卒 業 研 究 報 告 書

題　目　　ソフトウェアリポジトリにおける版分岐に関する考察

指導教員　　水野　修　教授

京都工芸繊維大学 工芸科学部 情報工学課程

学生番号　　13122014

氏　　名　　北脇　知春

平成29年2月14日提出

ソフトウェアリポジトリにおける版分岐に関する考察

平成 29 年 2 月 14 日　　13122014　北脇　知春

概　要

近年ソフトウェア工学分野で git 等のバーション管理システムを用いたソースコード履歴に注目した様々な研究がなされている．しかしながら，枝分かれを持つバージョン管理システムの枝分かれに注目した研究は盛んではない．そこで，本研究ではバーション管理システムの版分岐の解析を行い，その特徴を調査した．

本研究では以下の調査を行った．調査は今日一般的な分散型バージョン管理システムである git を対象とした．1000 のプロジェクトから repository を収集し，（１）版分岐の研究の上で問題となる点，（２）git の版分岐によるグラフの特徴，（３）git の版分岐の際の各分岐における編集量の差異，を調査した．

結果，以下のことが分かった．（１）git にはタイムスタンプと親子関係の矛盾等の分岐解析上の問題があるが，解決可能である．（２）git の repository は様々な規模・形態で運用されているが，多くは (最大の規模，最も複雑なものと比較すると) 小規模で単純な運用となっている．（３）git の枝分かれには主従関係を示唆する経路による編集量の差異が存在する．

これらのことから，バージョン管理システムの版分岐の分析は可能かつ有用であると考えた．

目　次

1. 緒言

近年ソフトウェア工学分野でgit等のバーション管理システムを用いたソースコード履歴に注目した様々な研究がなされている．しかしながら，枝分かれを持つバージョン管理システムの枝分かれに注目した研究は盛んではない．そこで，本研究ではバーション管理システムの版分岐の解析を行い，その特徴を調査した．

版分岐を対象とした研究は少ない．特に分岐そのもののに注目した先行研究は調べた限りでは存在しなかった．そのため，本研究はそもそもバージョン管理システムの版分岐分析が可能であるか，有用であるかの分析を行い，この分野の研究が今後継続すべき価値のあるものかを調べるものである．

第２章で研究背景と本研究で研究対象としたgitの概要説明を行う．第３章で研究設問を明らかにし，研究目的を定める．第４章で本研究で用いた２つの実験の実験手法を説明する．第５章で得られた実験結果とその分析結果を示す．第６章で結果および分析の考察を行う．第７章で以上の結果をまとめる．

2. 研究背景

2.1 既存研究

現在，多くの開発現場で git 等のバーション管理システムを使用した版管理を用いた開発が主流となっている．これにより，ソフトウェア開発において過去のバージョンの収集が可能となっている．

そのため，ソフトウェア工学分野でバーション管理システムを用いたソースコード履歴に注目した様々な研究がなされている．例えば，各種メトリクスを用いてどのモジュールに不具合が多いのかを予測する不具合予測等がある．しかしながら，枝分かれを持つバージョン管理システムの枝分かれに注目した研究は盛んではない．

2.2 git

分散型バージョン管理システムの代表例として git[1] が挙げられる．近年，多くのプロジェクトのバージョン管理が git を用いて行われている．本研究では git を対象に調査を行った．

以下，本研究の内容の理解に必要な git の説明を示す．git は Linux kernel のバージョン管理を目的として Linus Torvalds によって開発された分散型バージョン管理システムであり，大規模プロジェクトのソースコードも管理できるように設計されている．各開発者は最新のスナップショットだけでなく更新履歴をすべて含んだプロジェクトの管理単位である repository のミラーをローカルディレクトリ上に持ち，必要に応じて相互に更新し合うシステムとなっている（分散型）．また，git はソースコードの差分を管理していた初期のバージョン管理システムと異なり，各状態でのスナップショットの集合で repository を構成する．

2.2.1 commit

git は個々のバージョンを commit と呼ばれる単位で管理する．commit はその親の commit，編集者情報，タイムスタンプ，ソースコードのスナップショット等の情報を持つ．

2.2.2 branch

git はバージョン管理の枝分かれをサポートする branch 機能を有する．git で管理されるプロジェクトは，ある commit から幾つもの分岐で様々な機能を平行して追加し（あるいは機能修正し），最終的にそれらを marge という操作で統合して新たな commit を生成することで開発される．後述する内部構造の関係から branch は過去の名前を保持せず，現在の枝分かれの葉の名前情報しか持たない．

2.2.3 git の内部構造

git の内部構造は SHA-1 hash で管理された commit，tree，blob の集合である．[2] tree と blob はある commit でのソースコードのディレクトリ構造とファイルの内容のスナップショットを格納する機構である．（完全に同じファイルは同じ SHA-1 hash が割り当てられるため 1 つ分しか保存されない．）各 commit は固有の SHA-1 hash を持ち，SHA-1 hash で親 commit を指すことで重複する辺を持たない有向非巡回グラフを構成する．(なお，内部構造的には子から親への向きで辺を持つが，一般的には親から子への向きで辺を持つとして議論した方がわかりやすい．そのため，以下表面的には親から子への向きで辺を持つとして説明を行う．辺を逆向きにしても重複する辺を持たない有向非巡回グラフであることには変わりはない．) branch は commit グラフの子を持たない commit（グラフの葉ノードに相当する）の SHA-1 hash とその名前のリストを格納したファイルで管理される．つまり git の branch 機能は現在の枝分かれの名前のみを保持する．これらの git のデータは，プロジェクトディレクトリ内の不可視ディレクトリ.git に格納される．

3. 研究目的

先に述べたようにバージョン管理システムの枝分かれに注目した研究は盛んではない．そこで，バージョン管理システムの枝分かれを分析することでより高精度のバグ予測やその他の知見が得られると予想し，本研究ではその前段階としてプロジェクトにおけるバージョン管理の枝分かれがどのように運用されているかを調査し，バージョン管理システムの版分岐分析が有用であるかの分析を行う．

当初 git が過去の枝分かれ情報を保持しないことに注目して「過去の枝分かれラベルの推定」や「枝分かれの主流支流の推定」等をテーマに研究することを考えたが，先行研究が少なかったため実用的な精度での実現が困難であると考えた．そこで，これらの前段階として git の枝分かれの分析を行う．

git の枝分かれ分析のために，3 つの研究設問を設定した．

RQ1 git の仕様に枝分かれ解析の上で問題となり得る特徴はあるか？

RQ2 git repository のグラフにはどのような特徴があるか？

RQ3 git repository において，枝分かれの枝の間に主従関係は得られるか？

3.1 RQ1:git の仕様に枝分かれ解析の上で問題となり得る特徴はあるか？

git の枝分かれの解析を行う上で問題となり得る，一般的ではないが仕様上可能な機能等などが存在するかを調べる．また，存在するのならばそれらの量を調べる．

3.2 RQ2:git repository のグラフにはどのような特徴があるか？

枝分かれに関する各種特徴量を算出し，それらの分布や，お互いの相関を観察することで，git repository の一般的な運用を解明する．

3.3 RQ3:git repository において，枝分かれの枝の間に主従関係は得られるか？

当初の目的であった「枝分かれの主流支流の推定」の前段階として，枝分かれの経路間に何らかの特徴があるかを調べる．今回は，「commit 数が多い経路の方が各種編集量が多くなる」との仮説を立て，それを検証した．一般に git の枝分かれはリリース用の安定したものと不安定なもので主従関係を持つような運用をされている場合が多く，これらの運用の結果が数値的に観測可能かを検証する．

4. 実験概要

本研究では 3 つの RQ に答えるため，2 つの実験を行った一つは git repository のグラフの概形を調べる実験，もう一つは枝分かれの特徴を調べる実験である．実験 1 とその実装過程のバグ修正記録で RQ1・RQ2 を調べ，実験 2 で RQ3 を調べる．

4.1 実験対象

2017 年 1 月 18 日時点で GitHub[3] 内のスターの多い 1000 個の repository を実験対象とした．GitHub は最も一般的な Git リポジトリホスティングサービスで，多くのプロジェクトが GitHub 上に repository を置いている．また，各 repository は 2017 年 1 月 27 日時点の状態を使用した．

4.2 実験 1

1000 個の repository について以下の 12 のメトリクスを算出する．

commit num commit の総数

edge num commit 間の親子関係の総数

current width 現在の repository の幅

max width 過去最大だった repository の幅

width sum repository の幅の総和

branch num 枝分かれの数

merge num 統合の数

merged branch 統合された枝分かれの数

max branch 1 つの commit から枝分かれした commit の最大数

max merge 1 つの commit に統合された commit の最大数

orphan commit 孤児 commit(親を持たない commit) の数

size .git ディレクトリのサイズ (KB)

そしてこれらの最小値，最大値，度数分布，相互の相関の有無を分析し，RQ1・RQ2 の答えを得る．

4.2.1 commit num

グラフの頂点の数に相当する．repository の規模を表す．

4.2.2 edge num

出次数が 0 の頂点の個数に相当する．グラフの辺の数に相当する．repository のグラフの密度を表す．

4.2.3 current width

repository の幅とはある時点での branch の数である．子を持たない commit の数と同値である．現在の repository で平行して行われている作業の数を示す．

4.2.4 max width

過去最も開発が盛んだった時期に平行して行われていた作業の数を示す．

4.2.5 width sum

現在までの平行して行われた作業の量を表す．

4.2.6 branch num

各 commit の子の組み合わせの総和．グラフの各頂点の出次数とそれから 1 引いたものの積を 2 で割ったものの和に相当する．

4.2.7 merge num

各 commit の親の組み合わせの総和．グラフの各頂点の入次数とそれから 1 引いたものの積を 2 で割ったものの和に相当する．

4.2.8 merged branch

最終的に merge された枝分かれの数．

4.2.9 max branch

グラフの各頂点の出次数の最大値に相当する．

4.2.10 max merge

グラフの各頂点の入次数の最大値に相当する．

4.2.11 orphan commit

孤児 commit(親を持たない commit) の総数．一般に，最も最初の commit(initial commit) でないが，親を持たない commit を孤児 commit と呼ぶ場合が多いが，今回は initial commit も孤児 commit としてカウントしている．入次数が 0 の頂点の個数に相当する．

4.2.12 size

.git ディレクトリのサイズ (KibiByte)．repository のディレクトリサイズではどの branch に checkout しているかによって大きさが変わってしまうため，それに影響されない repository の内容のみを含む.git ディレクトリのみの容量を測る．

4.3 実験 2

実験 1 で merged commit の数が 1000 以上だったものについて，その枝分かれから合流までの 2 経路それぞれの差分の総和を求める．差分として以下のメトリクスを求めた．ただし，差分の総和が一致もしくは近い値となってしまうことを防ぐため，合流する際の差分は除いて求めた．

add file num　追加されたファイル数

delete file num 削除されたファイル数

add loc　追加された行数

delete loc　削除された行数

add todo　追加されたソースコード中の「todo」の文字列の数

delete todo　削除されたソースコード中の「todo」の文字列の数

issue num　commit message 中の#の数

message length　commit message の長さ

fix message　commit message 中の「fix」文字列の数

なお，これらは差分の和であり，例えば経路中に一度追加されてその後削除された行が存在した場合もそれぞれ追加 1 行と削除 1 行として行数を数え上げられる．

これらのメトリクスを経路間の commit 数で割り，1commit あたりの平均のメトリクスを算出する．この値を commit 数の多い経路のものと commit 数の少ない経路のもので差を取る．

$$(long\ route\ metrics/long\ route\ commit\ num)-(short\ route\ metrics/short\ route\ commit\ num) \tag{4.1}$$

仮説「commit 数が多い経路の方が各種編集量が多くなる」が正しければこの値は正を取るはずである．各 repository について平均が 0 よりも優位に大きくなるかを確かめる．

4.3.1　add/delete file num

大きな単位での機能追加・削除の量を調べる．

4.3.2　add/delete loc

LOC(Line of Code) は一般的にソフトウェアの規模を示すメトリクスである．機能の追加削除や修正等の編集量を調べる．

4.3.3　add/delete todo

”todo”の文字列は機能追加等を後回しにする際の表現で，様々なプログラミング言語で使用される．大文字小文字の区別をなくして，コメントアウト等の判定をせず，追加削除された行中の todo の文字列を数えた．

4.3.4 issue num

GitHub において，issue(不具合修正) の追跡に#が用いられており，commit message 内の#の数を調べる．

4.3.5 message length

commit message に機能の追加削除修正の概要が記されるため，commit message の文字数を調べることで機能追加削除修正の量を推測できると考えた．

4.3.6 fix message

上記の通り，commit message には編集内容の概要が記されるため，”fix”の文字列の数で機能修正の量を調べる．大文字小文字の区別なく，commit message 中の”fix”の文字列の数を数えた．

4.4 アルゴリズム

二つの実験で使用するアルゴリズムは共通である．以下，アルゴリズムの概要を示す．

1. 各 branch を始点として深さ優先探をし，全ての commit を収集する．
2. commit の集合をタイムスタンプ順にソートする．
3. 全ての commit について親 commit との差分を計算する．(実験 2 のみ)
4. コミットを親から子の順に探索しながら各種メトリクスを算出する．

4.4.1 全ての commit の探索

全ての commit を探索するため，全ての branch を探索してスタックに格納し，そのスタックを用いて深さ優先探索で全ての commit を収集する．

4.4.2 ソート

Commit を古いものから順に探索するために，全ての commit をソートする．git の commit には author date と committer date の二つのタイムスタンプがあるが，今回

は commit した時刻を強く反映する committer date を使用した．実際には後述するタイムスタンプと親子関係の矛盾問題からトポロジカルソートを使用した．

4.4.3 差分算出

実験 2 で必要な commit 間の差分はあらかじめ先に演算しておくことで計算量を減らした．

4.4.4 メトリクスの算出

ソート済みの commit 列に対して最も古い commit から順に commit を探索して各メトリクスを算出する．実験 1 の merged branch メトリクスと実験 2 の差分和については，素集合データ構造を用いて算出または不必要な探索の枝刈りを行う．素集合データ構造（disjoint-set data structure, 別名 Union-Find データ構造）は集合を素集合に分割して保持し，「与えられた二つの要素が属する集合を一つに統合する (Union)」と「与えられた二つの要素が同じ集合に属するかの判定 (Find)」の二種類のクエリを実行可能なデータ構造である．これは高速に実行するアルゴリズムが知られている．実験 2 の差分の総和は二つの経路それぞれについて，親方向に探索して同じ commit に到達したらそこまでの差分の和を出力する各経路で複数の経路が存在した場合は，なるべく commit 数が少なく，同じ commit 数ならば各 commit のタイムスタンプが新しくなるものを使用する．なお，実装は二つの探索を同時に行い，探索中に訪れた commit までの差分を格納する配列を用意して，それを適宜更新することでメモ化して無駄な演算を防いだ．更新の際にその commit までの経路が複数存在するならば，経路間の commit 数が最も少ないなかで子のタイムスタンプが最も新しいもので更新する．

5. 実験結果

5.1 実装中のバグ修正からの知見

git の仕様が当初想定していた（一般に想定されている使用方法）よりも広い操作を許容するものであり，それらを使用した痕跡のある repository が散見された．実装中のバグ修正から分かったことは以下の通りである．

5.1.1 タイムスタンプと親子関係の矛盾

commit のタイムスタンプは親 commit よりも子 commit の方が新しいものになるはずであるが，親 commit よりも子 commit のタイムスタンプが古い親子関係が存在した．一度 commit したものを，後から内容の変更を行う操作（git の仕様上可能である）をした結果であると考えられ，その結果 commit のタイムスタンプに矛盾が生じる結果となった．

当初タイムスタンプ順に探索をすればグラフを辺の向きに探索が可能であると考えていた（タイムスタンプでのソートがトポロジカルソートと同値であると仮定していた）が，タイムスタンプ順では矛盾が生じたため，グラフを一方方向に探索しつつなるべくタイムスタンプの順に探索するためにトポロジカルソートを用いた．

5.1.2 トポロジカルソート

トポロジカルソートとは，グラフにおいて任意の 2 頂点間に経路が存在するならば，経路方向にが順序関係が決定されるように並べるソートのことである．有向非巡回グラフは必ずトポロジカルソートが可能でる．(逆に閉路を持つならばトポロジカルソートは不可能である．) トポロジカルソートは解が一意に定まらないため，今回はトポロジカルソートを満たした上で，commit のタイムスタンプがなるべくソートされるようにした．

5.1.3 孤児 commit

全ての commit は最初の commit(initial commit, first commit) の子孫であると一般に思われているが，git の仕様上 initial commit 以外にも親のいない commit を生成する

ことが可能である．これは孤児 commit と呼ばれるもので，いくつかの repository にみられた．

5.1.4　3 つ以上の親を持つ commit

一般に merge 操作は 2 つの branch を一つにすることだと思われているが，git の仕様上 3 つ以上の commit からの merge 操作が可能で，それを使用した 3 つ以上の親を持つ commit が存在した．

5.2　実験 1

前述の通り，実験 1 は 1000 個の repository に対して行った．

5.2.1　各メトリクスの最小中央最大値

得られた各メトリクスの最小中央最大値を表 5.1 に示す

表 5.1　repository の各メトリクスの最小中央最大値

	min	median	max
commit num	5	1110.0	649019
edge num	4	1339.0	700492
current width	1	3.0	917
max width	1	14.0	10796
width sum	5	5207.0	1002525626
branch num	0	246.5	1282634
merge num	0	165.5	206744
merged branch	0	165.5	206740
max branch	1	6.0	404
max merge	1	2.0	66
orphan commit	1	1.0	159
size	212	6488.0	2360136

ここで特筆すべきは，max branch,max merge,orphan commit の最大値である．それぞれ 404,66,159 と一般的な運用では想定しにくい値が出ており，このような特異な運用をされている repository の存在を確認することが出来た．

5.2.2 度数分布

3 つ以上の commit からの merge と孤児 commit を持つ repository の存在確率を調べるために，max merge と orphan commit の度数分布の解析を行う．max merge，orphan commit の度数分布を表 5.2 に示す．

ほとんどの repository が親 commit の最大個数が 2 以下の commit のみで構成されていることが分かる．特に，66 の親を持つ commit が希なケースであることが分かる．

孤児 commit の数は，6 割の repository で一個，ほとんどの repository で 2,3 個以内であることが分かる．特に，159 個の孤児 commit を持つ repository が希なケースであることが分かる．

5.2.3 孤児 commit 間の関連性の調査

孤児 commit 同士がどれくらい繋がっているかを調べるために，孤児 commit が 2 以上の repository に対して，merge num と merged branch の差を orphan commit-1 で割った値

$$(merge\ num - merged\ branch)/(orphan\ commit - 1) \tag{5.1}$$

を算出する．この値は branch していないのに merge された枝分かれの割合である．

孤児 commit が 2 以上の repository は 376 個であった．表 5.3 に結果の度数分布を示す．

結果，5 割の repository で孤児 commit の子孫間での merge は見られなかった．

5.2.4 repository size

GitHub では本来 1 個の repository を 1GB 未満にするように推奨・要請 (recommend) されている [4] が，今回表 5.4 に示す 5 つの repository が 1GB 以上であった．これらは GitHub 内部で扱われている容量とは異なる場合があるが (概ね，GitHub 内部では今回算出した値よりも少ない容量と見積もられているようである)，明らかに 1GB を超える repository が存在することは事実である．

表 5.2　実験 1 のメトリクスの度数分布表

(a) max merge の度数分布表

max merge	Frequency
1	14
2	963
3	12
4	4
5	1
6	1
8	1
9	1
11	1
12	1
⋮	
66	1

(b) orphan commit の度数分布表

orphan commit	Frequency
1	624
2	284
3	53
4	15
5	5
6	4
7	3
8	2
9	1
10	2
13	3
19	1
21	1
29	1
⋮	
159	1

5.2.5　外れ値除去

各メトリクス間の相関関係を調べるために，外れ値除去を行う．

まず，各メトリクスの確率分布を分析する．シャピロ–ウィルク検定により，全てのメトリクスは正規分布に従うものではないことが分かった．そのため，正規性を前提とした外れ値検定手法を用いることが出来ない．

そこで，ノンパラメトリックな手法である，四分位点を用いた外れ値同定手法を用いて外れ値除去を行う．これは，第 1 四分位点から第 3 四分位点の閉区間を正負方向それぞれに四分位範囲の 1.5 倍拡張した閉区間

$$[Q_{\frac{1}{4}} - 1.5IQR, Q_{\frac{3}{4}} + 1.5IQR] \; where \; IQR = Q_{\frac{3}{4}} - Q_{\frac{1}{4}}, Q_{\frac{n}{4}} = \text{第 } n \text{ 四分位点} \qquad (5.2)$$

表 5.3　branch していない merge された枝分かれの割合の度数分布表

un branchcd merge rate	Frequency
(-0.1,0]	206
(0,0.1].	0
(0.1,0.2]	1
(0.2,0.3]	0
(0.3,0.4]	1
(0.4,0.5]	29
(0.5,0.6]	0
(0.6,0.7]	5
(0.7,0.8]	2
(0.8,0.9]	3
(0.9,1]	128
⋮	
(5.5,5.6]	1

外の観測値を外れ値とする手法である．今回の実験で得られたデータでの上記閉区間を表 5.5 に示す．全てのメトリクスがこの閉区間内にあるデータのみを抽出すると，677 個の repositry が該当した．

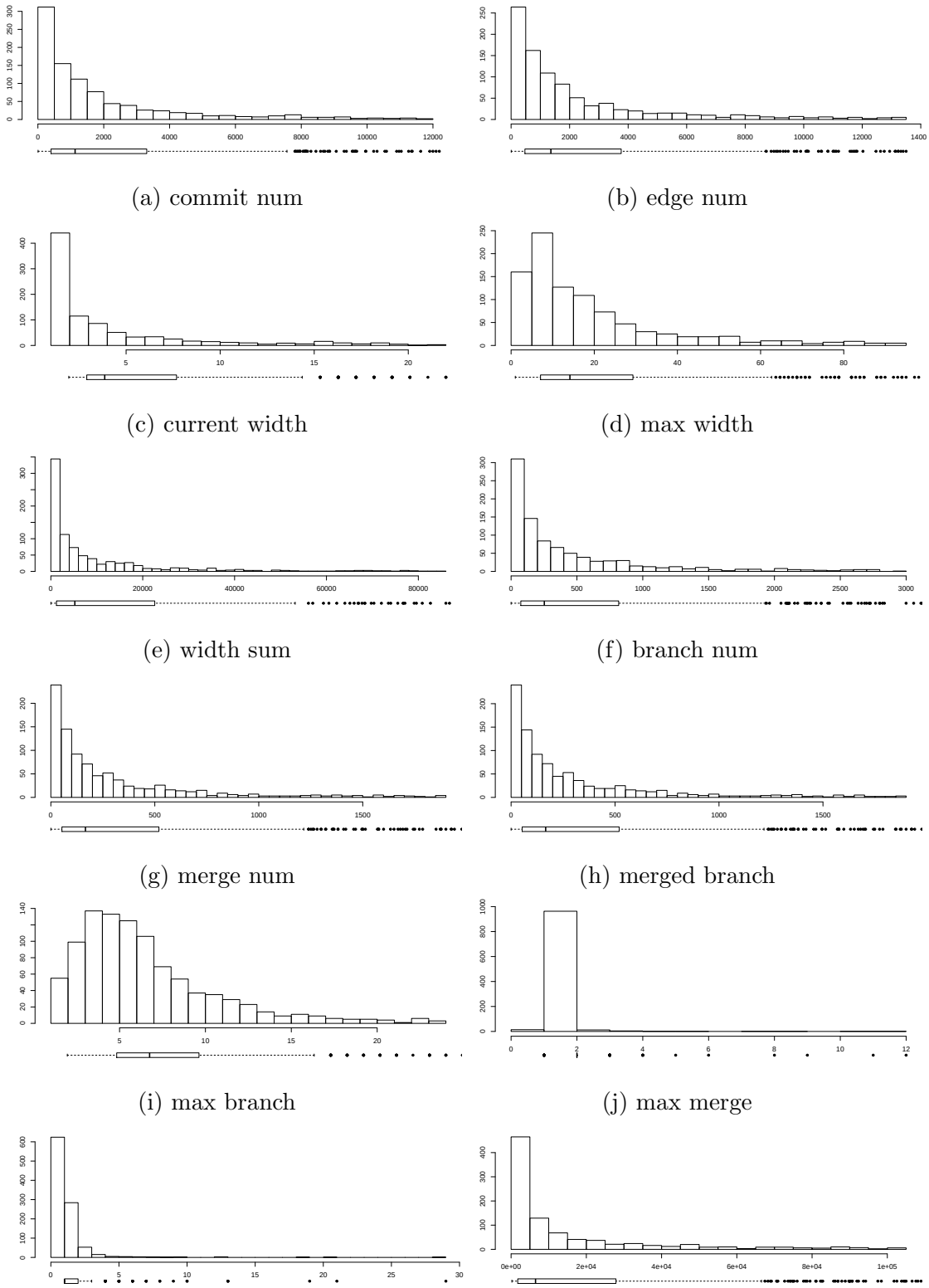
また，図 5.1 に各メトリクスのヒストグラムと箱ひげ図を示す．各メトリクスの値の範囲が広いため，max merge と orphan commit については明らかに外れ値である最大値を除いた区間を，その他のメトリクスについては強い外れ値でない閉区間

$$[Q_{\frac{1}{4}} - 3IQR, Q_{\frac{3}{4}} + 3IQR] \; where \; IQR = Q_{\frac{3}{4}} - Q_{\frac{1}{4}}, Q_{\frac{n}{4}} = \text{第 } n \text{ 四分位点} \tag{5.3}$$

を示す．

5.2.6　メトリクス間の相関分析

いずれのメトリクスにも外れ値を含まない 677 個の repositry についてメトリクス間の相関を分析する．ただし，メトリクス max merge については 2 以外の観測値が全て外れ値となり，分散が 0 になったために，除外する．図 5.2 に散布図行列を示す．



(a) commit num

(b) edge num

(c) current width

(d) max width

(e) width sum

(f) branch num

(g) merge num

(h) merged branch

(i) max branch

(j) max merge

(k) orphan commit

(l) size

図 5.1　実験 1 の各メトリクスのヒストグラムと箱ひげ図

表 5.4　1GB 以上の容量を持つ repository

repository	size(GB)
android/platform_frameworks_base	2.25
torvalds/linux	1.81
odoo/odoo.	1.74
arduino/Arduino	1.23
google/WebFundamentals	1.11

表 5.5　実験 1 の各メトリクスの外れ値でない閉区間

commit num	[-3883.125,7517.875]
edge num	[-4390.75,8559.25]
current width	[-5.5,14.5]
max width	[-26,62]
width sum	[-30733.88,54485.12]
branch num	[-1017,1887]
merge num	[-643,1213]
merged branch	[-643,1213]
max branch	[-3.5,16.5]
max merge	[2,2]
orphan commit	[-0.5,3.5]
size	[-37131.5,66600.5]

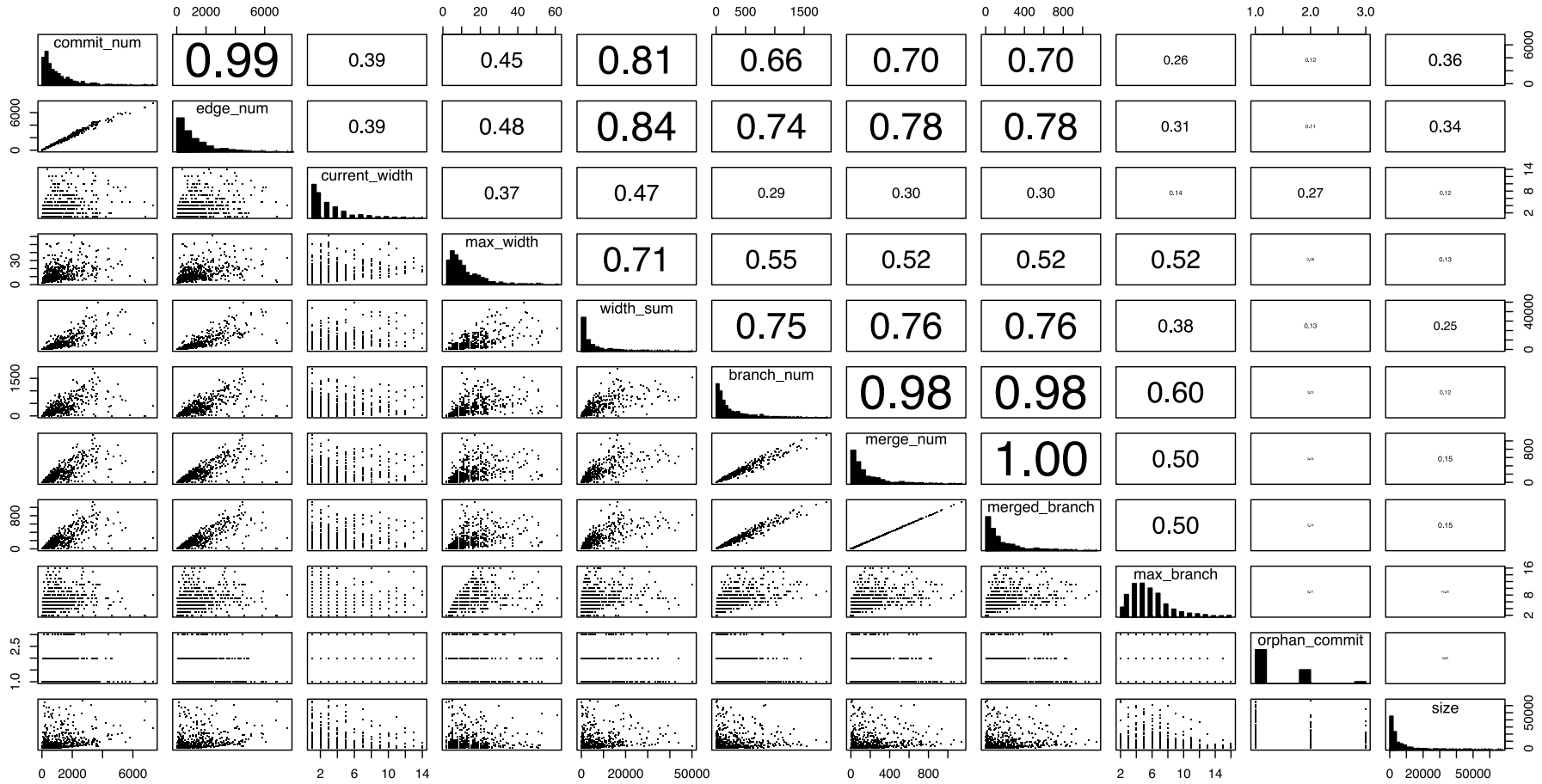


図 5.2　実験 1 の散布図行列

commit num と edge num の間，branch num と merge num と merged branch の間には非常に強い相関があった．commit num ， edge num , width sum ，branch num ，merge num ， merged branch の間，max width と with sum の間にも強い相関があった．逆に，current width ， max branch ， orphan commit ， size は他のメトリクスと強い相関を持たないか相関がなかった．

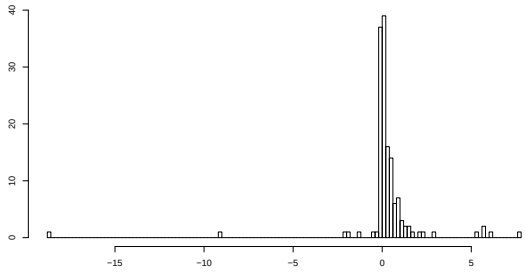
5.3 実験 2

実験 1 で merged commit の数が 1000 以上だった 144 個を対象に行った．ただし，torvalds/linux と android/platform_frameworks_base については，一定時間 (12 時間) 内に演算が終了しなかったため，除外した．（主に commit 間の diff の数が多すぎてディスク IO がボトルネックとなり，演算に時間がかかったようである．）結果 142repository についてデータを得られた．表 5.6 に各メトリクスの負，有意差なし，正だった repository の個数と最小値中央値最大値を示す．中央値の算出には有意差のなかったデータは含まない．図 5.3 にメトリクス毎のヒストグラムを示す．なお，有意差がなかったデータは 0 としてプロットした．

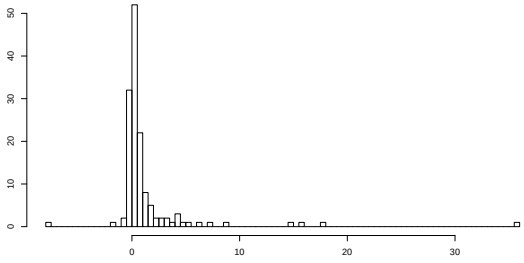
表 5.6 実験 2 の各メトリクスの有意差の有無と正負，最小中央最大値

	minus	no difference	plus	min	median	max
add file num	8	36	98	-18.6483	. 0.2745	7.6264
delete file num	6	30	106	-7.5821	0.4358	35.7717
add loc	7	32	103	-7118.3593	77.1949	3940.0944
delete loc	4	33	105	-2902.3379	115.8170	9621.0062
add todo	4	35	103	-7.6612	0.0854	3.4219
delete todo	3	36	103	-20.2049	0.1119	6.1141
issue num	0	2	140	0.0024	0.1629	0.4012
message length	0	2	140	5.9891	26.36619	138.3044
fix message	0	0	142	. 0.0036	0.0467	0.2720

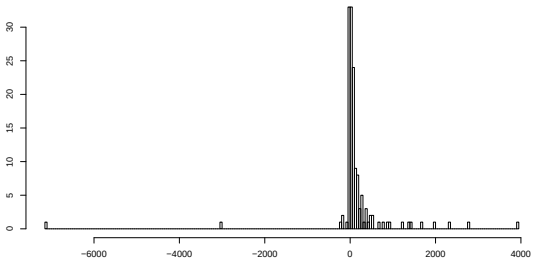
結果，差の大小はあるが 9 つのメトリクス全てで，commit 数の多い経路の方が commit 辺りのメトリクス量が多くなることが分かった．



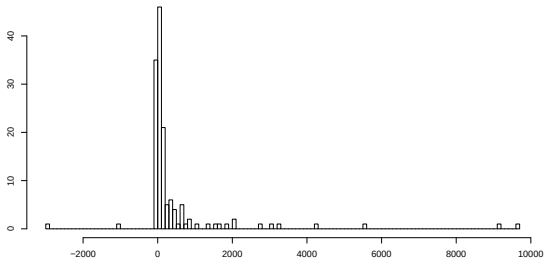
(a) add file num



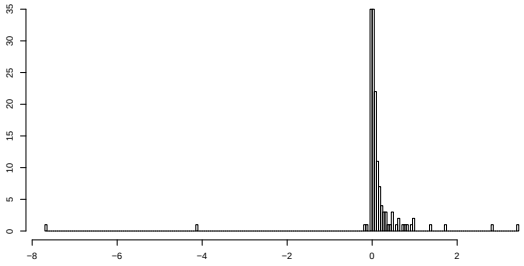
(b) delete file num



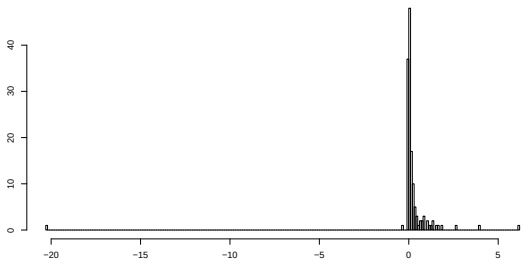
(c) add loc



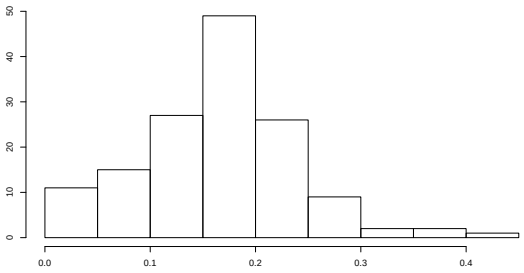
(d) delete loc



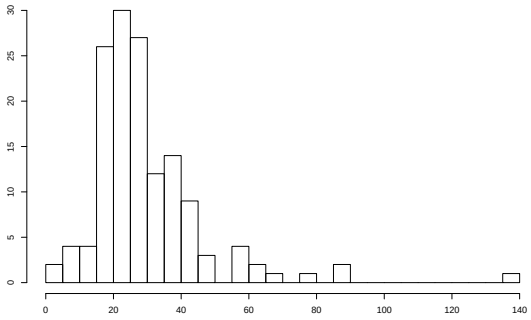
(e) add todo



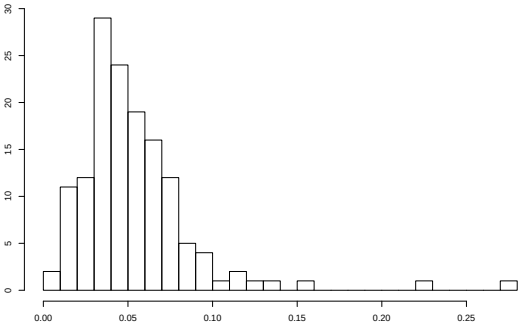
(f) delete todo



(g) issue num



(h) message length



(i) fix message

図 5.3　実験 2 の各メトリクスのヒストグラム

6. 考察

6.1 RQ1:git の仕様に枝分かれ解析の上で問題となり得る特徴はあるか？

枝分かれ解析の上で障壁となる特徴は存在する．事実，実装時 git の仕様が汎用性の高いものであるために存在する，タイムスタンプの矛盾や孤児 commit，3 つ以上の親を持つ commit，大きすぎる repository などが問題となった．

タイムスタンプの矛盾については当該プロジェクト開発者のミスであると予想される．タイムスタンプの矛盾は前述のトポロジカルソートの利用で解決することが出来る．

孤児 commit についてはドキュメント等の管理を独立して行うために利用されており 2,3 個の孤児 commit を持つ repository も数多くあった．また，複数の孤児 commit を持つ repository のうち半分以上が孤児 commit の子孫間での merge を行っていないことから意図的に独立させていることがうかがえる．

3 つ以上の親を持つ commit については多すぎた branch を一度に集約するために苦肉の策として行われているようであったが，必要となる場面は少なく，ほとんどの repository では見られなかった．

大きな repository はプロジェクトの開発規模上仕方がないものである．GitHub の 1GB 未満の推奨・要請は本来の git の趣旨に合わない大きなファイル（大量大容量の動画像音声やバイナリファイルなど）の管理を git 上でしている場合についての警告が主であるようである．大きいファイルを扱う場合には Git Large File Storage(LFS)[5] が推奨されている．

6.2 RQ2:git repository のグラフにはどのような特徴があるか？

git repository が多様な大きさ・運用方法で使用されていることが分かった．しかし，それぞれのメトリクスの中央値はどれもそれぞれの最大値よりも遙かに小さい値を取っており，多くの repository が今回調査した中で最大のものに比べると遙かに小規模かつ単純に運営されていることが分かった．

以下各メトリクスについて考察する．

6.2.1 commit num

数 commit しかない repository から数十万 commit の非常に大きな repository までがあった．git が様々な規模で運用されていることが分かる．

6.2.2 edge num

commit num と非常に強い相関があった．Commit の総数と親子関係の数の比はほぼ一定の比率になるようである．

6.2.3 current width

枝分かれを使用していないものから，900 以上の葉を持つ repository までがあった．しかし，中央値は 3 と小さく，多くの repository が現在は高々数個の branch で運用されていることが分かった．

6.2.4 max width

枝分かれを使用していないものから，10000 以上の最大の幅を持つ repository までがあった．中央値は 14 と current width と同様にその値域に比べて小さい値に収まっている．このことから，repository は一時的に非常に太くなりうることが分かる．意外なことに current width との相関は少なく，現在の開発の並列性は過去の開発の並列性の高さによらないようである．

6.2.5 width sum

commit num の最小値と同じ最小値 5 から，1000000000 以上までと，広い分布を示す．current width よりも max width と強く相関し，開発途中で最も盛んだった時期の影響を受けた数値になるようである．また，幅を足し合わせる回数である commit num と，それに非常に強く相関している edge num とも強い相関を持っていた．

6.2.6 branch num

他のメトリクスと同様に大きい最大値とそれに比べて小さな中央値を持つ．

6.2.7 merge num・merged branch

この二つのメトリクスは initial commit 以外の孤児 commit が存在しなければ同値になるものであるが，initial commit 以外の孤児 commit があっても半数以上が同値になり，残りも近い値となった．そのため非常に強い相関関係を持つ．また，branch num とも強い相関関係を持ち，branch と merge の比率がほぼ一定であることが分かった．

6.2.8 max branch

404 と予想外の最大値を観測した．1 つの commit から 404 も派生して管理可能なのかは疑問ではあるが，そのような運用を行った痕跡があった．なお，中央値は 6 と一般的な数値が出た．このメトリクスは max width，branch num，merge num，merged branch と相関を持っており，開発の並列性が高いと，どうしても複雑な枝分かれを強いられるのだろうと予想した．

6.2.9 max merge

予想外の 66 との最大値を観測した．repository の管理としてはあまり良くないと思われる運用である．中央値は 2 と一般的な merge 動作の親の数であった．また，ほとんどがこの中央値であり，3 つ以上の親を持つ merge は極少数であった．

6.2.10 orphan commit

予想外の最大値 159 を観測した．また，前述のように initial commi を含めて 2,3 個の孤児 commit は一般的であるようだ．このメトリクスは他のメトリクスと全く相関がなかった．

6.2.11 size

repositry の規模を示す commit num 等との強い相関を予想したが，意外にも相関係数は小さく，commit 量と実ファイルサイズの関連性はみられなかった．他のメトリクスとの相関はなかった．

6.3 RQ3:git repository において，枝分かれの枝の間に主従関係は得られるか？

「commit 数が多い経路の方が各種編集量が多くなる」という予想は概ね正しかった．しかしながら，逆に編集量が少ないものや，有意差がないものなどの例外もあった．

6.3.1 add/delete file num

ファイルの追加削除は頻度が少なく，多くの分岐で両方ないしどちらかが 0 であった．結果として，各 repository の平均値は有意差がないか，あっても 0 から 1 の範囲のものがほとんどであった．そのため，全 repository を集計した中央値も小さな値となった．しかし，この値は 1commit あたりのファイルの追加削除数とみれば十分に大きく，commit 数が多い経路の方が編集量が多いと言える．が，1/3 程度に有意差がないか負の値を取る repository があったため，repository の運用によっては編集量の多い経路が逆転したり，差がなかったりするようである．（これは loc,todo も同様）

6.3.2 add/delete loc

この実験で最も大きな差を観測できたメトリクスである．ただ，追加よりも削除の loc の方が編集量の差が大きいという結果は予想外であった．

6.3.3 add/delete todo

todo も file と同等に出現確率が低く，大きな値とならなかった．

6.3.4 issue num

出現頻度が低く，大きな値とならなかった．しかしながら，上記 6 つのメトリクスと異なり，有意差のないデータが少なく，ほぼ全てが正の値であるため，編集量の差の存在を示す事が可能である．（これは message length，fix message も同様）

6.3.5 message length

このメトリクスは全ての commit に正の値が存在するため，十分大きな差が得られた．

6.3.6 fix message

issue num と同様に出現頻度が低く，大きな値とならなかった．

6.4 今後の課題

今回の調査で，git repository の枝分岐の解析が可能であり，また枝分岐の複数経路間の特徴に差異があることが示された．よって，当初の目的であった「過去の枝分かれラベルの推定」や「枝分かれの主流支流の推定」などの実現は可能だと思われる．今回の経路間の差異として用いたメトリクスの中では大きな差を出した追加削除 LOC，commit message 長が利用可能であると思われる．

7. 結言

git repository の概形を調べる実験と枝分かれの特徴を調べる実験の二つを行った．

枝分かれ解析の上で，タイムスタンプの矛盾や孤児 commit，3 つ以上の親を持つ commit，大きすぎる repository などが問題となるが解決は可能である事が分かった．

git repository は様々な規模・運用方法で使用されているものがあることが分かった．ただし，多くの repository は小規模で単純な運用で使用されている．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学部門水野修教授に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂きました，本学情報工学専攻森啓太先輩原田禎之先輩，情報工学課程近藤将成君をはじめとする，ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった家族や友人に深く感謝致します．

参考文献

[1] Git, （オンライン）, 入手先〈https://git-scm.com〉（参照 2017-2-7）.

[2] Git - Git オブジェクト,（オンライン）, 入手先〈https://git-scm.com/book/ja/v2/Git の内側-Git オブジェクト〉（参照 2017-2-7）.

[3] GitHub, （オンライン）, 入手先〈https://github.com〉（参照 2017-2-7）.

[4] What is my disk quota? - User Documentation, （オンライン）, 入手先〈https://help.github.com/articles/what-is-my-disk-quota/〉（参照 2017-2-7）.

[5] Git Large File Storage - Git Large File Storage (LFS) replaces large files such as audio samples, videos, datasets, and graphics with text pointers inside Git, while storing the file contents on a remote server like GitHub.com or GitHub Enterprise., （オンライン）, 入手先〈https://git-lfs.github.com〉（参照 2017-2-7）.