修士学位論文

題目

ソフトウェアプロダクト集合に対する派生関係木の構築

指導教員
井上 克郎 教授

報告者
神田 哲也

平成 25 年 2 月 5 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 24 年度 修士学位論文

ソフトウェアプロダクト集合に対する派生関係木の構築

神田 哲也

内容梗概

ソフトウェアプロダクトラインエンジニアリングは，ソフトウェアをプロダクト間で共通するコア資産と，個別の要求に応じて開発される機能部品とに分割して開発・管理を行う開発理論である．またこの考え方を適用して開発された一連のソフトウェアをソフトウェアプロダクトラインと呼ぶ．ソフトウェアプロダクトラインエンジニアリングは，製品の効率的な開発・保守に役立つ．

既存のソフトウェアプロダクトファミリからソフトウェアプロダクトラインを構築することは重要である．このためには，既存のソフトウェア集合を解析・比較しなければならない．すべてのソフトウェアに対し解析を行うことはコストがかかる作業であるため，ソフトウェアプロダクトライン構築に利用するソフトウェアを選択する必要がある．

ソフトウェアの選択には，どのソフトウェアがどのソフトウェアから作られ，また開発がどこで分岐したのかといった派生関係が有用である．派生関係は，ソフトウェアの開発履歴から得ることができるが，類似ソフトウェア間での開発管理が行われていない状況や，開発が複数の組織にまたがって分岐した状況などでは，開発履歴を得ることができない．

本研究では，既存のソフトウェアプロダクト集合からソフトウェアプロダクトラインを構築するに当たり，機能比較を行う対象となるソフトウェアの選定作業の助けとなるよう，ソフトウェアの派生関係を模した木，派生関係木を構築する．類似ソフトウェア間での開発管理が行われていない状況に対応するため，派生関係木の構築を，ソフトウェアのソースコードのみを入力とし，その類似度をソフトウェア間の距離とみなした最小全域木を求める問題として定義した．オープンソースソフトウェアに対して行った適用実験では，木の形は約 88%が、派生の方向も約 86%が開発履歴と一致し，開発履歴を近似した派生関係木をソースコードのみから構築できることを確認した．

主な用語

ソフトウェアプロダクトライン
ソフトウェア進化

ソースコード比較
可視化

目 次

1　はじめに

ソフトウェアプロダクトラインエンジニアリング (Software Product Line Engineering, SPLE) は，プラットフォームと大量個別生産を用いてソフトウェアプロダクトファミリを開発するパラダイムである [15]．SPLE では，ソフトウェアをプロダクト間で共通するコア資産と，個別の要求に応じて開発される機能部品とに分割して開発・管理を行う．新たなソフトウェアを開発する際は，コア資産に既存の機能部品や新たに開発した機能部品を組み合わせる．これにより，すでに開発されたソフトウェアの再利用が実現され，ソフトウェアプロダクトファミリとして開発・管理することができる．このようにして開発された一連のソフトウェアをソフトウェアプロダクトラインと呼ぶ．

ただし，ソフトウェプロダクトファミリは，必ずしも開発の最初期からコア資産と機能部品とに分けて設計されるわけではない．プロダクトラインが意識されていない設計のソフトウェアを改変して新しい機能を持ったプロダクトを作る時には，開発者はソースコードをすべてコピーし (fork)，必要とされる機能に応じた修正 (modify) をして顧客の要求を満たすソフトウェアを開発する [17]．この fork-and-modify のことを日本語では派生と呼ぶ．こうして細かな改変が加えられたソフトウェアが次々と作成されていくことになり，類似する多数のソフトウェアの開発が独立に，一貫した管理がないまま進んでいく [3]．つまり，あるソフトウェアプロダクトを基に開発が数多く分岐したことで，それらのプロダクト群が 1 つのプロダクトファミリとして認識されるようになる．

ソフトウェアプロダクトファミリとして派生したソフトウェアに対し，ソフトウェアプロダクトラインの開発手法を導入することは重要である [21]．既存のソフトウェアプロダクトから共通な部分をコア資産として抽出しておけば，機能拡張やバグ修正のコストを削減することができる．また，複数のプロダクトで利用されている機能部品は，それだけ重要であるとも考えられるため，集中的に保守を行ったり，新たなプロダクトを作る際に導入を優先的に検討するなど，良いプロダクトを作るための材料となる．

既存のソフトウェアプロダクトファミリからソフトウェアプロダクトラインを構築するために，機能モデルを抽出することがよく行われている．しかし，機能モデルを抽出してソースコードを共通化するためには，まず今現在どのような機能が実現されており，それが複数プロダクトに共通するものなのか，それとも一部あるいは単独のプロダクトにのみ出現するものなのかを判断しなければならない．これは時間のかかる作業である．また，ソフトウェアプロダクトファミリ内に多数のソフトウェアが含まれている場合，そのすべてについて差分を取り解析を行うことは現実的ではない．

Kruger は，既存のソフトウェアからコア要素と機能部品を抽出するアプローチにおいては，既存のソフトウェアプロダクトを一度にすべて解析する必要はなく，初めにいくらかの

プロダクトを解析した後，残りは必要に応じて順次解析していくことになるだろうと述べている [8]．この考えに従うと，ソフトウェアプロダクトラインの構築はまず，ソフトウェアプロダクトファミリからいくつかのソフトウェアを選び出して行うことになる．

ソフトウェアの選択基準については，多くバグ修正が適用されたもの，多くの機能を持つもの，開発が分岐する直前など，様々なものが考えられる．これらの情報は，ソフトウェアがどのように分岐し，その後進化していったかという開発の履歴から得ることができる．ところが，開発者はすべての開発履歴を覚えているわけではなく [14]，すべての履歴が参照可能であるとも限らない．近年ではバージョン管理システムの普及により，開発の履歴が記録されることが多くなったが，それでも記録に残らない変更，例えばファイルをコピーしたという記録などが存在する．また，開発組織自体の分離によって履歴が追跡できなくなることも起こりうる．

このように，類似ソフトウェア間での開発管理が行われていない状況に対応するため，ソフトウェアプロダクトファミリのソースコードのみを利用してソフトウェアプロダクトライン構築のためのソフトウェア選択を支援する技術が必要とされている．

本研究では，既存のソフトウェアプロダクト集合からソフトウェアプロダクトラインを構築するに当たり，機能比較を行う対象となるソフトウェアの選定作業の助けとなるよう，ソフトウェアの派生関係を模した木，派生関係木を構築する．類似ソフトウェア間での開発管理が行われていない状況に対応するため，派生関係木の構築を，ソフトウェアのソースコードのみを入力とし，その類似度をソフトウェア間の距離とみなした最小全域木を求める問題として定義した．距離関数の定義は類似したソースファイルの数や差分の行数を用いて 10 通りを定義し，派生の方向を示すラベルを取り出すヒューリスティックを 3 種類定義した．オープンソースソフトウェアに対して行った適用実験では，類似ソースファイルの数をソフトウェア間の距離として構築した派生関係木の形は約 88%が，またそのときにソースファイルのファイルサイズの増減を基準として計算した派生の方向も約 86%が開発履歴と一致し，開発履歴を近似した派生関係木をソースコードのみから構築できることを確認した．

第 2 章では，本研究の背景となる事項と関連する研究について述べる．その後，第 3 章で派生関係木を定義し，第 4 章で提案する解析手法について説明する．第 5 章で複数のオープンソースソフトウェアに対し，手法を適用した結果を述べる．第 6 章に関連研究を示し，最後に第 7 章で，まとめと今後の課題を記述する．

2　背景

2.1　ソフトウェアプロダクトラインエンジニアリング

プロダクトラインエンジニアリングは，元々ハードウェアの分野で提唱された大量個別生産に対応するための考え方である．個々の顧客のニーズを満たす製品を効率よく生産するために，全製品に共通のプラットフォームを導入する．例えば自動車の場合，サスペンションシステムやギアボックスなどがプラットフォームに含まれる．このプラットフォームの上に，個々の部品である大きさの異なるエンジン，オープンカーやセダンといった形の違うボディ，その他にも多くの可変部分を組み合わせて，共通のプラットフォームから個別の製品を開発する．

ソフトウェアの開発においても，複数の類似するソフトウェアが大量に生産される例があり，同一製品シリーズの機種別のソフトウェアを作成する場合や，同一目的のシステムを，利用する顧客毎に作成する場合などが挙げられる．このようなソフトウェア群を開発するにあたり，開発費用の低減，品質の向上，市場投入期間の短縮を動機として SPLE を導入する動きがある．

しかし，ソフトウェア開発の現場において，必ずしも開発の初期段階から再利用を考慮した設計がなされているとは限らない．Rubin らも述べている通りソフトウェアの開発はしばしば ad-hoc である [17]．つまり，必要な時に，必要なソフトウェアをコピーし改変することで，新たなソフトウェアを生産するのである．

このため，SPLE が適用されていない開発現場において，既存の類似ソフトウェア集合からソフトウェアプロダクトラインを構築する手法が必要とされている．

2.2　ソフトウェアの進化における分岐とその原因

ソフトウェアの進化は必ずしも一本道ではなく，複数のソフトウェアに分岐したり，また逆に複数のソフトウェアが持つ機能を 1 つにまとめて統合したりすることがある．ソフトウェアの分岐，統合は，1 つのソフトウェアプロダクトの中で起きることもあれば，複数のプロダクトにまたがっておこることもある．

ソフトウェアの分岐が起こる原因として，以下の 3 つが考えられる．

- 原因 1: 同一プロダクト内での開発ブランチと保守ブランチの分離
- 原因 2: 要求する機能の違いによる分離
- 原因 3: 開発組織の分離

以下それぞれについて詳しく述べる．

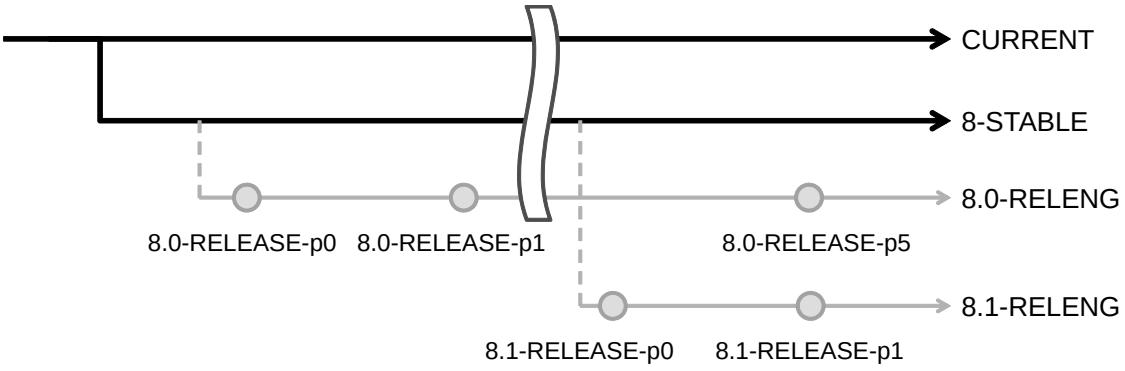


図 1: FreeBSD のブランチ

2.2.1 原因 1: 同一プロダクト内での開発ブランチと保守ブランチの分離

ソフトウェアの開発において，リリース用のリリースブランチと機能追加・テスト用の開発ブランチとにソースコードを分岐させることが行われている．また，リリースした後のソフトウェアに対して，次のバージョンへの機能追加を行うリリースブランチとは別に，セキュリティ修正やバグ修正などを行うための保守ブランチを分岐させることもある．

- FreeBSD[1]の開発においては，RELENG(RELease ENGineering) ブランチから各 RELEASE が安定版として公開される他に，CURRENT と STABLE の 2 つの開発ブランチが存在する．これらのブランチの関係を図示すると図 1 のようになる．CURRENT には開発中の機能やソフトウェアが含まれ，STABLE は CURRENT から分岐してリリース準備に入るためのブランチである．

2.2.2 原因 2: 要求する機能の違いによる分離

必要とする機能や目的の違いにより，開発が分岐する．必要とする機能の違いとは，単純に機能追加や欠陥修正によるバージョンアップではなく，現在あるソフトウェアとは別の機能を付加したものが要求されるようなケースを指す．また，目的の違いとは，対象とする顧客が元のソフトウェアとは異なるために，別製品として分岐するようなケースを指す．

- ある企業で開発された組み込みソフトウェアでは，元となった 1 つの製品から保守による改版や機能の差異による派生によって 25 の製品群に進化した [22]．さらに 1 製品の中でもバージョンアップ以外にソフトウェアの派生が発生していることが確認されている (図 2).

[1]The FreeBSD Project, `http://www.freebsd.org/`

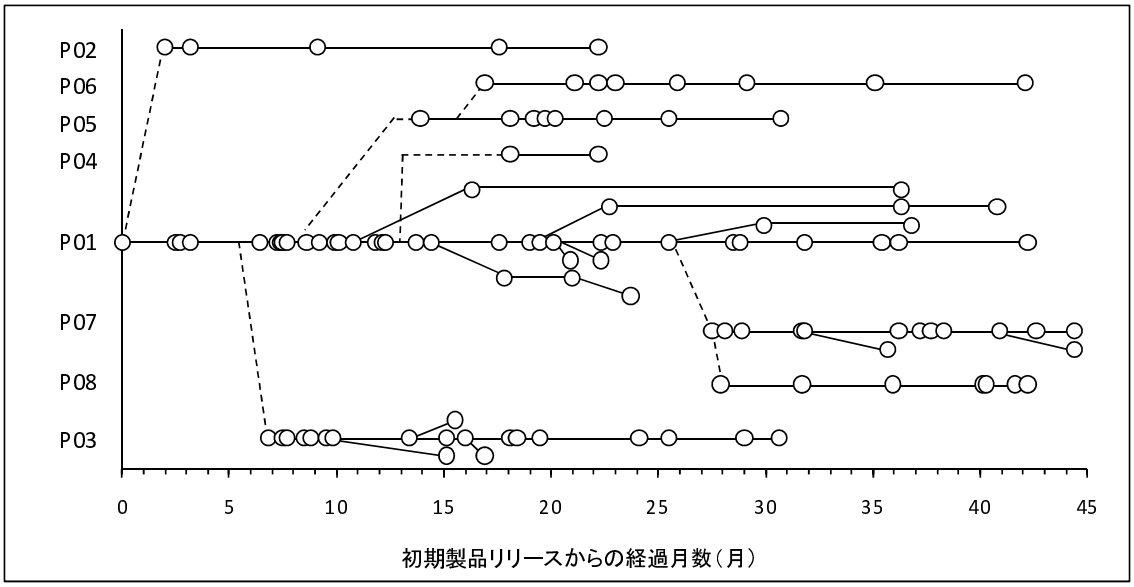


図 2: 製品の改版履歴:文献 [22] より引用

- Fedora[2]と Red Hat Enterprise Linux(RHEL)[3]は，共に Red Hat Linux を源流に持つ，そして今も関連の強い Linux ディストリビューションである．Red Hat Linux は米 Red Hat 社が開発していたオープンソースの Linux ディストリビューションであった．無料で利用できたが，有償でのサポートも用意されていた．米 Red Hat 社は無料の Linux ディストリビューションとして Fedora プロジェクトを独立させ，自身は企業向けに有料の RHEL を提供することになった．その後は Fedora での開発成果が RHEL に取り込まれている．また，RHEL はオープンソースライセンスに基づきソースコードが公開されており，そのソースコードから商標や商用パッケージを取り除いた OS が作成されている．このような RHEL のクローンの中で代表的なものとしては CentOS[4]が挙げられる．これらの派生ディストリビューション間の関係を簡単に図 3 にまとめる．

- FreeBSD や NetBSD[5]，OpenBSD[6]などの BSD 系のオペレーティングシステムは，どれも 4.4BSD Lite や 4.4BSD Lite2 から派生したものである．

開発ブランチや保守ブランチによる分岐は，1 つのソフトウェア内での出来事であるが，要求する機能や目的の違いにより分岐したソフトウェアは，開発プロジェクトが別個のもの

[2]Fedora Project Homepage, `http://fedoraproject.org/`
[3]Red Hat — Red Hat Enterprise Linux, `http://www.redhat.com/products/enterprise-linux/`
[4]www.centos.org - The Community ENTerprise Operating System, `http://www.centos.org/`
[5]The NetBSD Project, `http://www.netbsd.org/`
[6]OpenBSD, `http://www.openbsd.org/`

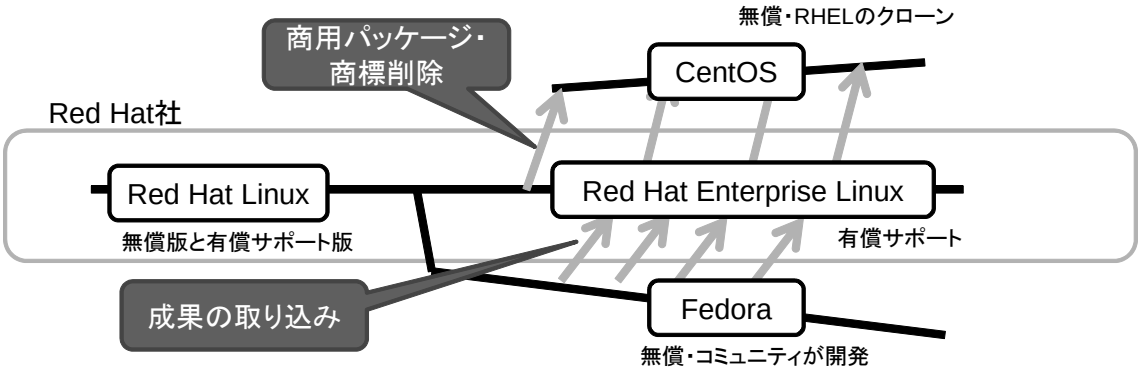


図 3: Red Hat Linux の派生

となることがある．ある製品が分岐した時，同一の製品の機能が違う別バージョンとなるか，別個の製品として扱われるかは開発側の主観によって区別される．

2.2.3 原因 3: 開発組織の分離

目的が同一のソフトウェアを開発していても，開発組織が分離してプロジェクトとして別個のものになるケースが見られる．オープンソースソフトウェアでいくつかこのケースに当てはまる例が知られているが，オープンソースでないソフトウェアであっても派生ソフトウェアのすべてが同じ組織で開発されるとは限らない．例えばあるソフトウェアの派生版が，開発元とは別の部署でも利用され，その部署で機能追加や改変を行う場合が考えられる．このとき，開発の履歴が元のソフトウェアを開発した部署と派生先の部署で共有されない可能性もある．

このようにしてリポジトリの分断が起こると，以降互いのソフトウェアがどのような機能を取り込んで進化したか，その情報が失われてしまう．互いの成果を取り込みあったり，同一のパッチをあてたりといった情報は改めて探索しなければならない．

- FFmpeg は動画・音声などのマルチメディアデータを扱うことのできるライブラリ・プログラム群である．このプロジェクトは開発体制の対立により，FFmpeg と Libav に分岐した [13].

- オープンソースのオフィススイートとして有名だった OpenOffice.org は，当時の開発元であったサン・マイクロシステムズがオラクルに買収された際，プロジェクトの主要メンバーが開発した LibreOffice[7] とオラクルが Apache 財団にソースコードを寄贈し

[7] LibreOffice, `http://www.libreoffice.org/`

て発足した Apache OpenOffice[8]とに分岐した．

これらののオープンソースソフトウェアは分岐後も互いの成果をプロジェクトに取り込みあっているが，開発のリポジトリは分断されている．

リポジトリが共有されているソフトウェアでも多くのローカルな変更点が共有されない状況では，リポジトリが分断されたときと同様，開発の履歴を追跡することが難しくなる．Linux カーネルは様々な企業の組み込み製品にも利用されており，各企業で独自のカスタマイズが行われている．その変更点のパッチは，公開されていても Linux カーネル開発のメインラインには共有されないことがあり，また独自のカスタマイズがメインラインと同期されず古いものとなってしまう，といった問題点が Hemel らの研究で明らかにされた [5]．

2.3 コードクローン検出技術

ソースコードの類似性に関する研究として，コードクローン検出技術が挙げられる．コードクローンとは，ソースコード中に存在する互いに一致・類似したコード片のことであり，大規模なソフトウェア集合においてもソフトウェア間でのコードクローンが多く存在することが知られている．

コードクローン検出技術は様々な手法が開発されており [1, 7]，大規模なソフトウェアに対して適用するために，コピー&ペーストを検出してバグを発見する手法 [10] や，分散処理によるコードクローン検出ツールの開発 [11]，ファイル単位でのクローンに着目した高速な検出 [23] などの研究が行われている．

複数のソフトウェアから類似するファイルを特定する目的で，コードクローン検出技術が利用されている例がある．Yoshimura らは産業向けシステムのソースコードについて，類似度の高いファイルの存在を可視化した [20]．Yoshimura らの手法では，どのファイル同士が類似しているかをグラフの形で図示することができる．Inoue らが開発した Ichi Tracker[6] では，コード断片をソースコード検索エンジンで検索し，検索結果のソースファイルをコード断片との類似度でクラスタリングする．その上で，ソースファイルを時系列順に並べることによりソースコードの再利用の経緯を可視化する．このツールでは開発者が注目している 1 ファイルとの類似度だけを用いてファイルの再利用関係の可視化を行っている．

[8] Apache OpenOffice - The Free and Open Productivity Suite, `http://www.openoffice.org/`

3　派生関係木の定義

あるソフトウェア A を変更してソフトウェア A' を作成したとき，A と A' の間には派生関係があると言える．このとき，派生関係の向きは A から A' となる．また，あるソフトウェアの製品系列から別のソフトウェア製品系列へ開発が分岐することもある．派生関係は，ソフトウェアの理解においても重要な意味を持つ．

ソフトウェアの最新版は，同じソフトウェア系列の古い版に比べて，バグ修正が適用済みであるという利点がある．また，バージョンアップに伴い機能が増えることは多いが，機能が削除されることは少ない．逆に，最も古い版や分岐直前のバージョンを調査することもソフトウェアの変化を知るうえで重要である．ソフトウェア間の派生関係がわかっていれば，ソフトウェア系列がどこでいくつに分岐していて，またそれらの最新版がどれかがわかるため，このような選択が容易になる．

そこで本研究では，ソフトウェアプロダクト集合に対して派生関係木というものを構築し，ソフトウェアプロダクト集合の実際の派生関係に近い情報を得られるようにする．派生関係木は，ソフトウェアプロダクト集合 $P = \{p_1, p_2, ..., p_n\}$ に対して，ある距離関数 C を用いて最小全域木を求めたものに，各辺の派生方向 $\{\leftarrow, \rightarrow, =\}$ をラベルづけしたものである．$\{=\}$ は派生方向が定まらなかったときにつける．

距離関数は，ソースファイル間の類似度や差分に基づいて計測する．ソフトウェアは，ソースコードを追加・修正・削除することによって進化する．よって，派生関係にあるソフトウェア間では，ソースファイル同士が特に似ていると考えることができる．そこで，ソースファイル間の類似度を測定し，類似ファイル集合を抽出し，その結果をもとにソフトウェア間の距離を表現する距離関数を定義した．

このようにして構築した派生関係木は，ソフトウェアの進化・分岐を模した木である．派生関係木を得ることで，バージョン管理システムなどの開発の履歴が残っていないような場合でも，ソフトウェアがどこから始まりどこで分岐し，分岐後の最新版がどれなのかを知ることができる．

3.1　最小全域木

全域木とは，あるグラフの部分グラフのうち，すべての頂点が連結されており，かつ閉路がないグラフである．最小全域木とは，グラフを構成する辺の重みの総和が最小となる全域木である．グラフ $G = (V, E)$ に対する最小全域木 $T = (V, E')$ は，辺 $(v, w) \in E$ の重みを表す関数を $C(v, w)$ として重みの合計 $\sum_{(v,w)\in E'} C(v, w)$ が最小となるような木である．

最小全域木を求めるアルゴリズムとして，Prim 法 [16] がある．Prim 法では，始めに元となるグラフから任意の頂点を木に追加する．木に含まれる頂点から木に含まれない頂点への

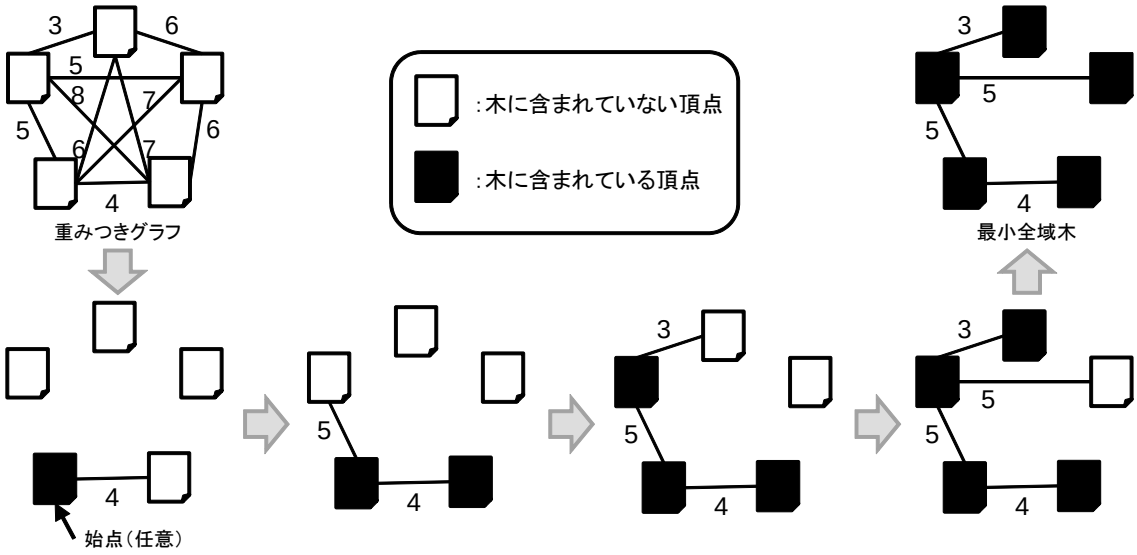


図 4: Prim 法による最小全域木の構築

辺のうち，重みが最も小さいものを順次木に追加していく．図 4 の例では，初めに左下の頂点を選択した後，その頂点からの重みが最も小さい重み 4 の辺とその辺で結ばれた頂点とを木に追加し，以後同様の作業を，閉路ができないようすべての頂点が木に含まれるまで繰り返している．重みが最も小さい辺が複数ある場合は，そのうちどれか 1 つを選択する．選択する辺によって最小全域木の形は変わるものの，最終的な重みの合計は変化しないことがアルゴリズムで保証されている．この作業をすべての頂点が木に含まれるまで行った時，作成した木は最小全域木となっている．なお，辺の重みについては負の値も許す．

3.2 ソースファイル間の派生関係

ソフトウェア間の比較は，そのソースファイル同士を比較する作業である．このことから，派生関係にあるソフトウェアを調べたい場合は，派生関係にあるファイルを調査すればよいのではないかと考えた．

ここでは，派生関係がないファイル間よりも，派生関係があるファイル間の方が差分の行数が小さいと仮定する．つまり，あるソースファイル a と，a から見て最も差分の行数が小さいソースファイル b との間には派生関係があるとみなす．ソフトウェアの派生関係木と同様に，ソースファイル集合 $F = \{f_1, f_2, ..., f_n\}$ に対して，差分の行数を距離として最小全域木を求めたものに，各辺の派生方向 $\{\leftarrow, \rightarrow, =\}$ をラベルづけしたものをソースファイル間の派生関係木とする．

4　派生関係木構築手法

ソフトウェアプロダクトライン構築の際のソフトウェアの選択を支援するために，類似ソフトウェア間の派生関係木を構築する手法を提案する．類似ソフトウェア間での開発管理が行われていない状況を想定し，ソースコードから得られる情報のみによって派生関係木の構築を行う．

本手法は，ソフトウェアプロダクト集合のソースコードを入力として与えると，ソフトウェアプロダクトの派生関係木を出力する．

提案手法は，以下の 3 ステップで実現する．

ステップ 1．類似ファイル集合の構築　類似ソフトウェア集合のソースコードからソースファイル間の差分をとり，類似度を計算する．類似度計算の結果から，類似度の高いファイル同士を集めた類似ファイル集合グラフを構築する．ここまでの手順は Yoshimura らの手法 [20] に基づいている．

さらに，構築した各類似ファイル集合部分グラフに対し，特に差分の行数が少ないファイル同士を接続したファイル間の派生関係木を計算する．

ステップ 2．ソフトウェア間の距離計算　類似ファイル集合グラフやファイル間の派生関係に含まれる辺，類似度，差分の行数をもとに，ソフトウェア間の距離を計算する．ソフトウェア間の距離は複数の定義を提案する．

ステップ 3．ソフトウェア間の派生関係の取得　ソフトウェアプロダクト集合に対し，ソフトウェア間の距離に基づいた最小全域木を構築する．さらに派生方向を示すラベルを全域木の各辺に付して，ソフトウェア間の派生関係木とする．

以下では各ステップについて順を追って説明する．

4.1　ステップ 1．類似ファイル集合の構築

このステップではソフトウェア間の距離を計算する準備として，ソフトウェア内に含まれるソースファイル間の類似度および派生関係を抽出する．

まず，入力されたソフトウェア集合のソースファイル同士が，どれだけ類似しているかを計算し，類似度の高いファイル同士を集めて類似ファイル集合を構築する．

次に，類似ファイル集合グラフ内で差分が小さいファイル同士を接続して最小全域木を作り，ソースファイル間の派生関係を得る．

4.1.1 ソースファイル間の類似度

2 つのソースファイル間の類似度 $sim(a,b)$ は以下のように計算する．

ファイル a,b からコメント・空白空行を除去し，トークン分割してトークン列 a_t,b_t を作成する．次に，トークン列 a_t,b_t を比較して，式 (1) の計算を行う．

$$sim(a,b) = \frac{LCS(a_t,b_t)}{LCS(a_t,b_t) + ADD(a_t,b_t) + DEL(a_t,b_t)} \tag{1}$$

ただし

$$LCS(a_t,b_t) = \text{トークン列 } a_t,b_t\text{間の最長共通部分列の長さ}$$
$$ADD(a_t,b_t) = \text{トークン列 } a_t\text{から } b_t\text{に追加された部分列の合計長}$$
$$DEL(a_t,b_t) = \text{トークン列 } a_t\text{から } b_t\text{で削除された部分列の合計長}$$

とする．

ファイルの比較には Unix diff[12] を用いるのが一般的である．Unix diff は行単位での比較を行うものであるため，1 トークン 1 行としたファイルを入力として Unix diff による差分を得る．Unix diff の出力例は図 5 のようになっており，出力から LCS を求めるよりも ADD と DEL を計測したほうが早い．そこで，差分の行数 $diff(a,b)$ を $diff(a,b) = ADD(a_t,b_t)+DEL(a_t,b_t)$ とし，置換を追加・削除に分割したうえで $LCS(a_t,b_t) = (|a_t| + |b_t| - diff(a_t,b_t))/2$ の関係式を用いて式 1 を式 2 に変形して計算を行う．

$$sim(a,b) = \frac{|a_t| + |b_t| - diff(a_t,b_t)}{|a_t| + |b_t| + diff(a_t,b_t)} \tag{2}$$

ファイル間類似度を計算したことにより，N 個のソースファイルに対しソースファイルを頂点，ファイル間の類似度 sim を辺の重みとした無向重みつきグラフ G が得られる．

$$\begin{aligned} V &= \{v_i\}, i = 1, ..., N \\ E &= \{e_{i,j}\}, i \neq j \\ G &= (V, E, sim) \end{aligned}$$

4.1.2 類似ファイル集合グラフ

計算で得られた類似度をもとに，互いに類似関係にあるファイルをグループ化した類似ファイル集合グラフ Gs を構築する．

類似度計算で得られた，ソースファイルを頂点，ファイル間の類似度を辺の重みとした無向重みつきグラフ G から，類似度がある一定のしきい値 t 未満の辺を削除する．ただし

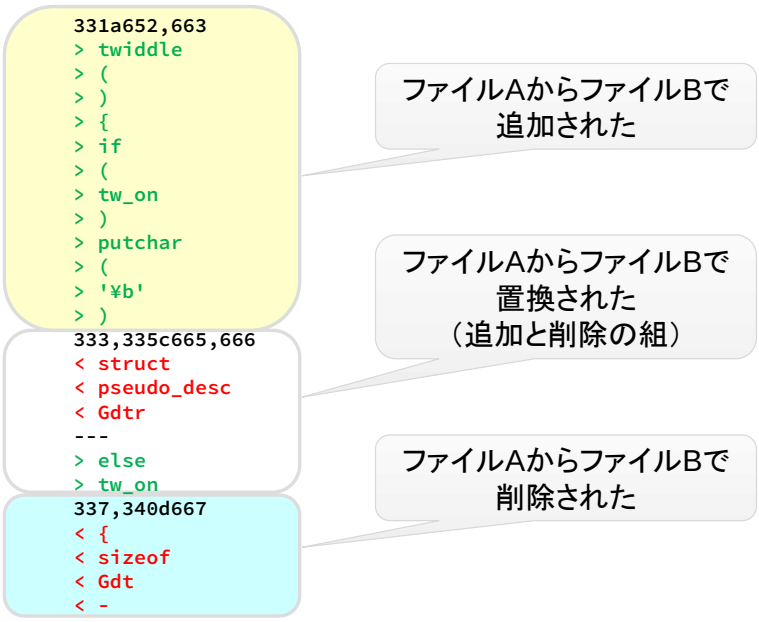


図 5: Unix diff の出力例

$0 < t \leq 1$ である．これにより類似度の高いファイル同士のみが連結されたグラフ Gs が生成される．

$$
\begin{aligned}
Es &= \{e \in E \mid sim(e) \geq t\} \\
Gs &= (V, Es, sim)
\end{aligned}
$$

辺を削除したことにより，グラフは非連結な複数のグラフに分割される．この分割されたそれぞれのグラフのうち，2 つ以上の頂点を含むものを以後類似ファイル集合部分グラフと呼ぶ．分割された各グラフは，大量のソースファイル集合から類似関係にあるファイル同士をグループ化したものとなる．

図 6 では，グラフ G から類似度未満の辺を削除して Gs を作っている．Gs からは 3 つのファイルが属する部分グラフ Gs_1，4 つのファイルが属する部分グラフ Gs_2 が類似ファイル集合部分グラフとして抽出される．また，他のどのファイルとも接続されていないファイルが 1 つあるが，これは類似ファイル集合としては抽出しない．

この定義では，類似関係にないファイル同士も同一の類似ファイル集合部分グラフに属することがある．例えば図 6 中の Gs_2 においては，ファイル a とファイル b 間の類似度が閾値未満であったためこの 2 ファイル間に辺はひかれていないが，ほかのファイルを介して同一の類似ファイル集合部分グラフに属している．以降の計算では，類似ファイル集合部分グラフの辺を用いて計算を行うため，ファイル a とファイル b は同一の類似ファイル集合部分グ

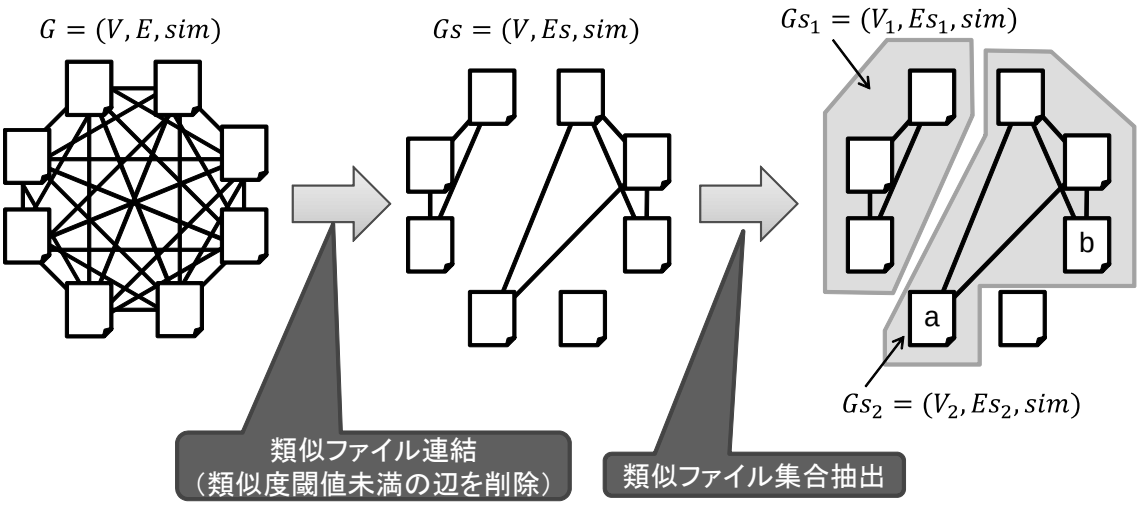


図 6: 類似ファイル集合部分グラフの抽出

ラフに属しているが互いに類似しているファイルとしては扱わない．

4.1.3　ソースファイル間の派生関係構築

類似ファイル集合内に含まれるソースファイル間の派生関係 Gt を構築する．

類似ファイル集合 Gs の各類似ファイル集合部分グラフ Gs_i に対し，辺の重みを差分の行数とした無向重みつきグラフ $Gd_i = (V_i, Es_i, diff)$ を生成する．無向重みつきグラフ Gt_i に対し最小全域木 Gt_i を構築し，ソースファイル間の派生関係とする．各類似ファイル集合について最小全域木を構築した後のソースファイル集合全体を表すファイル間派生関係グラフ Gt は以下のようになる．

$$
\begin{aligned}
Gt_i = &\quad (V_i, Et_i, diff), \text{ただし } Gt_i \text{は } Gd_i \text{から構築した最小全域木} \\
Gt = &\quad (V, Et, diff), Et = \bigcup Et_i
\end{aligned}
$$

4.2　ステップ 2. ソフトウェア間の距離計算

ソフトウェア間の距離を表現するための距離関数 C を複数定義した．ソフトウェア間の距離は，そのソースファイル間の類似度などをもとに計算され，類似しているファイルの組の個数を数える関数や類似度を合計するような値が大きいほど距離が小さい関数と，差分となるトークン数に着目した値が小さいほど距離が小さい関数がある．値が大きいほど距離が小さい関数は，値に負号をつけて値が小さいほど距離が小さいようにする．このようにすることで，どちらの関数でも同一のアルゴリズムで最小全域木を求めることができる．

4.2.1 ソースファイルの関係グラフの辺の本数ベース

類似度の高いまたは派生関係にあるソースファイルがいくつあるか，ステップ 1 で得られたグラフの辺の本数を数える．辺の本数が多ければ多いほど，ソフトウェア間の距離が近いと考えられる．

類似度の高いソースファイルの数　ソフトウェア A,B 間について，類似度の高いソースファイルの数 $C_s(A,B)$ を計測する．ソフトウェア A のソースファイル a, ソフトウェア B のソースファイル b を結ぶ辺が類似ファイル集合グラフ Gs に含まれているとき，これを 1 本と数える．このような辺の集合 $E_s(A,B)$ と，その数を表す距離関数 $C_s(A,B)$ は以下のように定義される．

$$\begin{aligned} E_s(A,B) &= \{(a,b) \in Es \mid a \in A, b \in B\} \\ C_s(A,B) &= -|E_s(A,B)| \end{aligned}$$

派生関係にあるソースファイルの数　また，互いに類似しているというだけでなく，その中で派生関係にあるファイルの個数を数えることも考えた．ファイルごとの派生関係がソフトウェアのバージョン履歴と比べて正確なら，類似度の高いソースファイルの数よりも派生関係にあるソースファイルの数のほうがよりバージョン履歴に近い距離関数になると考えたからである．

ソフトウェア A,B 間について，ファイル間派生関係グラフ Gt 内で結ばれているソースファイルの数 $C_t(A,B)$ を計測する．ソフトウェア A のソースファイル a, ソフトウェア B のソースファイル b を結ぶ辺がファイル間派生関係グラフ Gt に含まれているとき，これを 1 本と数える．このような辺の集合 $E_t(A,B)$ と，その数を表す距離関数 $C_t(A,B)$ は以下のように定義される．

$$\begin{aligned} E_t(A,B) &= \{(a,b) \in Et \mid a \in A, b \in B\} \\ C_t(A,B) &= -|E_t(A,B)| \end{aligned}$$

以上の 2 種類の距離関数は，2 つのソフトウェア間で類似度の高いまたは派生関係にあるファイルの組の個数を計測するものである．図 7 の例では，類似度の高いソースファイルがソフトウェア A,B 間で 4 組あるため $C_s(A,B) = 4$, このうち 1 つの辺は，ファイル間派生関係を構築すると消えるため $C_t(A,B) = 3$ となる．ソフトウェア間の関係であるため，ソフトウェア B 内にひかれている辺は数えないことに注意する．

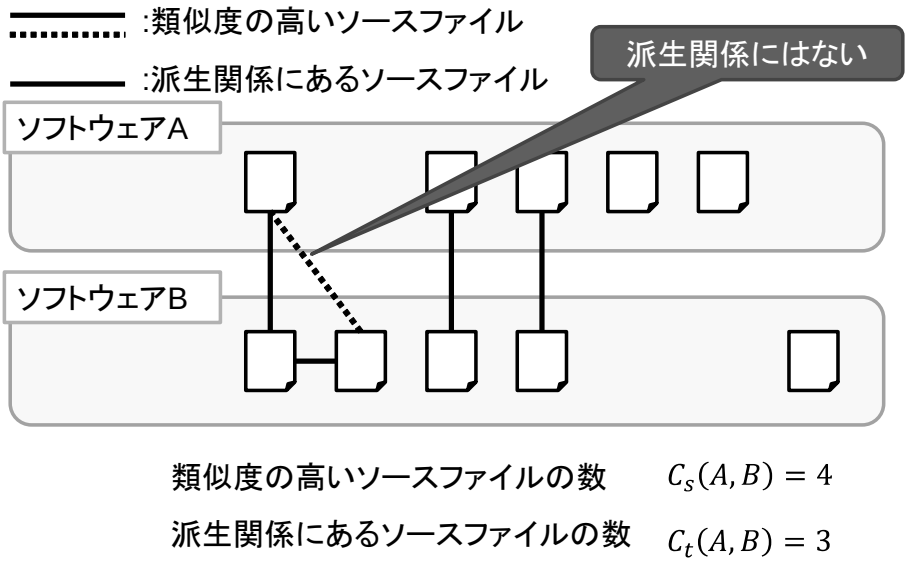


図 7: C_s, C_t の求め方

4.2.2 辺に重みを付けた距離関数

4.2.1 項で計測した距離関数は，ソフトウェア間で類似度の高いファイルの組の個数，または派生関係にあるファイルの組の個数を計測するものであった．ここでは，その各辺に重みを付けた距離関数を以下に定義する．

ソースファイル間の差分の行数の合計・平均　ソフトウェア A,B 間について類似度の高いソースファイル間の差分の行数を合計した ${C_s}^{diff}(A,B)$，派生関係にあるソースファイル間の差分の行数を合計した ${C_t}^{diff}(A,B)$ を定義する．

ソフトウェアに対する機能の追加・削除・修正はソースコードの追加・削除・修正に対応するはずであるから，ソフトウェア同士が似ている場合，ソースコードの差異は小さくなると仮定することができる．

$$
\begin{aligned}
{C_s}^{diff}(A,B) &= \sum_{e \in E_s(A,B)} diff(e) \\
{C_t}^{diff}(A,B) &= \sum_{e \in E_t(A,B)} diff(e)
\end{aligned}
$$

また，ファイル間の差分の行数の合計ではなく，平均的な編集量がソフトウェアの距離と関係していると仮定することもできる．そこで，ソースファイル間差分のファイル平均である $\overline{{C_s}^{diff}}(A,B)$ と $\overline{{C_t}^{diff}}(A,B)$ を導入する．

ソースファイル間の類似度の合計・平均　ソフトウェア A,B 間について類似度の高いソースファイル間の類似度を合計した ${C_s}^{sim}(A,B)$，派生関係にあるソースファイル間の類似度を合計した ${C_t}^{sim}(A,B)$ を定義する．

類似度が閾値以上の辺を対象とするため辺の本数 C_s,C_t と近い結果になると推測されるが，ソフトウェア間でファイル数に変化がないときに細かな差異を見るための 1 つの指標になると期待できる．

$$
\begin{aligned}
{C_s}^{sim}(A,B) &= -\sum_{e \in E_s(A,B)} sim(e) \\
{C_t}^{sim}(A,B) &= -\sum_{e \in E_t(A,B)} sim(e)
\end{aligned}
$$

これらについても，ソースファイル間が平均的にどの程度似ているかを示す平均類似度 $\overline{{C_s}^{sim}}(A,B)$ と $\overline{{C_t}^{sim}}(A,B)$ を導入する．

4.3　ソフトウェア集合に対する派生関係木の取得

ソフトウェア集合に対する無向派生関係木は，解析対象のソフトウェアを頂点，各ソフトウェア間の距離を辺の重みとしたグラフから構築した最小全域木で表現される．

派生関係木の形を求めた後，それぞれの辺に対し派生方向を計算してラベルづけをする．

4.3.1　無向派生関係木の取得

4.2 節で定義したコスト関数を重みにもつアプリケーション集合のグラフに対し，最小全域木を構築する．コスト関数はアプリケーションの向きを考慮せず計算するため，この段階では辺で結ばれた 2 つのソフトウェアのどちらが派生元でどちらが派生先かは明らかではない．

4.3.2　派生方向の計算

ここまでで得られた派生関係木は，ソフトウェア間の距離を基準とするため，向きが定まっていない．

そこで，派生関係木の各辺に対して派生の向きを計算しラベルづけを行う．ラベルは，$\{\rightarrow, \leftarrow, =\}$ の 3 種類である．$\{=\}$ は派生方向が定まらなかったときに付ける．派生関係の計算方法として，ソフトウェアのソースコード全体でのメトリクスを比較する方法と，ソフトウェア間で対応するファイル間の派生関係の辺に対して方向を求め，その合計本数によって向きを定める方法とを考えた．

d_{ALL}**：ソフトウェア全体での計測**　ソフトウェアは進化の際，ソースコードの削除よりも追加・変更のほうが多く行われると仮定する．

ソフトウェア A,B について，類似関係にあるファイル間の差分の増減を調べる．増減を全類似ファイル集合について合計し，増加する方向に派生方向を定める．

$$\text{派生方向:} \begin{cases} A \to B, & \sum_{e \in E_s(A,B)} ADD(e) > \sum_{e \in E_s(A,B)} DEL(e) \\ A = B, & \sum_{e \in E_s(A,B)} ADD(e) = \sum_{e \in E_s(A,B)} DEL(e) \\ A \leftarrow B, & \sum_{e \in E_s(A,B)} ADD(e) < \sum_{e \in E_s(A,B)} DEL(e) \end{cases}$$

ファイル単位での派生方向の合計　ファイル単位の派生関係を表す最小全域木の各辺に対し，派生方向を定める．ソフトウェア A のソースファイル a とソフトウェア B のソースファイル b を結ぶ辺 e が，ある類似ファイル集合に含まれているとき a,b 間の派生方向を調べ，a → b である辺の数を E_f，b → a である辺の数を E_b とする．この本数が多い方向が，ソフトウェア間の派生方向である．

$$\text{派生方向:} \begin{cases} A \to B, & |E_f| > |E_b| \\ A = B, & |E_f| = |E_b| \\ A \leftarrow B, & |E_f| < |E_b| \end{cases}$$

ファイル間の派生方向については，以下の 2 種類の基準を用いて調べる．

d_{SIZE}**：ファイルサイズが小さいファイルから大きいファイル**　ソフトウェアは進化の際，ソースコードの削除よりも追加・変更のほうが多く行われると仮定する．

ファイル a のトークン化前のファイルサイズ $size(a)$ とファイル b のトークン化前のファイルサイズ $size(b)$ を比較し，小さいほうから大きいほうへ派生方向を定める．

$$\begin{aligned} E_f &= \{(a,b) \in E_s(A,B) \mid size(a) \leq size(b)\} \\ E_b &= \{(a,b) \in E_s(A,B) \mid size(a) \geq size(b)\} \end{aligned}$$

d_{INC}**：トークン列が増加する方向**　ソースコード全体の量ではなく，トークンがいくら増えたかに注目する．コメントなどを除去しているので，ソースコード自体の増加量ともいえる．類似度計算の際に，計算したトークン列の差分を用い，トークンの量が増加する方向に派生方向を定める．

$$\begin{aligned} E_f &= \{(a,b) \in E_s(A,B) \mid ADD(a_t,b_t) \geq DEL(a_t,b_t)\} \\ E_b &= \{(a,b) \in E_s(A,B) \mid ADD(a_t,b_t) \leq DEL(a_t,b_t)\} \end{aligned}$$

4.4 手法の制約

全域木を構築するため，抽出した派生関係には閉路が存在しない．これは，あるソフトウェアに分割と統合の両方が起こっていた場合でも，本手法では良くてもそのどちらかしか取得できないことを意味する．

ファイル間類似度の閾値 t は $0 < t \leq 1$ としたが，閾値 1 の時に限り距離関数 C_s，C_t 以外の距離関数はすべてのソフトウェア間で値が等しくなるため意味をなさない．またこのとき，派生方向の計算についても，すべてのファイル間の派生方向が定まらないため意味をなさない．よって，類似度の閾値を 1 にした時に得られる情報は C_s，C_t の距離関数を用いて作成した派生関係木の形のみである．

5　実験

提案手法を実装して実験を行った．対象言語は C 言語とし，トークン化を行う．
以下のような状況を想定し，データセットを作成した．

(1) 開発が分岐していないソフトウェアプロダクト集合
　– PostgreSQL-magor

(2) 組織内で開発が分岐したソフトウェアプロダクト集合
　– PostgreSQL-8-ALL

(3) ソフトウェア製品ファミリのうち新しいいくつかの製品しか残っていない場合
　– PostgreSQL-latest

(4) ソフトウェア製品ファミリのうち過去の製品の一部が欠落している場合
　– PostgreSQL-annually

(5) プロジェクトが 2 つに分岐した場合
　– FFmpeg

(6) プロジェクトが 3 つ以上に分岐した場合
　– *-BSD

5.1　評価方法

実験結果に対する評価は，派生関係木の形と，辺に対する方向付けの 2 段階で行う．

派生関係木の形についての評価は，抽出した派生関係木の辺とソフトウェア開発でのバージョン履歴とを照合して行う．ソフトウェア開発でのバージョン履歴は，バージョン管理システムの履歴，プロジェクト分岐の履歴などから取得した，あるソフトウェアの特定のバージョンがどのソフトウェアのどのバージョンから作られたものかを示す情報である．まず，抽出した派生関係木の辺が，バージョン履歴と一致した本数を数える．抽出した派生関係木はソフトウェア数 N に対し $N-1$ 本の辺となっているため，この値は 0〜$N-1$ の範囲で変動し $N-1$ が最良である．この時点である程度，派生関係が正しく抽出できたかがわかる．

次に，抽出した派生関係木の辺のうちバージョン履歴と一致しなかったものと，バージョン履歴に出現するが派生関係では抽出できなかった辺について調べる．これについては，一致しなかった本数を数えることだけは不十分である．抽出した派生関係のうち，バージョン履歴に一致しない本数は $(N-1)$ − 正解数 になるが，不正解だった辺がバージョン履歴上で近しい関係にあるわずかにずれていたバージョン同士を結んでいたのか，全く関連のな

いバージョン同士を結んでいたのかによって，同じ不正解でも大きく意味が変わるからである．また，不正解だった辺がバージョン履歴では表現されていないソフトウェア間の関係を示している可能性も考えられる．例えば，リポジトリとしては分離していたが実際は互いの変更点を取り込んでいた場合である．また，分岐統合によってバージョン履歴の辺の本数が $N-1$ 本を超えることもある．

このような例があるため，単に辺の正解・不正解を判定するだけでなく，不正解の要因を検証することが重要である．

また，派生関係木の辺に対する方向付けについては，派生関係木の形のうち良い結果が出たものについて，提案した手法を適用することにする．派生関係木の形が一致した辺について，その派生方向が一致したか，逆転したか，派生方向が定まらなかったかを数える．

以下では，実験対象のデータセットごとに結果と考察を述べる．

5.2 PostgreSQL-major: 開発が分岐していないソフトウェアプロダクト集合

PostgreSQL[9] は，オープンソースのデータベース管理システムである．開発言語は C 言語である．プロジェクトは git でバージョン管理されており，リリース版が出るたびにバージョン管理システム上でタグが付与されている．

PostgreSQL-major は，PostgreSQL のメジャーリリースのうち 7.0 以降 9.2 までを集めたデータセットで，7.0, 7.1, 7.2, 7.3, 7.4, 8.0.0, 8.1.0, 8.2.0, 8.3.0, 8.4.0, 9.0.0, 9.1.0, 9.2.0 の計 13 バージョンからなる．拡張子.c のファイルが 96448 ファイル，コメント空行を除いた総行数は 48478395 行あった．表 1 に，各バージョンのファイル数と総行数を示す．ファイル数，総行数ともにリリースが行われるたびに増加している．

マイナーリリースによる分岐の影響を受けないため，派生関係はバージョン番号順に一列に並ぶことが期待される．7.4 の次は 8.0.0 のベータ版の開発が始まっており，8.4.0 の次も 9.0.0 のアルファ版（当初は 8.5 系とされていた）が開発されている．また，メジャーリリース後はマイナーバージョンが別のブランチで管理されるため，これらのバージョン間にはアルファ・ベータ・RC リリースのみが存在し分岐も発生していない．表 1 に，各バージョンのファイル数と総行数を示す．

結果

抽出した派生関係がバージョン履歴と一致した本数は表 2 のようになった．12 本すべての派生関係が一致した距離関数は，C_s,C_t,${C_s}^{sim}$,${C_t}^{sim}$ であった．ファイル間のトークンの差分について定めた類似関数 ${C_s}^{diff}$,${C_t}^{diff}$,$\overline{{C_s}^{diff}}$,$\overline{{C_t}^{diff}}$ を用いて構築した派生関係木は，バー

[9] PostgreSQL: The world's most advanced open source database, `http://www.postgresql.org/`

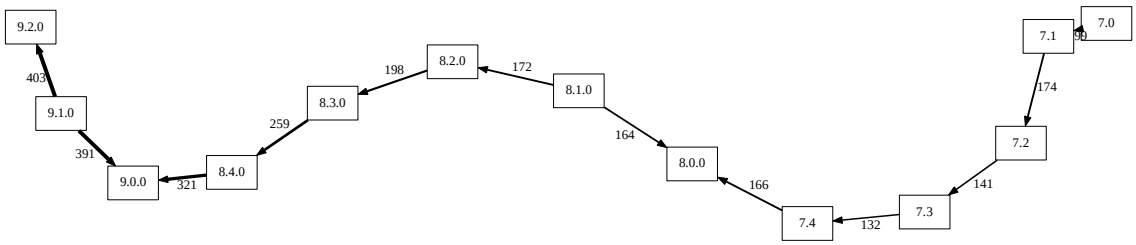


図 8: PostgreSQL-major，類似度閾値 0.9，距離関数 C_t，派生方向 d_{ALL}

ジョン履歴とまったく一致しなかったか，1 本を除き一致しなかった．

また，12 本すべての派生関係が一致した距離関数 C_t の派生関係木に対し方向付けを行った．バージョン履歴と方向が一致した本数は表 3 のようになった．最も一致した辺の多かった類似度閾値 0.9，距離関数 C_t，派生方向 d_{ALL} の場合を図 8 に掲載した．

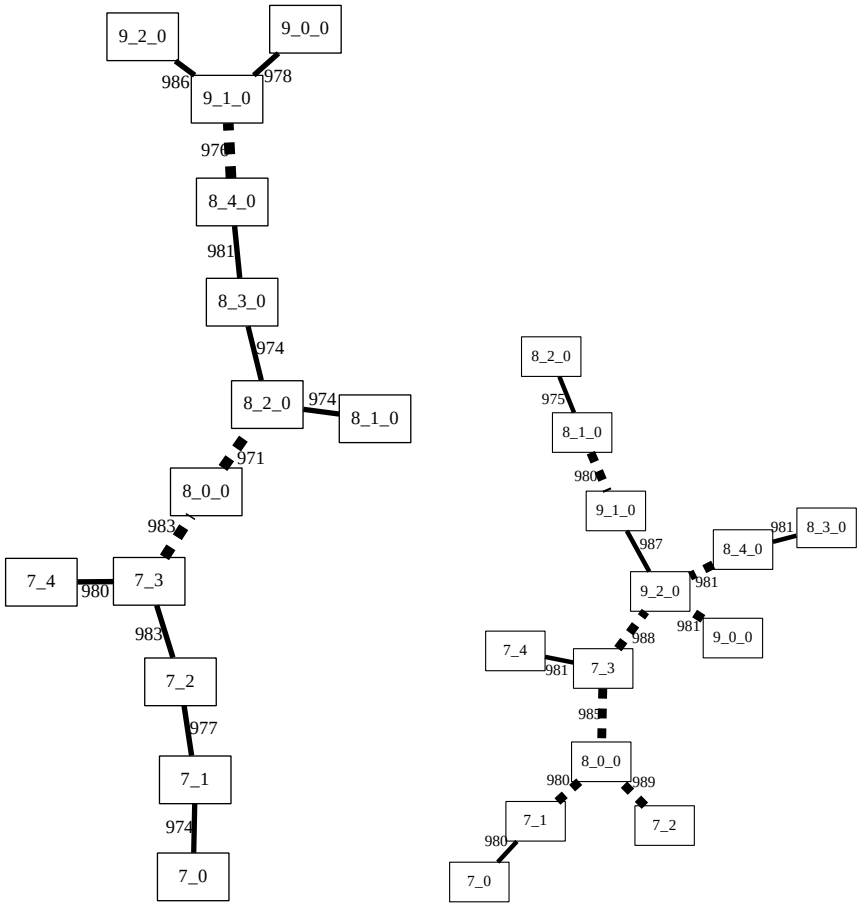
考察

ファイル間のトークンの差分について定めた類似関数 ${C_s}^{diff}$,${C_t}^{diff}$,$\overline{{C_s}^{diff}}$,$\overline{{C_t}^{diff}}$ を用いて構築した派生関係木は，バージョン履歴と一致しなかった．今回差分はトークンの追加と削除の合計数としている．

ファイル間の平均類似度 $\overline{{C_s}^{sim}}$,$\overline{{C_t}^{sim}}$ の両関数を用いて構築した派生関係木については，半数ほどの辺が元のバージョン履歴と一致した．閾値 0.7 の時の，類似関数 $\overline{{C_s}^{sim}}$(図 9(a)) と $\overline{{C_t}^{sim}}$(図 9(b)) で構築した派生関係木を例にとり，一致しなかった残りの辺について詳しく述べる．図は，バージョン履歴と一致した辺を実線で，一致しなかった辺を破線でそれぞれ示している．また，辺の横に距離関数の値を付している．$\overline{{C_s}^{sim}}$ を用いて構築した派生関係木では，バージョン履歴と一致しなかった辺は履歴上でのバージョンを 1 つ飛ばしたソフトウェア間を結んでいた．$\overline{{C_t}^{sim}}$ を用いて構築した派生関係木は，7.3 と 9.2.0 をつなぐ辺が見られるなど，元の履歴からの逸脱が大きかった．

類似ファイル集合を生成する際の閾値については，実験結果に大きな影響がなかった．また，ファイル単位での派生関係を計算しても結果が変化しなかった．どちらの事象も，メジャーリリース間での比較であったのでソフトウェア間の差異が比較的大きめの集合だったことが影響していると考えられる．

ソフトウェアの派生方向に関しては，ファイル単位の派生方向を考えるよりもソフトウェア全体のトークンの増減を調べたほうが，バージョン履歴と方向が一致する数が多かった．派生方向が一致しなかったソフトウェア間では，識別子名の変更やコードの整理などで類似ファイル間でのソースコードの量が減少しているものが見られた．全体としてはソースファ



(a) $\overline{C_s{}^{sim}}$ (b) $\overline{C_t{}^{sim}}$

図 9: PostgreSQL-major, 類似度閾値 0.7，派生方向ラベルなし

イルの数は増えているが，今回用いた計測方法では新しいバージョンで追加されたファイル，つまり前のバージョンと比較したときに類似ファイルが見つからないものを計測対象に含めていないためであると推測される．

表 1: PostgreSQL 各バージョンのファイル数と総行数

PostgreSQL	C #files	C LOC
7.0	488	175758
7.1	501	191495
7.2	511	210845
7.3	517	210361
7.4	559	247829
8.0.0	582	277200
8.1.0	605	297158
8.2.0	666	324395
8.3.0	762	392120
8.4.0	790	421390
9.0.0	824	444446
9.1.0	849	472733
9.2.0	879	497397
計	8533	4163127

表 2: PostgreSQL-major，13 ソフトウェア間 12 本の辺のバージョン履歴との一致数

閾値	C_s / C_t	${C_s}^{diff}$ / ${C_t}^{diff}$	$\overline{{C_s}^{diff}}$ / $\overline{{C_t}^{diff}}$	${C_s}^{sim}$ / ${C_t}^{sim}$	$\overline{{C_s}^{sim}}$ / $\overline{{C_t}^{sim}}$
0.7	12 / 12	0 / 0	0 / 0	12 / 12	8 / 4
0.8	12 / 12	0 / 0	1 / 0	12 / 12	7 / 6
0.9	12 / 12	0 / 0	0 / 0	12 / 12	6 / 5
1.0	12 / 12	- / -	- / -	- / -	- / -

表 3: PostgreSQL-major，距離関数 C_t で派生関係木の形が一致した 12 本の辺の派生方向の一致，逆，不明

閾値	d_{ALL}	d_{SIZE}	d_{INC}
0.7	10, 2, 0	5, 7, 0	7, 5, 0
0.8	10, 2, 0	5, 7, 0	8, 4, 0
0.9	10, 2, 0	4, 8, 0	6, 6, 0

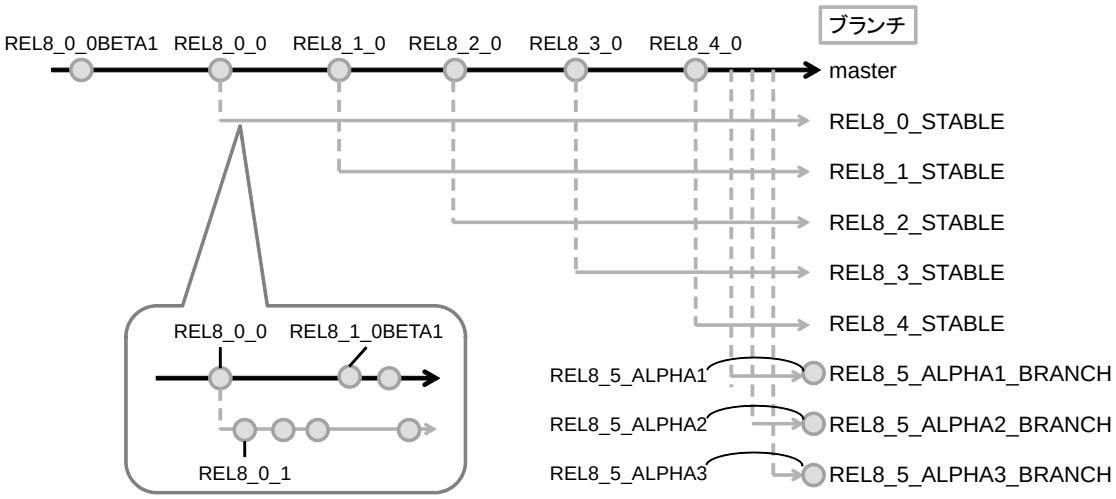


図 10: PostgreSQL8 系の分岐

5.3 PostgreSQL-8-ALL: 組織内で開発が分岐したソフトウェアプロダクト集合

PostgreSQL はあるメジャーバージョンがリリースされた後も一定期間，既存のバージョンに対するマイナーリリースが行われている．例えば 8.0 の 1 回目のマイナーリリースの際には，同時に 7.2.7, 7.3.9, 7.4.7 の 3 つのマイナーリリースが行われた．このような開発形態をとっているため，PostgreSQL ではメジャーバージョンをリリースした後にブランチを分岐させて保守を続けている．

git リポジトリのログから得られた，PostgreSQL のバージョン 8 系の開発の分岐の履歴を図 10 に示す．各メジャーバージョンは master ブランチにおいてベータ版 (BETA)，リリース候補版 (RC) の開発が行われ，最初のリリースが行われる．その後，マイナーリリースを行うブランチが分岐し，master ブランチでは次のベータ版以降の開発が継続される．

PostgreSQL-8-ALL は，PostgreSQL の git リポジトリから，“REL8” で始まるタグの付与されたリビジョンを集めたデータセットである．計 144 のリビジョンがあり，拡張子.c のファイルが 96448 ファイル 48478395 行あった．開発は大きく分けて 8.0 系，8.1 系，8.2 系，8.3 系，8.4 系に分かれている．また，8.5 のアルファリリースが 3 回あり，これは後に 9.0 系にバージョン番号が変更された．これらのリリースは 8.4 系までとは違い，すべて 8.4.0 から直接分岐して別々のブランチとなっている．

5.2 節で示した結果に基づき，類似関数は C_s,C_t,${C_s}^{sim}$,${C_t}^{sim}$ の 4 つを用いた．

結果

表 4 より，抽出した派生関係は多くがバージョン管理システムの履歴と一致していることがわかる．また，最も一致した辺の本数が多かった類似度閾値 0.9 のときの派生方向についても，表 5 に示した通り基準によって差はあるが約 9 割の辺で正しいことがわかる．最も一致数の多かった類似度閾値 0.9，類似関数 C_t，派生方向 d_{SIZE} の派生関係木は図 11 に示した．

考察

次のバージョンの初めのベータリリースへ向かう辺はすべての場合でバージョン履歴と一致しなかった．たとえば 8.0 系から 8.1_BETA1 へは抽出した派生関係では 8.0.4 から辺が引かれたが，バージョン履歴のブランチを見ると 8.0.1 は 8.0 系のブランチに分かれており，master ブランチ上では 8.0.0 の次に 8.1_BETA1 のリリースが存在する．

8.0 系と 8.1 系ベータリリース前後の時系列を図 12 に，8.1 系から 8.2 系を図 13 にそれぞれ示す．これらの図より，抽出した派生関係ではベータリリースの直前・直後のリリースとつながっていることがわかる．バージョン 8.1.4 のリリースまでに保守ブランチに対し行われたコミットの数は，283 件，そのうち保守ブランチと master ブランチで同様の変更が行われたコミットの数（バージョン管理システムにおいて，同じ日付に同一のコメントでコミットが行われているもの）は 255 件であった．つまり，master ブランチとリリース版の保守ブランチでほぼ同様の変更が行われてきたため，開発時期が近いバージョン同士が辺で結ばれたものと推測される．ベータリリースには新機能も含まれるため，以降のベータリリース 2,3,... は分岐した後のベータリリースとつながる．8.5 系のアルファリリース 3 バージョンについても，同様の原因によりバージョン履歴と一致しなかったと考えられる．

また，すべての結果において 8.0.15 から 8.0.17 にかけての派生関係はバージョン履歴の順番と合致しなかった．一方，同時期にリリースされた 8.1.11 から 8.1.13 の派生関係は類似度の閾値が 0.7 の時を除き 1 直線に抽出できている．調査の結果，8.0.16 から 8.0.17 にかけ

表 4: PostgreSQL-8-ALL，144 ソフトウェア間 143 本の辺のバージョン履歴との一致数

閾値	C_s / C_t	${C_s}^{sim}$ / ${C_t}^{sim}$
0.7	132 / 133	133 / 135
0.8	134 / 135	135 / 135
0.9	136 / 136	136 / 136
1.0	135 / 135	-

ての変更行数が少なく (git diff の出力行数だけ見ても 351 行)，8.0.15 から 8.0.17 と 8.0.16 から 8.0.17 で変更されたファイル数を比べても，ともに 306 ファイルであることがわかった．辺の重みが同じだったため，8.0.15 からどちらのバージョンへ辺がひかれるかが定まらなかったとわかる．8.1.11 から 8.1.13 にかけて変更されたファイル数は 8.1.11 から 8.1.12 よりも多かったため，こちらのブランチではこの問題は起こらなかった．

ファイル間で派生関係をとった場合ととっていない場合では類似度閾値を 0.8 以下にしたとき，ファイル間派生関係をとった場合の方がわずかに良い結果を示した．また類似関係にあるファイルの個数のみを調べるのか，類似度で重みづけをするのかでは，類似度閾値を 0.7 以下にしたとき，重みづけがあったほうがわずかに良い結果を示した．いずれの場合もわずかな差異であるため，計算のコストも考慮すると効果をあげているとは言い難い．

表 5: PostgreSQL-8-ALL， 類似度閾値 0.9, 距離関数 C_t で派生関係木の形が一致した 136 本の辺の派生方向の一致，逆，不明

閾値	d_{ALL}	d_{SIZE}	d_{INC}
0.9	123/13/0	133/3/2	119/15/11

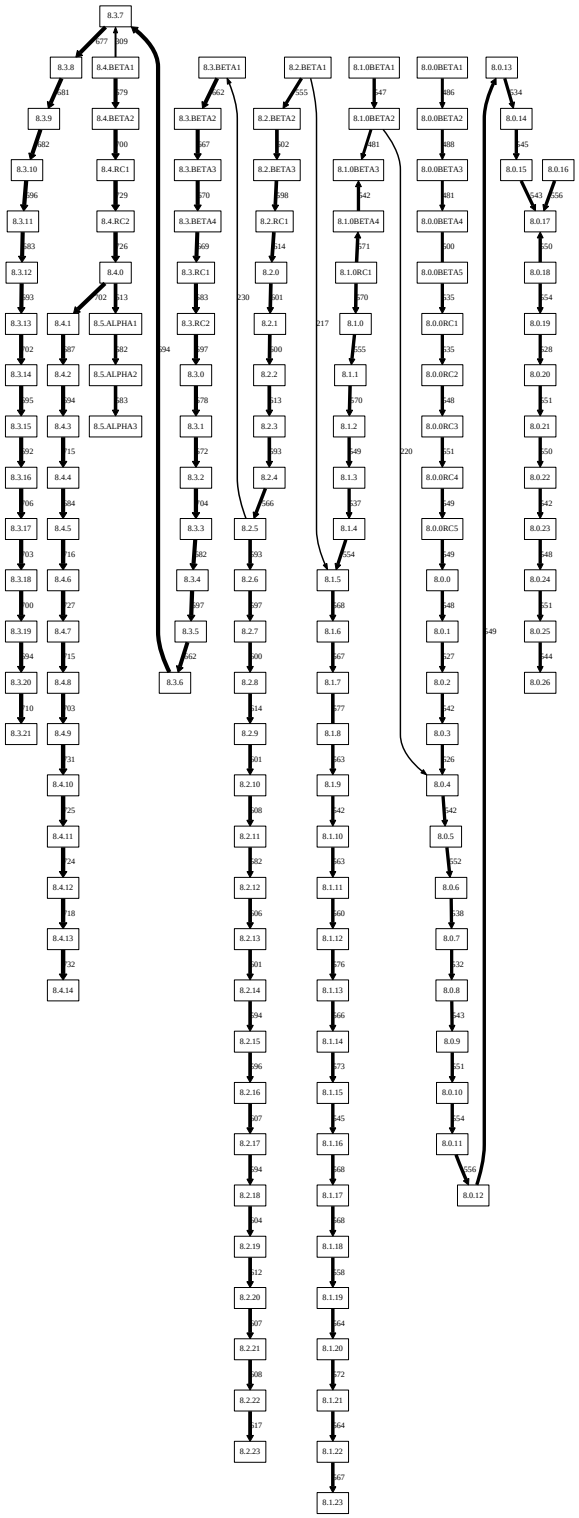


図 11: PostgreSQL-8-ALL, 類似度閾値 0.9, 距離関数 C_t, 派生方向 d_{SIZE}

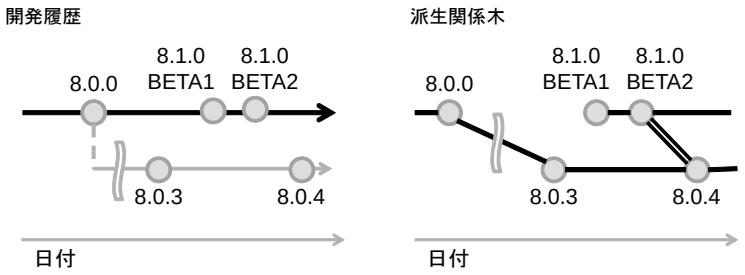


図 12: PostgreSQL8.0 系から 8.1 系への分岐

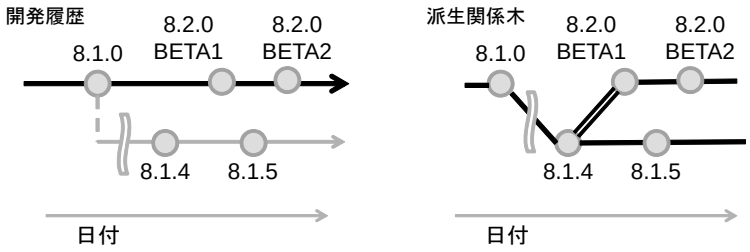


図 13: PostgreSQL8.1 系から 8.2 系への分岐

5.4 PostgreSQL-8-latest: ソフトウェア製品ファミリのうち新しいいくつかの製品しか残っていない場合

過去のソフトウェア製品はもう残っていないか入手困難であるが，最近のソフトウェア製品が入手できた場合に本手法がどのような結果を示すかをシミュレーションした．分岐したタイミングのソフトウェア製品は入手できていないような状況である．

PostgreSQL-8 のうち，各ブランチの最新 7 バージョンずつを取得したデータセットを用いた．ただし 8.5 のアルファリリースは 3 回しか行われていないため，この系列のみ 3 バージョンを取得した．よって合計 38 バージョンに対して解析を行った．

なお，このデータセットにはバージョン系列間において分岐が発生したとされるバージョンが含まれていない．バージョン履歴においては各バージョン系列は前のバージョン系列の最初のメジャーリリースから分岐しているため，各バージョン系列のもっとも古いソフトウェア同士が接続された状態を正解とする．

結果

例として，これまでの実験でバージョン履歴と一致した割合が大きかった類似度閾値 0.9, 類似関数 C_t で構築した派生関係木を挙げる．図 14 に，類似度閾値 0.9，類似関数 C_t，派生方向 d_{SIZE} で構築した派生関係木を示す．その他の派生方向に関しても表 6 にまとめた．

派生関係木の形については，派生関係木の 37 本の辺のうち，バージョン履歴との一致は 30 本であった．PostgreSQL-8-ALL でも一致しなかった 8.5 系アルファリリースへの辺のほか，バージョン系列間の辺もバージョン履歴と一致しなかった．

すべてのバージョン系列内において，バージョンアップされた順番に木の上にノードが並んでいる．バージョン系列間の距離はバージョン系列内の距離と比べ大きな値を示した．

考察

同じバージョン系列内でバージョンアップされた順番に並ぶことは，PostgreSQL-8-ALL と同様である．派生方向がバージョン履歴と一致しなかった辺も PostgreSQL-8-ALL と同様であった．

表 6: PostgreSQL-8-latest， 類似度閾値 0.9, 距離関数 C_t で派生関係木の形が一致した 30 本の辺の派生方向の一致，逆，不明

閾値	d_{ALL}	d_{SIZE}	d_{INC}
0.9	29/1/0	30/0/0	28/1/1

PostgreSQL-8-ALL と異なる点は，分岐点となったソフトウェア製品が含まれていないことである．構築した派生関係木では，バージョン系列間の接続は同時にリリースされたバージョンの間で発生していた．これらのバージョン系列は分岐後は主に機能追加ではなく保守を行うためのブランチとなっているため，同時期のソフトウェアは同様な修正を施された共通する部品が多いソフトウェアであるからだと推測できる．しかし，どのタイミングのソフトウェアが接続されるかについては，前のバージョン系列の最終版と結合している辺が多いものの決まった関係性を見いだせない．

このような理由もあり，図 14 は一見バージョン系列間の境界がわからない．しかし，バージョン系列間にひかれた辺はバージョン系列内での辺に比べ距離が大きい．このことから，ソフトウェア間の距離のちがいを利用して，バージョン系列を切り分けることができる．またこれは，本手法が複数の別個のソフトウェア製品ファミリを入力に与えたとしても，ソフトウェア間の距離に閾値を設けることでソフトウェア製品ファミリ間の区別が可能であることを示している．

なお，表 6 中には記載されていないバージョン系列間の派生方向については，6 系列間の 5 本中 d_{ALL} が 4 本，d_{SIZE} が 3 本，d_{INC} は 0 本がバージョン履歴と一致した．

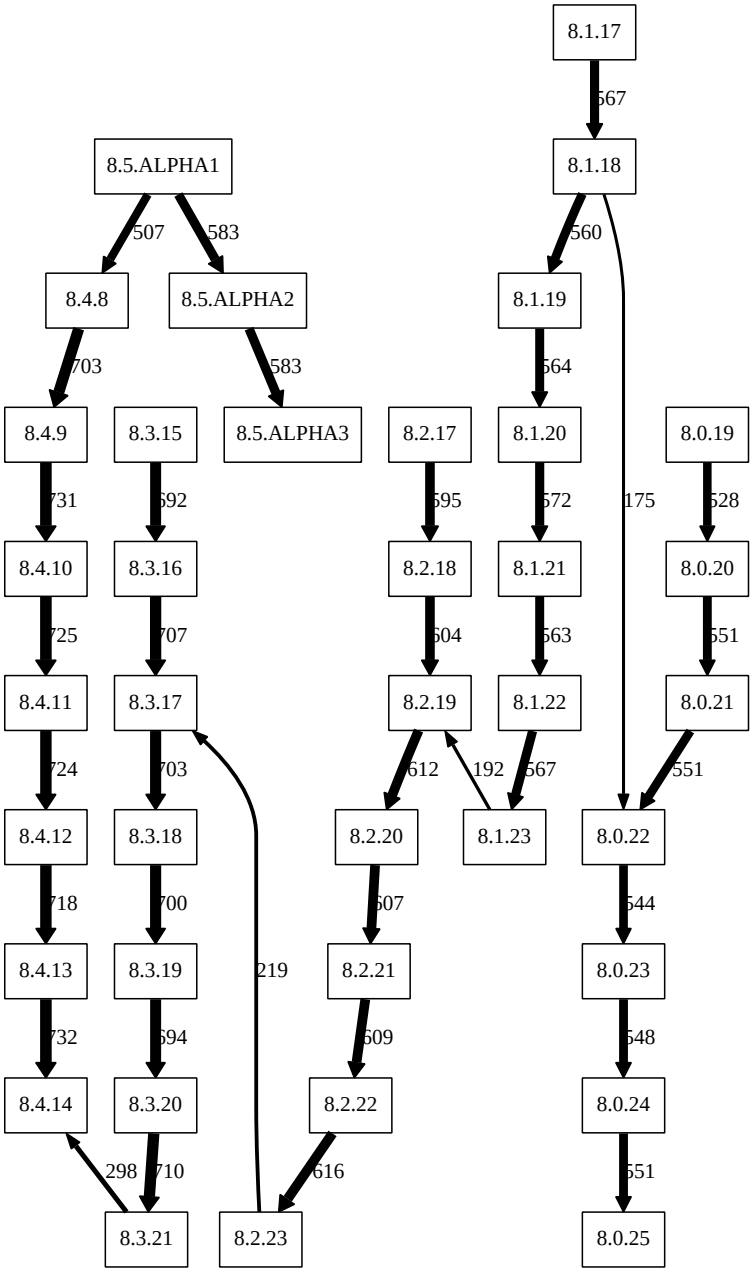


図 14: PostgreSQL-8-latest，類似度閾値 0.9, 距離関数 C_t，派生方向 d_{SIZE}

5.5 PostgreSQL-8-annually: ソフトウェア製品ファミリのうち途中が一部欠落している場合

過去に作られたソフトウェア製品ファミリのうち，その一部が入手できず連続したバージョンが得られなかった場合に本手法がどのような結果を示すかをシミュレーションした．

このデータセットは，PostgreSQL-8 のうち，毎年 9 月前後に各バージョン系列で同日にリリースが行われたものを集めた．同時にリリースされた版同士は同様なバグ修正を含む．2005 年 10 月 3 日から 2012 年 9 月 19 日まで，8 つのタイミングでリリースを取得し全 26 バージョンを解析した．

このデータセットにもバージョン系列間において分岐が発生したとされるバージョンが含まれていないため，PostgreSQL-8-latest と同様各バージョン系列のもっとも古いソフトウェア同士が接続された状態を正解とする．

結果

例として類似度の閾値 0.9，類似関数 C_t，派生方向 d_{SIZE} で構築した派生関係木を図 15 に示す．また，その他の派生方向に関しても表 7 にまとめた．

木の形は 25 本中 22 本がバージョン履歴と一致した．すべてのバージョン系列内において，バージョンアップされた順番に木の上にノードが並んでいる．バージョン系列間の距離はバージョン系列内の距離と比べ大きな値を示した．

考察

ソフトウェア製品ファミリのうち一部が欠落していても，本手法ではある程度ソフトウェア製品間の関係性を明らかにできている．

分岐のタイミングに関しては正確ではない，辺の重みで製品ブランチを区別できるなど，木の性質は PostgreSQL-8-latest と近いものとなっている．

派生方向については，すべての派生方向が一致するという結果になった．これまで実験したデータセットとの違いを考えると，以下の 2 点が原因として考えられる．

表 7: PostgreSQL-8-annually， 類似度閾値 0.9, 距離関数 C_t で派生関係木の形が一致した 22 本の辺の派生方向の一致，逆，不明

閾値	d_{ALL}	d_{SIZE}	d_{INC}
0.9	22/0/0	22/0/0	22/0/0

- PostgreSQL-8-major と比べて，解析したバージョン間のリリースに間隔がある点は共通している．しかし，保守ブランチ内での比較では機能追加が少ないため，ソースコードの大きな改変がなくメジャーリリース間を比較したときのような誤検出が起こる要因がなかった．

- PostgreSQL-8-ALL，PostgreSQL-8-latest は連続したリリースを比較したため変更点の少ないリリース付近で誤検出があった．このデータセットは 1 年おきのリリースを比較しているためバージョン間で変更の少ないものがなかった．

なお，バージョン系列間の派生方向については，5 系列間の 4 本中 d_{ALL} が 4 本，d_{SIZE} が 2 本，d_{INC} は 1 本がバージョン履歴と一致した．これも PostgreSQL-8-latest と同様の傾向を示している．

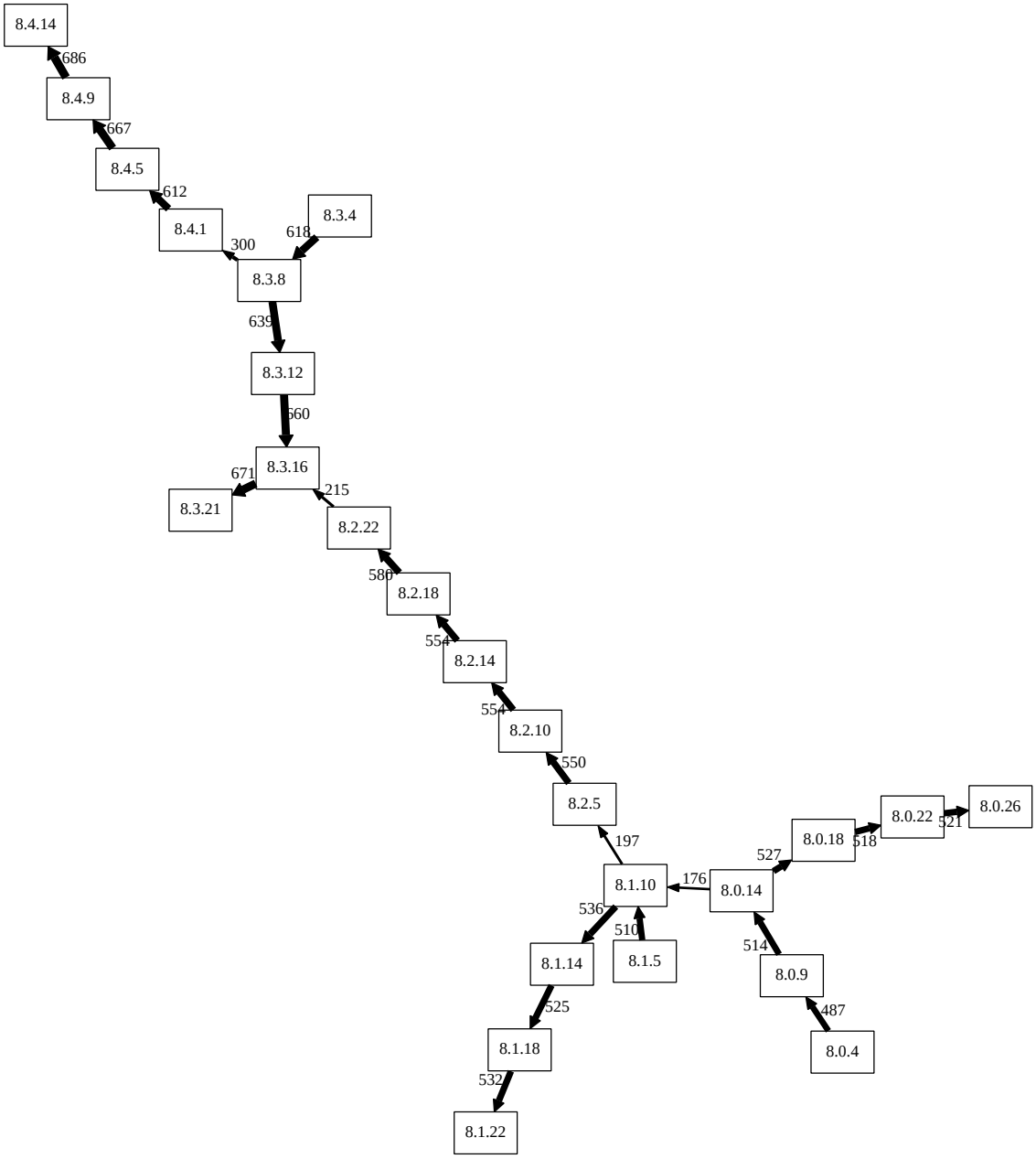


図 15: PostgreSQL-8-annually, 類似度閾値 0.9, 距離関数 C_t, ラベルづけ d_{SIZE}

5.6 FFmpeg: プロジェクトが 2 つに分岐した場合

ここまでのデータセットは，同一のソフトウェアリポジトリから取得したものであった．次のデータセットは，プロジェクト自体が分裂した場合を扱う．一部互いの変更点を取り込みあっているものの，別個に開発が進んでいるソフトウェア集合に対して実験を行う．

FFmpeg は 2.2.3 項でも紹介した動画・音声などのマルチメディアデータを扱うことのできるライブラリ・プログラム群である．FFmpeg と Libav に分岐し，どちらも git でバージョン管理されている．

今回は分岐以前から開発されていた FFmpeg バージョン 0.5 から 0.5.3，分岐後の FFmpeg バージョン 0.5.5 から 0.5.10，Libav バージョン 0.5.4 から 0.5.9 の計 16 バージョンを対象に解析した．

分岐の発生は FFmpeg のバージョン 0.5.3 のリリース後である．解析対象のリリース日時一覧を表 8 に示す．f から始まるバージョンが分裂前，F から始まるバージョンが FFmpeg（分裂後）L から始まるバージョンが Libav である．

結果

類似度閾値 0.9，類似関数 C_t，派生方向 d_{ALL} で構築した派生関係木を図 16 に示す．f から始まるノードが分裂前，F から始まるノードが FFmpeg（分裂後）L から始まるノードが Libav である．派生方向を変えた結果は表 9 に示した．

木の形は 15 本中 13 本がバージョン履歴と一致した．構築した派生関係木からは，大きく F と L の 2 つの分岐が発生していることがわかる．分岐の箇所については，プロジェクトが分岐した 0.5.3 前後ではなく，同時期にリリースがあった FFmpeg0.5.5 と Libav0.5.5 前後にある．また，FFmpeg の 0.5.1 もバージョン順に接続されていない．

考察

FFmpeg と Libav は分岐後も互いの成果を取り込みあっているため，分岐の始点が FFmpeg0.5.3 ではなく後にリリースされた Libav0.5.5 になったと推測される．厳密な分岐は再現できなかったものの，2 つに分かれたあとの最も端の頂点にそれぞれのプロジェクトの最新版が出現しているため，目的としている比較対象のソフトウェア選択の補助には役立つと考えられる．

FFmpeg0.5.1 がバージョン順に接続されなかった理由は，FFmpeg0.5 から見て FFmpeg0.5.1 と FFmpeg0.5.2 で変更されたファイル数に差がなかったためであった．変更がわずかなソフトウェア同士が含まれている場合，距離関数が同じになったり，誤差の影響が大きくなる可能性があることがわかる．

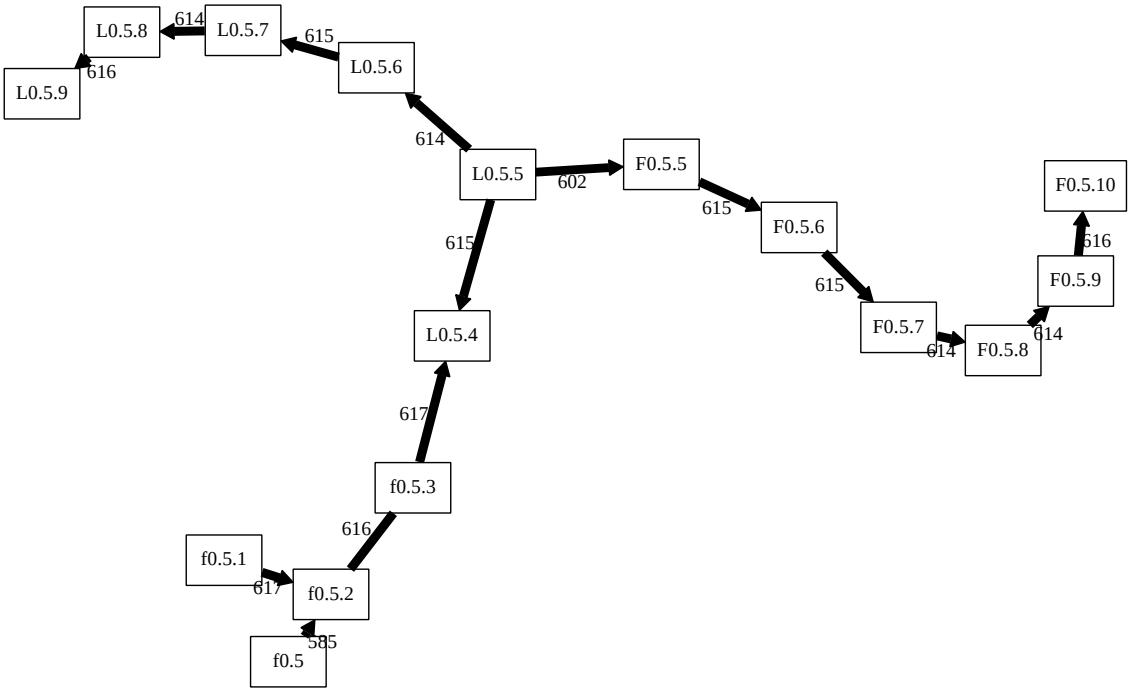


図 16: FFmpeg, 類似度閾値 0.9, 距離関数 C_t，派生方向 d_{ALL}

表 8: FFmpeg と Libav のリリース日時

2009-03-08 22:13:48	f 0.5
2010-03-02 16:03:06	f 0.5.1
2010-05-24 21:58:47	f 0.5.2
2010-10-18 19:43:55	f 0.5.3
2011-03-13 BRANCHED	
2011-03-17 13:10:27	F 0.5.4
2011-11-05 12:57:22	L 0.5.5
2011-11-06 20:57:55	F 0.5.5
2011-11-21 22:22:04	F 0.5.6
2011-12-25 10:18:18	L 0.5.6
2011-12-25 21:43:56	F 0.5.7
2012-01-10 22:22:05	L 0.5.7
2012-01-12 22:19:09	F 0.5.8
2012-05-10 20:40:38	L 0.5.8
2012-05-11 22:39:50	F 0.5.9
2012-06-09 12:13:27	L 0.5.9
2012-06-09 22:18:07	F 0.5.10

表 9: FFmpeg, 類似度閾値 0.9, 距離関数 C_t で派生関係木の形が一致した 13 本の辺の派生方向の一致，逆，不明

方向付け	d_{ALL}	d_{SIZE}	d_{INC}
0.9	11, 1, 1	11, 0, 2	9, 0, 4

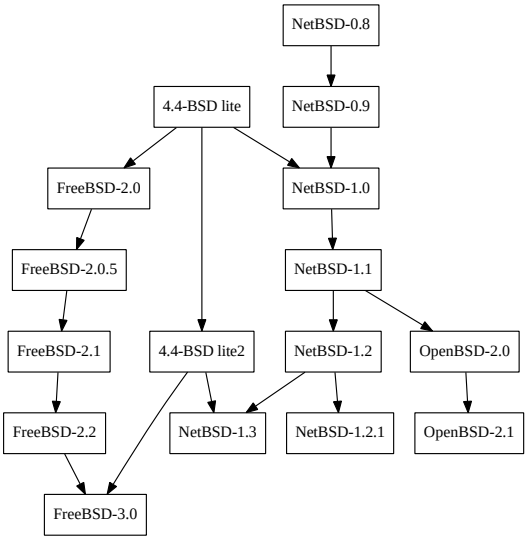


図 17: BSD 系 OS の家系図

5.7 *-BSD: プロジェクトが 3 つ以上に分岐した場合

2 つのプロジェクトが 3 つ以上に分岐した例として，4.4BSD Lite, 4.4BSD Lite2 およびその派生 OS の FreeBSD,NetBSD,OpenBSD を取り上げる．これらは別個のプロジェクトとして独立しているが,「家系図」が公開されており[10]，その進化の過程を知ることができる．この図からは，特に開発初期において単にプロジェクトが分岐しただけでなく複雑に絡み合っていることがわかる．

解析対象のデータセットとしては，4.4BSD Lite, 4.4BSD Lite2, FreeBSD から 5 バージョン，NetBSD から 7 バージョン，OpenBSD から 2 バージョンを選択した．これらの家系図上の関係は図 17 のようになる．

このデータセットの別な特徴として，家系図における閉路の存在が挙げられる．4.4BSD Lite から FreeBSD-3.0 までは，2 つの経路が存在しており，このような履歴をもつソフトウェア集合に対して本手法がどのような派生関係木を示すかを確認する．

結果

類似度閾値 0.9，類似関数 C_t，派生方向 d_{SIZE} で構築した派生関係木を図 18 に示す．木の形は 15 本中 12 本が家系図と一致した．また派生方向を変えた結果は表 10 に示した．

4.4BSD Lite から FreeBSD-3.0 にかけては家系図の順序通りの派生関係が構築できてい

[10] http://www.freebsd.org/cgi/cvsweb.cgi/src/share/misc/bsd-family-tree?rev=1.147.2.2;content-type=text\%2Fplain

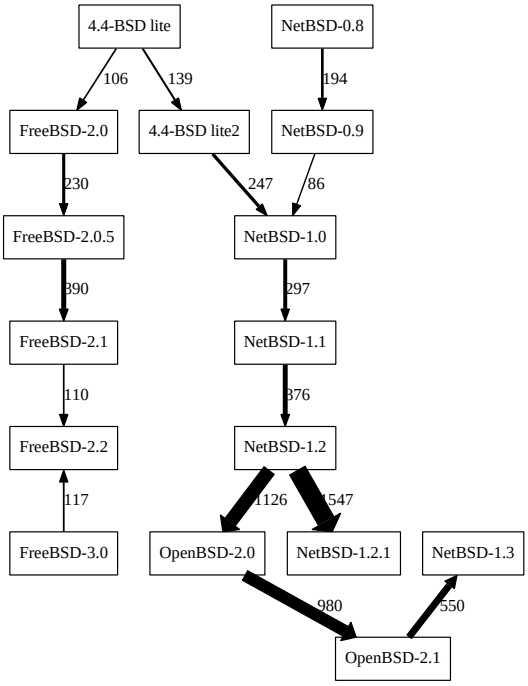


図 18: *-BSD, 類似度閾値 0.9, 距離関数 C_t，派生方向 d_{SIZE}

る．家系図と一致しなかった派生関係木の辺は，4.4BSD Lite2 から NetBSD-1.0（家系図は 4.4BSD Lite から），NetBSD-1.2 から OpenBSD-2.0（家系図は NetBSD-1.1 から），OpenBBSD-2.1 から NetBSD-1.3（家系図は NetBSD-1.2 から）の 3 本だった．また，これら以外で家系図にあって派生関係木にない辺としては，4.4BSD Lite2 から FreeBSD-3.0, 4.4BSD Lite2 から NetBSD-1.3 の 2 本があった．

考察

構築した派生関係木のうち FreeBSD の派生については，派生方向が一致しない辺もあったが形は家系図と一致した．4.4BSD Lite2 から FreeBSD-3.0 への派生が得られなかったのは，本手法が閉路を許さないためである．類似度閾値 0.9 での C_t(4.4BSD Lite, FreeBSD-3.0) は

表 10: *-BSD, 類似度閾値 0.9, 距離関数 C_t で派生関係木の形が一致した 12 本の辺の派生方向の一致，逆，不明

方向付け	d_{ALL}	d_{SIZE}	d_{INC}
0.9	9,3,0	11,1,0	10,2,0

11 であり，C_t(FreeBSD-2.2, FreeBSD-3.0) の 117 と比べてかなり低い．このとから 4.4BSD Lite2 から FreeBSD-3.0 へは家系図上のつながりはあるものの「影響を受けた」程度の関係性であると推測できる．

4.4BSD Lite および 4.4-BSD Lite2 と，NetBSD 系の間で派生関係木と家系図が一致しないあるいは派生関係木に辺が存在しないものが 2 本あった．提案手法はソースファイル間の類似度を計算し類似ファイル集合を抽出しているため，このデータセットのように単純に開発が分岐しただけでなく大きな改変が加えられているとバージョン履歴との一致が少なくなることが予想される．

5.8 全体を通しての考察

提案手法で構築したソフトウェア間の派生関係木は，派生関係にあるファイルの個数をもとに構築すると全データセットを平均して約 88%が元のバージョン履歴に合致することがわかった．今回の実験対象は C 言語で実装されたソフトウェアであり，ある程度機能ごとにファイルが分割されている．そのため，追加・変更された機能の量が追加・変更されたファイル数に影響を及ぼしたと推測される．他の言語においても，実際にどのような機能を表現しているかを特定せずとも，ソースコードをファイルやメソッドなど一定の単位で区切って提案手法を適用することにより，元のバージョン履歴に近い派生関係木が構築できると期待できる．

PostgreSQL-8-ALL や FFmpeg で見られたように，微小な変更のみが加えられたバージョンが解析対象に混ざっていると，その前後の派生関係がバージョン履歴と一致しない傾向にある．ソフトウェア間の距離が非常に高いソフトウェア同士はあらかじめ 1 つにまとめて計測する，このような現象が起きないような補正メトリクスを重みに追加する，などの対策が挙げられる．しかし，誤検出であっても元のバージョン履歴と近いソフトウェアと接続されていることが多く，派生関係木を調査すればこの影響を取り除くことは難しくないと考えている．

ソフトウェア間の派生方向についても，バージョン履歴と約 86%程度の一致を見た．方向付けについては， d_{SIZE} がおおむね高い一致率を示したが， PostgreSQL-major では d_{ALL} が最も一致率が高かった．

派生方向の誤検出によっては，一直線に進化している途中のバージョンが派生の分岐点に見えることもある．本来の分岐点と離れたバージョンに誤検出が生じた場合，木の形の誤検出よりも調査の手間が増える可能性がある．

閉路を許さないため，BSD 系 OS に対する解析では家系図の辺が再現できないものがあった．特にソフトウェアの統合については，機能が多く違う 2 つのソフトウェアを統合すると統合前後でソフトウェアの類似度が下がるため，本手法での検出は難しいといえる．

6 関連研究

ソフトウェア間の類似度を定義しようという試みはこれまでにも行われている．Yamamoto らは，2 つのソフトウェアシステムを比較するために，対応する行の数に基づく類似度メトリクス S_{line} を定義し，UNIX 系 OS のソースコードについてシステム間の類似度を測定した [19]．また，類似度メトリクスからクラスタ分析を行うことで樹状図を作成し，OS の分類を派生通りにあらわしているという結果を得た．樹状図で得られるものは「どのソフトウェアとどのソフトウェアが近いのか」という情報である．提案手法では派生の関係をグラフで表現することで，どのソフトウェア間で機能の分岐が発生したのか，また各分岐の中ではどのソフトウェアが最も変更が多くくわえられた版なのかを見ることができる．

多くのソフトウェアを比較することや，またその結果を可視化する難しさは Duszynski らも指摘しており [2]，彼らはバリアント解析の結果を様々な見せ方で表現した．

また，ソフトウェアプロダクトライン構築に重要な技術として Feature Location（機能捜索）技術がある．Feature Location 技術は，ソフトウェアの機能がソースコード上で実装されている個所を捜索する技術である．ソフトウェアプロダクト集合に含まれる機能を明らかにすることで，コア資産と機能部品の抽出も容易になると考えられている．Haslinger らは，ソフトウェア製品ファミリから機能モデルを抽出するために，まず Feature Sets という機能表を作り，そこから機能モデルを作成した [4]．このような技術に対して入力として与えるソフトウェアが非常に多くなるような場合，本研究のように解析対象とするソフトウェアを絞り込む技術は有用であると考える．

ソフトウェアプロダクトラインの構築を支援する試みとして Marco らの提案した CIDE は，ソフトウェア製品ファミリに含まれる機能のうち追加・削除可能な機能と色をあらかじめ指定すると，指定した機能に対応するコードを IDE 上で色付けする [18]．色付けには開発者が機能を指定する必要がある．また特定の色が表示されない状態にすれば，その色に対応した特定の機能を廃止したソフトウェアを開発することができる．開発者が機能を知っていることが前提であるため，ソースコードのみを入力して扱う本研究とは別の観点からの技法といえる．

Lavoie らはコードクローン検出技術を用いて，リリースバージョン間のファイルの潜在的な移動を検出した [9]．潜在的な移動とは，バージョン間のファイルの移動のうち，バージョン管理システムのリポジトリに記録されていないもののことである．この技術により，ソフトウェアシステムのファイル構造を復元することができる．Lavoie らがファイル単位での構造を復元したことに対し，本研究はソフトウェア間の構造を復元した点が大きく異なる．

7　まとめ

本研究では，既存のソフトウェアプロダクト集合に対し，ソフトウェアプロダクトラインを構築する際のソフトウェアの選択を補佐するために派生関係木を構築した．派生関係木は，ソフトウェア間の距離に基づいた最小全域木に，派生方向のラベルづけを行ったものである．

オープンソースソフトウェアに対する適用実験では，様々な状況を想定したデータセットに対し，類似度の高いファイルの個数に着目することでバージョン履歴と派生関係木が近いものになるという結果が得られた．また派生方向についても，ファイルサイズの増減を基準とすることで多くが開発履歴と一致した．派生関係木を用いて，バージョン履歴の失われたソフトウェアプロダクト集合に対し，それに代わるソフトウェア間の関係を提供することができることを確認した．

今後の課題としては，スケーラビリティの向上と産業向けシステムのソースコードへの実験対象の拡大，派生関係木を利用したソフトウェアプロダクトライン構築手法の提案が挙げられる．

本手法は入力されたすべてのファイル間で類似度計算を行っているため，計算機に対する負担が大きくなっている．現在行っている対策として，ファイルサイズや出現する字句による類似度が低いファイルのフィルタリングがある．実験結果では，類似度の閾値が 1 の時でも良い結果を示していたため，ファイル単位のクローンを検出することでより高速な解析を行うことも考えられる．

また，産業向けシステムでも本手法が有効であるかを確認することは重要な課題である．ソフトウェアプロダクトラインエンジニアリングは，ソフトウェア開発を行っている企業にとって大きな関心事である．言語やソースコードの書き方，管理方法もオープンソースソフトウェアとは違うことも多いため，提案手法がオープンソースソフトウェアと同じように有効であるのか，あるいはどのように調整する必要があるのかを調査したい．

その後，ソフトウェアプロダクトライン構築手法に実際に派生関係木を利用することで，本研究の有用性を示したいと考えている．

謝辞

本研究において，常に適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上 克郎 教授に心より深く感謝いたします.

本研究において，随時適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下 誠 准教授に深く感謝いたします.

本研究において，逐次適切な御指導及び御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾 隆 助教に深く感謝いたします.

本研究において，適時適切な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 眞鍋 雄貴 特任助教に深く感謝いたします.

本論文の執筆にあたり，様々な御指導及び御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 伊達 浩典 氏に深く感謝いたします.

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします.

参考文献

[1] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[2] S. Duszynski, J. Knodel, and M. Becker. Analyzing the source code of multiple software variants for reuse potential. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pp. 303 –307, 2011.

[3] D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience, John Wiley & Sons, Ltd*, Vol. 33, pp. 933–955, 2003.

[4] E.N. Haslinger, R.E. Lopez-Herrejon, and A. Egyed. Reverse engineering feature models from programs' feature sets. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pp. 308 –312, 2011.

[5] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pp. 357 –366, 2012.

[6] K. Inoue, Y. Sasaki, P Xia, and Y. Manabe. Where does this code come from and where does it go? – integrated code history tracker for open source systems –. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 331–341, 2012.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[8] C. W. Krueger. Easing the transition to software mass customization. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pp. 282–293, 2001.

[9] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou. Inferring repository file structure modifications using nearest-neighbor clone detection. In *Proceedings of the 19th Working Conference on Reverse Engineering*, pp. 325 –334, 2012.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 176–192, 2006.

[11] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proceedings of the 29th international conference on Software Engineering*, pp. 106–115, 2007.

[12] E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, Vol. 1, No. 1, pp. 251–266, 1986.

[13] Development of FFmpeg under new management. `http://www.h-online.com/open/news/item/Development-of-FFmpeg-under-new-management-1172919.html` 2013 年 1 月 21 日閲覧.

[14] D. L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pp. 279–287, 1994.

[15] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[16] C. R. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, Vol. 36, pp. 1389–1041, 1957.

[17] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference*, pp. 156–160, 2012.

[18] M. T. Valente, V. Borges, and L. Passos. A semi-automatic approach for extracting software product lines. *IEEE Transactions on Software Engineering*, Vol. 38, No. 4, pp. 737–754, 2012.

[19] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring similarity of large software systems based on source code correspondence. In *Proceedings of the 6th International Conference on Product Focused Software Process Improvement*, pp. 530–544, 2005.

[20] K. Yoshimura and R. Mibe. Visualizing code clone outbreak: An industrial case study. In *Proceedings of the 6th International Workshop on Software Clone*, pp. 96–97, 2012.

[21] K. Yoshimura, F. Narisawa, K. Hashimoto, and T. Kikuno. FAVE: factor analysis based approach for detecting product line variability from change history. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pp. 11–18, 2008.

[22] 野中誠, 桜庭恒一郎, 舟越和己. 組込みソフトウェア製品ファミリにおける是正保守の予備的分析. 情報処理学会研究報告, Vol. 2009-SE-166, No. 13, pp. 1–8, 2009.

[23] 佐々木裕介, 山本哲男, 早瀬康裕, 井上克郎. 大規模ソフトウェアシステムを対象としたファイルクローンの検出. 電子情報通信学会論文誌. D, 情報・システム, Vol. 94, No. 8, pp. 1423–1433, 2011.