obs[...]` を、それの意味するところの値の名称に置き換えてある。本文中の他のプログラムも同様。

ンドウサイズがごく小さくなっているのではないかと考えられる。

プログラム 4: コード G の異常状態の輻輳ウィンドウサイズの増加

```
        elif state == DISORDER:
            if ssThreshold > RTT:
                new_cWnd = segment_size
                new_ssThresh = ssThresh
                    old
            else:
                new_cWnd = ssThreshold
                new_ssThresh = 9
        elif state == RECOVERY:
            if RTT == num_acked_segmen
                t:
                new_cWnd = 9
                new_ssThresh = num_acke
                    d_segment % 6
            else:
                new_cWnd = bytes_inflig
                    ht
                new_ssThresh = 4
        elif state == LOSS:
            new_cWnd = bytes_inflight
            new_ssThresh = 6 - 2 % segm
                ent_size + cwnd * min_RT
                T
```

プログラム 5: コード B の異常状態の輻輳ウィンドウサイズの増加

```
        elif state == DISORDER:
            if num_acked_segment >= RT
                T:
                new_cWnd = num_acked_se
                    gment
                new_ssThresh = 6
            else:
                new_cWnd = min_RTT
                new_ssThresh = cwnd
        elif state == RECOVERY:
            new_cWnd = min_RTT
            new_ssThresh = 8 + 3 * ssTh
                resh
        elif state == LOSS:
            if RTT >= cwnd:
                new_cWnd = 6
                new_ssThresh = min_RTT
            else:
                new_cWnd = min_RTT
                new_ssThresh = ssThresh
```

本研究には、いくつか改善が必要な点がある。ひとつは、文法の定義がまだ最適ではない点である。例えば、コード G では 16 行目の条件式でスロースタート閾値とラウンドトリップタイムを比較しており、意味のある条件式ではない可能性がある。また、ネットワークが異常な状態になった際に、輻輳ウィンドウサイズを減少させるような文法を定義していない。例えば、コード G では、31 行目でスロースタート閾値が増加している可能性がある。そのため、ネットワークに異常が発生した場合でも、輻輳ウィンドウサイズを増加させてしまう個体が生まれる可能性がある。

良いと思われる個体でなくても、適応度が高いことも問題点である。本研究では、個体の評価にスループットを用いている。そのため、スループットは高いが他の通信を遮ってしまうような輻輳制御アルゴリズムが生き残ってしまうのが問題である。

7. 結論

本研究では、輻輳制御アルゴリズムの自動生成を目的として、Endo らの研究に対して Grammatical Evolution の代わりに Structured Grammatical Evolution を適用したものを実装した。生成された輻輳制御アルゴリズムを、ネットワークシミュレーションにて観察し、その輻輳ウィンドウサイズの変化を確認した。Structured Grammatical Evolution を用いてアルゴリズムを生成した結果、Endo らの研究では確認されなかった、特徴的な輻輳ウィンドウの変化が見られるアルゴリズムを得ることができた。

7.1 今後の課題

本研究にはいくつかの発展の可能性がある。ひとつは文法定義をさらに場合分けして、探索空間を絞り込むことである。現在の BNF の文法定義はまだ最適とは言えず、改善の余地がある。例えば、条件分岐の際に意味のある条件式を生成するように修正するといったことが考えられる。また、ネットワークが異常な状態になった際に、輻輳ウィンドウサイズを減らすように定義するといったことも考えられる。

さらに、Structured Grammatical Evolution 内の評価関数の指標を変更するという手法も考えられる。本研究では、生成された輻輳制御アルゴリズムの評価にスループットを用いたが、別の数値を用いることで、より効率的に個体を探索することができる可能性がある。例として、複数の通信でのネットワークの公平性を指標として用いるということが考えられる。

謝辞 本研究は JSPS 科研費 21H03414 の助成を受けたものです。

参考文献

[1] Widmer, J., Denda, R. and Mauve, M.: A survey on TCP-friendly congestion control, *IEEE network*, Vol. 15, No. 3, pp. 28–37 (2001).
[2] Nishida, Y.: The NewReno modification to TCP' s fast recovery algorithm, *Standards Track, PP*, pp. 1–16 (2012).
[3] Ha, S. et al.: CUBIC: a new TCP-friendly high-speed TCP variant, *ACM SIGOPS operating systems review*, Vol. 42, No. 5, pp. 64–74 (2008).

[4] Endo, T. et al.: Towards automatic generation of congestion control algorithms by coevolving the environment, *The Fourth Workshop on Open-Ended Evolution* (2021).
[5] Brakmo, L. S., O'Malley, S. W. and Peterson, L. L.: TCP Vegas: New techniques for congestion detection and avoidance, *Proceedings of the conference on Communications architectures, protocols and applications*, pp. 24–35 (1994).
[6] Winstein, K. and Balakrishnan, H.: TCP Ex Machina: Computer-Generated Congestion Control, *SIGCOMM Comput. Commun. Rev.*, Vol. 43, No. 4, p. 123–134 (online), DOI: 10.1145/2534169.2486020 (2013).
[7] Wang, R. et al.: Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions (2019).
[8] O'Neill, M. and Ryan, C.: Grammatical evolution, *IEEE Transactions on Evolutionary Computation*, Vol. 5, No. 4, pp. 349–358 (2001).
[9] Lourenço, N. et al.: Unveiling the properties of structured grammatical evolution, *Genetic Programming and Evolvable Machines*, Vol. 17, No. 3, pp. 251–289 (2016).
[10] Rothlauf, F. and Oetzel, M.: On the locality of grammatical evolution, *European conference on genetic programming*, Springer, pp. 320–330 (2006).
[11] Byrne, J., MichaelO' Neill, McDermott, J., Brabazon, A.： An analysis of the behaviour of mutation in grammatical evolution, *European Conference on Genetic Programming*, Springer, pp. 14–25 (2010).
[12] Gawłowicz, P. and Zubow, A.: ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research, *ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)* (2019).
[13] Carneiro, G.: NS-3: Network simulator 3, *UTM Lab Meeting April*, Vol. 20, pp. 4–5 (2010).
[14] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W.: Openai gym, *arXiv preprint arXiv:1606.01540* (2016).

付　　録

付録 A　輻輳制御アルゴリズム生成に用いた BNF

(次ページに掲載)

プログラム 6: 定義した BNF

```
<class> ::= class CongestionControlAlgorithm(object):{:
                def __init__(self):{:
                    super(CongestionControlAlgorithm).__init__():}
            :}{:
                def set_spaces(self, obs, act):{:
                    self.obsSpace = obs{::}
                    self.actSpace = act
                :}
            :}{:
                def get_action(self, obs, reward, done, info):{:
                    new_cWnd = obs[5]{::}
                    new_ssThresh = obs[4]{::}
                    state = obs[12]{::}
                    <algorithm>{::}
                    new_ssThresh = new_ssThresh if new_ssThresh < 10000 else 99999{::}
                    new_ssThresh = new_ssThresh if new_ssThresh > 1 else 1{::}
                    new_cWnd = new_cWnd if new_cWnd < 100000 else 99999{::}
                    new_cWnd = new_cWnd if new_cWnd > 1 else 1{::}
                    return [int(new_ssThresh), int(new_cWnd), obs[2]]
                :}
            :}
            agent = CongestionControlAlgorithm

<algorithm> ::= if state == 0:{:
                    <code1>
                :}{::}
                elif state == 1:{:
                    <code2>
                :}
                elif state == 3:{:
                    <code2>
                :}
                elif state == 4:{:
                    <code2>
                :}
                else:{:
                    <code2>
                :}

<code1> ::= new_cWnd = <update>
        |   if <condition>:{:
                new_cWnd = <update>
            :}
            else:{:
                new_cWnd = <update>
            :}

<code2> ::= new_cWnd = <update>{::}new_ssThresh = <update>
        |   if <condition>:{:
                new_cWnd = <update>{::}
                new_ssThresh = <update>
            :}
            else:{:
                new_cWnd = <update>{::}
                new_ssThresh = <update>
            :}

<condition> ::= obs[<obs_index>] <comp_op> obs[<obs_index>]
<update> ::= <update> <arith_op> <update>
         | obs[<obs_index>]
         | <num>
<obs_index> ::= 4|5|6|7|8|9|10
<comp_op> ::= <|>|<=|>=|==|!=
<arith_op> ::= +|-|*|/|%
<num> ::= 1|2|3|4|5|6|7|8|9
```