大規模システム分析のためのコールグラフの多段階階層的クラスタリング
Multilevel hierarchical clustering of call graphs for large-scale system analysis

李丞浩
Seungho Lee
法政大学情報科学部コンピューター科学科
seungho.lee.2c@stu.hosei.ac.jp

Abstract

A static call graph, one of the methods for understanding the software structure, illustrates the calling relationship between functions in an easy-to-understand manner for humans and intuitively understands the software structure. However, in the case of a large call graph, the number of edges is so large that it is difficult to check the software structure. Previous studies have tried to solve the problem by clustering and mapping all execution paths hierarchically by dividing call graphs into packages, classes, and function levels. But there were still many edges and difficulties in identifying relationships between functions. In this paper, we aim to make the call graph easier to see by adding Modular Decomposition to the abstraction levels proposed in the previous study by reducing the number of edges compared to the call graph. Modular Decomposition has a short learning time and can reduce the number of the call graph's edge by more than 40% on average to make it visible. Adding this level reduces subjects' task-achieving time by more than 30% compared to existing studies. Therefore, we show that call graphs can be fully utilized for software analysis, static analysis, etc. even in large-scale systems using the method proposed in this study.

1 序論

プログラムの理解はソフトウェアの再利用、デバッグ、テスト、維持及び発展に必須である。その中でもコールグラフは関数間の呼び出し関係からの流れ理解、手続き間分析、静的分析などで使われている。コールグラフは動的コールグラフ、静的コールグラフに別れていて動的コールグラフは Runtime 時にどんな関数が呼び出されたか単一実行経路をグラフで表し、静的コールグラフはコンパイル時に生成される関数と関数間の呼出関係を到達可能な全ての経路を持つ有向グラフで表現する。

一般的にプログラムを分析するとき、ソースコードだけだと一々コードを読みながら分析しないといけないが、関数をノードに、関数間の呼出関係をエッジで表現するコールグラフで呼出関係を可視化するとどんな関数が一番多く呼び出されるか、余計に書かれた関数はないか、コードの設計が正しくなっているかをより簡単に把握できる。

特に、大規模システムの場合、一々全てのコードを読みながらシステムを分析するのは難しいことでコールグラフを用いてシステムの構造や流れを可視化し静的分析を行うことができる。しかし、既存のコールグラフ生成ツール [5] のように大規模システムではシステムの構造を図 1 のように全てのノードとエッジを一枚の図で表現する静的コールグラフで可視化すると、システムの規模が大きくなるほどコールグラフのノードとエッジの数が多くなってエッジがどのノードに繋がっているかが分かりづらくなる問題があった。

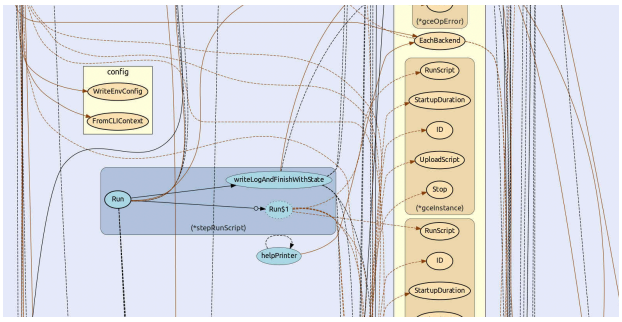


図 1: travis-ci コールグラフの部分拡大

本論文では大規模システムのコールグラフを可視化するとき情報量が多すぎる問題を解決するため、既存のコールグラフを多段階に抽象化し、すべての実行経路を階層的にクラスタリングする手法を参考に、コールグラフを多段階に可視化するとき、Modular Decomposition と呼ばれる edge compression 技術を用いてコールグラフのノードをモジュール化して可視化することでコールグラフを見やすくする方法を提案し、大規模システムでのコールグラフに Modular Decomposition を適用する方法を評価する。

2 関連研究

2.1 コールグラフの多段階抽象化

Codex では、大規模システムをコールグラフで可視化するとき、ノードとエッジの数が多くて関数間呼び出し関係が分かりにくい問題を解決するために、ソフトウェアシステムの大規模コールグラフを簡単に可視化し理解を手伝うデータ中心のアプローチを用いる。

コールグラフを異なるレベルに多段階抽象化しコールグラフのスケールを調節し、コールグラフから各抽象化段階で実行経路のクラスターを選択しながら段階的に探索したい経路のコールグラフだけ可視化する（図２）。コールグラフを最上位のパッケージ段階、次にクラス段階、最終的に関数段階まで多段階に抽象化させて、階層的にクラスタリングされた実行経路を各抽象化段階でユーザーが選択し、クラスターに属するノードを次

図 2: Codex のコールグラフ

Supervisor: Prof. Katunobu Itou

の抽象化段階で展開するツールを作成し、ユーザーが徐々にシステムを探索できる、被験者実験の結果、大規模システムを探索するとき役立つ、段階的にシステムを探索することは分かりやすいなどの評価を得た。しかし、この手法には、

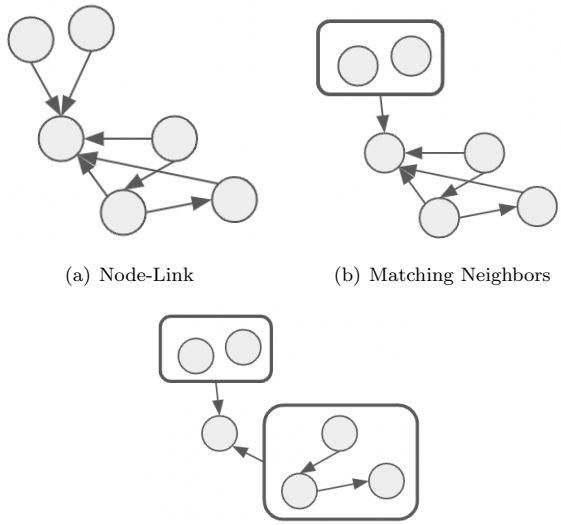
- パッケージ, クラス, 関数レベルの抽象化は OOP 言語でしか使えない。
- クラスレベルからすぐ関数レベルにコールグラフを展開するとノードとエッジの数がまだ多けて分かりづらい。

などの問題があった。本研究では特に、クラスレベルから関数レベルに抽象化段階を変えた時ノードとエッジの数が多くて分かりにくい問題を解決するための手法を提案する。

2.2 Modular Decomposition

Modular Decomposition はグラフのノードやエッジなどの情報を圧縮するための技術で情報の損失を伴う maximal modularity clustering[8] とは違ってノードとエッジの情報を損失なくモジュール化してグラフを圧縮する技術である。

この研究 [3] では三つのグラフのエッジの数を圧縮するための技術を紹介していて、各ノードの neighbors を一つのグループにまとめて可視化する Matching Neighbors(図 3-b)、各モジュール内部のノードは同じモジュール内の他ノードと繋がっていないか全部繋がっているかでグループ化する Modular Decomposition(図 3-c)、モジュールの定義を緩和してエッジがモジュールの境界を越えられるようにしてよりグラフを圧縮する Power Graph Analysis の三つの技術を比較している。



(a) Node-Link (b) Matching Neighbors

(c) Modular Decomposition

図 3: エッジを圧縮する方法

その結果、ユーザーのグラフ探索速度は Matching Neighbors で可視化したグラフが元のグラフより 36%、Modular Decomposition でグラフを可視化した場合、さらに 17% の速度向上を記録した。

本研究ではコールグラフ抽象化の最も低レベルである関数コールグラフに Modular Decomposition を用いてまとめることでコールグラフの見やすさを向上させる。

3 提案手法

本研究で提案する手法としてはまず、既存研究の抽象化段階である Package、Class、Function レベルが OOP 言語でしか使えない問題を図**??**のコールグラフ多段化抽象化を行うとき、大体の言語が持っている構造である Package、File、Function レベルを持つように構成する。また、大規模システムの場合、最終レベルのコールグラフでもまだノードとエッジの量が多すぎる問題に関してコールグラフが持つエッジの数を圧縮するため、コールグラフデータを可視化するツールに渡す前に Modular Decomposition を行なってコールグラフをモジュール化してからデータを渡すことにする。

この時、Modular Decomposition を行う前に Matching Neighbors で同じ外部へのエッジを持つノードをグループを先に定めてからモジュール化作業を行う。

モジュール化作業は $O(|V^2|)$ の時間複雑度を持つので、処理時間の限界を避けるために全体のコールグラフではなく書く抽象化段階にマッピングされるコールグラフにだけ適用するように範囲を限定する。

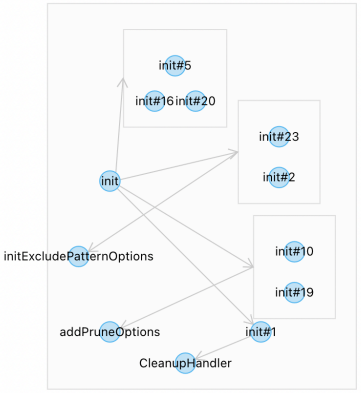


図 4: モジュールの可視化

また、モジュールを可視化する時は、図 4 のようにモジュールの内側でもエッジの情報を表現できるように枠とノード、エッジを持つように可視化し情報の損失を無くす。

4 可視化ツールの実装

本研究で提案する手法を実際試してみるために多段階モジュールコールグラフを可視化するツールを実装する。コールグラフの多段階抽象化はコールグラフ情報を動的にアップデートする必要があるため、コールグラフを可視化する部分は Web ページを用いて実装する。

また、コールグラフを加工する作業は python の networkx[6] を用いて実装し、ウェブ側とネットワークで通信しながらデータの交換を行う。

4.1 コールグラフ生成

まず、コールグラフオブジェクトを生成する。Go のソースコードからコールグラフのデータを抽出し dot 拡張子で出力する go-callvis[5] を拡張し caller-callee データを抽出する。その後、Nodes, Edges,Packages の情報を以下のように package、file、function に分けてエッジを表現したフォーマットで抽出するようにする。

```
1 Package/File/Function-Package/File/Function
```

次に、グラフは抽出したデータからノードとエッジの情報を分析し python の networkx[6] を用いて方向性を持つグラフオブジェクトを作ってノードとエッジを追加しコールグラフを構成した後、パッケージ、ファイル、関数三つの段階に抽象化したグラフオブジェクトを生成する。

4.2 コールグラフの多段階抽象化

本研究では Codex の手法を使ってコールグラフを多段階に抽象化する。但し、従来研究で提案したパッケージ、クラス、関数のレベルではなくパッケージ、ファイル、モジュールレベルに構成するようにし段階ごとに実行経路をクラスタリングさせるようにする。

ここでパッケージレベルはパッケージ間の呼び出し関係を、ファイルレベルではファイルごとの呼び出し関係、モジュール

レベルはコールグラフを Modular Decomposition でグループ化したモジュールを持つグラフを持つようにする。

4.3 実行経路の階層的クラスタリング

この段階ではユーザーが多段階に抽象化されたコールグラフを探索するとき、ユーザーが興味を持つ経路を選択できるようにするために各抽象化段階に渡されるコールグラフの実行経路を階層的クラスタリングする。

4.3.1 Feature Matrix

表 1: クラスタリングのための経路を行列化する

	f1	f2	f3	f4	f5	f6	f7	f8
P1	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
P2	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
P3	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0
P4	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0

この段階では抽象化された Package レベル、File レベルなどのコールグラフからサイクルを削除し、全ての実行経路を抽出する。抽出した経路は表 1 のように経路が通る関数は 1、通らない関数は 0 の Binary で表現した Feature Matrix を作成する。

4.3.2 linkage

クラスタリングをするときに使われる linkage 法としてソフトウェアをソースコードからファイル構造などの情報を抽出するために使われる software architecture recovery でよく使われるクラスター間距離の最小値を繋ぐ simple linkage、クラスター間距離の最大値をもつクラスターを繋ぐ Complete Linkage、クラスター間距離の平均値を基準に繋ぐ Average Linkage の中で、ソフトウェアのクラスタリングに一番適合だと言われている [4]Complete Linkage 法を用いてクラスタリングする。

4.3.3 類似度

本研究では隣接行列を用いてクラスター間の距離を測定する。この時ソフトウェアをクラスタリングする時 Complete Linkage 法と組み合わせることでより信頼できる結果を得られると言われた [4]Jaccard Similarity を使う。

$$d(P_i, P_j) = 1 - \frac{|P_i \cap P_j|}{|P_i \cup P_j|}$$

4.3.4 階層的クラスタリング

階層的クラスタリングとはツリーを利用して個別個体を順次、階層的に類似した個体ないしグループと統合して群集化するアルゴリズムであり、群集数を事前に定めなくてもクラスタリングを遂行できる。

特徴としては、個体が結合される順序を示すツリー形態の構造である Dendrogram を生成した後、適切な水準でツリーを切れば、全体データを希望する数だけの群集に分けるようになる。

本研究では Agglomerative Hierarchical Clustering を用いて抽象化された Package レベルと File レベルのコールグラフからの単純実行経路を階層的にクラスタリングする。

Algorithm 1 Agglomerative Clustering

```
Clusters ← list
while Clusters.length ≥ 2 do
    C_i, C_j ← min(dist)
    C_new ← C_i C_j
    Update Clusters
    Update DistanceMatrix
```

4.4 Edge Compression

この段階では各抽象化段階で選択されたクラスターに属している関数のコールグラフを可視化する前に Modular Decomposition でノードをグループ化しコールグラフのエッジの数を減らす。

4.4.1 Matching Neighbors

エッジ圧縮を提供するモジュールの最も簡単な定義で、モジュールの全てのノードが同一な親ノードと子ノードを持っているモジュールになっている。

つまり、u, v が同じグループであれば $N^+(u) = N^+(v), N^-(u) = N^+(v)$ の条件を満たす。この条件に加えて、本研究ではこの条件を緩和し u, v が同じグループであれば $u \in N^+(v), v \in N^+(u), N^+(u) \backslash \{v\} = N^+(v) \backslash \{u\}$ 及び逆の条件を満たすとき同じグループとする。

4.4.2 Modular Decomposition

Matching Neighbors 作業を完了した後、グラフからモジュール化できる部分を見つけてノードをモジュール化する。各モジュール内のノードは他モジュールのノードに繋がっていないか、モジュール内の全てのノードに繋がっている。

つまり、module M、vertex V で $M \subset V$ が modular decomposition で作成されるモジュールであるとしたら、$u, v \in M, N^+(u) \backslash M = N^+(v) \backslash M$ 及び逆も成り立つ条件を満たす。

この条件を満たすモジュールを探してモジュール化した後、そのモジュールを新しいノードとして用いることでグラフのエッジの数を減らす。

4.4.3 可視化ツールの実装

コールグラフを可視化し、探索するプロセスは動的に動かすためにブラウザーの画面からグラフを可視化し、既存の node-link グラフをパッケージレベルを可視化したあと、パッケージレベルから選んだ経路クラスターに属するファイルの関係をファイルレベルのコールグラフで可視化する。同じくファイルレベルから選択された経路クラスターをモジュールレベルに渡してクラスター内の関数の呼び出し関係を Modular Decomposition で生成されたモジュールにノードを入れ替えて可視化するプロセスを構築する。

5 実験

実験では多段階モジュール化コールグラフが既存の多段階コールグラフと比べてシステムの構造を理解するときより効率的かを調べるために、モジュール化による圧縮率を確認するための性能実験、実際ユーザーがシステムを分析する時の状況を仮定して既存のコールグラフと比較するための被験者実験を同時に行う。

5.1 性能実験

性能実験は M1 Pro、32gb メモリーを持つ PC から Go 言語で書かれたそれぞれ異なる規模を持つ四つのプロジェクト go-callvis、restic、d2、gh-cli を対象に行った。

測定方法としては、最初のパッケージレベルから経路のクラスターに属するノードの数が 6 つ以下になるまで繰り返してクラスターを選択し、コールグラフをパッケージレベルからファイルレベルにマッピングする。

ファイルレベルからも同じく経路のクラスターに属するノードの数が 6 個以下になるまで繰り返しながらランダムにクラスターを選択し、関数レベルのコールグラフにマッピングするようにして関数レベルのコールグラフでのノードとエッジの圧縮率、全体プロセスにかかった時間を測定する作業をプロジェクトごとに 20 回繰り返し測定された結果の平均値を求めた。

本研究で提案するモジュール化は全ての抽象化段階で行なっているが、本実験では実際実行経路を分析するとき使われる関数レベルでの圧縮率のみ測定した。

表 2 に測定結果をまとめた結果、ノードは平均 39%、エッジは平均 31% が圧縮されていた。この結果で gh-cli だけ平均圧縮率の 5 割しか圧縮できなかったのは、gh-cli は go-callvis、restic、d2 より独立した機能の実装が多くて同じ外部へのエッジを持つノードが少なかったことを挙げられる。

表 2: プロジェクトから多段階に抽象化した結果

		original	function level		圧縮率	処理時間
			before	after		
go-callvis	node	190	16.45	9	50%	0.05s
	edge	246	12.25	7.95	38%	
restic	node	808	39.45	24.35	42%	1.2s
	edge	1954	37.95	29.1	31%	
d2	node	160	35.25	20.6	42%	0.13s
	edge	245	35.75	23.15	36%	
gh-cli	node	197	20.1	15.7	22%	0.02s
	edge	322	16.9	13.7	19%	

今回の実験結果から、既存の関数レベルで関数の呼び出し関係をコールグラフで表すより、Modular Decomposition を用いてコールグラフを可視化させた方がノードの数は平均 39%、エッジの数は 31% 減らせる結果を得て、モジュール化はコールグラフの複雑さを減らせることが分かった。

5.2 被験者実験

被験者実験では既存研究のコールグラフと本研究のコールグラフを比較する。被験者にシステム分析に関した課題を出して既存のコールグラフでの探索時間と本研究で提案する探索時間を比較し、本研究で実装したツールの有用性を評価する。

5.2.1 準備

本研究のツールの有用性を評価してもらうために被験者には Matching Neighbors 及び Modular Decomposition の概念を理解させ、システムを分析する課題を出した。今回の実験では性能測定でノードとエッジの圧縮率が平均圧縮率と最も近く、ノードとエッジの数も多数持つ restic を対象に実験を行った。

5.2.2 課題

被験者に解かせる課題は静的分析で検知する問題を解決する状況を想定し、SonarQube[7] で提供する分析機能の中でバグと脆弱点で問題が起きた場合のシステム分析課題とツールを理解するための経路探索課題を出した。

- Q1. init$17 から loadPasswordFromFile までの経路を 3 つ求めてください。
- Q2. init$20 から stdinIsTerminal までの経路を 3 つ求めてください。
- Q3. decidePackAction 関数でバグが発生した場合、cmd_rebuild_index.go、 master_index.go ファイルの中だとどんな関数が影響を受けるか。
- Q4. OpenRepository 関数で脆弱点が発見されたとき、types.go、file_unix.go、main.go ファイルの中だとどんな関数が影響を受けるか。
- Q5. 削除すると最も大きな影響を与えるファイルは何か？

5.2.3 結果

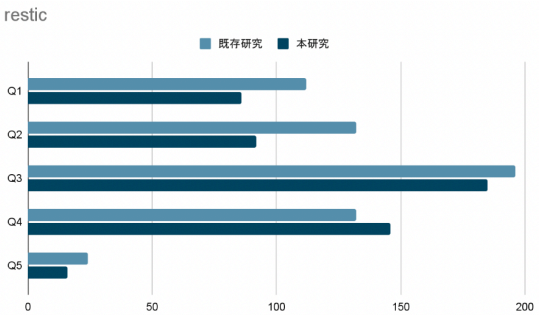


図 5: 多段階モジュール化コールグラフの可視化ツール

図 5 に一連の課題の既存方式のコールグラフと本研究で提案した多段階モジュール化コールグラフでの作業時間を示す。

その結果、Q1、Q2 のように単純な実行経路探索では本研究で提案する手法が平均 27.5% 早く、Q3、Q4 のように特定ファイルの範囲内でバグや脆弱点を探す課題に関しては有意な差はなかった。最後の Q5 に対してはシステムの中でもファイルの全体像を確認する課題でモジュール化されたコールグラフの方が約３０ % 早い結果を得られた。

6 考察

本研究での性能実験と被験者実験の結果から見ると、モジュール化されたコールグラフは既存のコールグラフよりエッジの圧縮されて経路などを探索するときかかる時間が短くなることがわかった。

しかし、既存研究で提案した多段階抽象化階層的クラスタリング手法にはまだ限界があって、性能の面ではノードとエッジの構造が複雑なコールグラフの場合、全ての実行経路を抽出する作業が出来なくなって階層的クラスタリングができない問題があった。

また、実行経路をクラスタリングしていて、コールグラフを関数レベルで細密化する場合、システムの全体的な構造ではなくクラスター内の一部実行経路のコールグラフしか確認できないことで被験者実験の Q3、Q4 のようにシステムの特定範囲全てを探索するのは難しくなる問題があった。

今後の課題としては実行経路のクラスタリングを使わずに複雑なノードとエッジを持つコールグラフを多段階に抽象化できる手段を考える必要がある。本研究の提案手法も既存研究の手法を採用しているため、コールグラフから全ての実行経路を抽出できるシステムに限って適用することができる。

7 結論

本研究では、大規模システムのコールグラフを可視化する時コールグラフを多段階に抽象化し modular decomposition をすることで全ての実行経路を探すことで全体のコールグラフを Modular Decomposition せずに大規模システムのコールグラフをモジュール化する方法を提案した。本手法は性能実験の結果、エッジの数を平均 31 %減らし、被験者実験の結果から大規模システムの場合だと、システムの全体の繋がりを確認する作業以外ではエッジの数が圧縮されコールグラフから経路を探索する時、コールグラフの複雑さが減ったことを体感できる。

参考文献

[1] Ofabry, go-callvis, https://github.com/ofabry/go-callvis

[2] Alanazi, R., Gharibi, G. & Lee, Y. Facilitating program comprehension with call graph multilevel hierarchical abstractions. *Journal Of Systems And Software*. **176** pp. 110945 (2021)

[3] T. Dwyer, N. Henry Riche, K. Marriott and C. Mears, "Edge Compression Techniques for Visualization of Dense Directed Graphs," in IEEE Transactions on Visualization and Computer Graphics, vol. 19, no. 12, pp. 2596-2605, Dec. 2013, doi: 10.1109/TVCG.2013.151.

[4] O. Maqbool and H. Babri, "Hierarchical Clustering for Software Architecture Recovery," in IEEE Transactions on Software Engineering, vol. 33, no. 11, pp. 759-780, Nov. 2007, doi: 10.1109/TSE.2007.70732.

[5] go-callvis, https://github.com/ofabry/go-callvis

[6] networkx, https://networkx.org

[7] https://www.sonarsource.com/products/sonarqube/

[8] M. Newman. Modularity and community structure in networks. Proceedings of the National Academy of Sciences, 103(23):8577–8582, 2006