OUKA
The University of Osaka
Institutional Knowledge Archive

Title	ソフトウェア部品分類手法へのコンポーネントランク法の応用
Author(s)	中塚, 剛; 松下, 誠; 井上, 克郎
Citation	情報処理学会研究報告. ソフトウェア工学研究会報告. 2007, 2007-SE-155(33), p. 41-48
Version Type	VoR
URL	https://hdl.handle.net/11094/50192
rights	ここに掲載した著作物の利用に関する注意 本著作物の著作権は情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－SE－155 (6)
2007／3／22

ソフトウェア部品分類手法へのコンポーネントランク法の応用

中塚 剛† 松下 誠† 井上 克郎†

† 大阪大学 大学院情報科学研究科
〒 560-8531 大阪府豊中市待兼山町 1-3

ソフトウェア部品を分類することによって，巨大なシステムを複数のサブシステムに分解するソフトウェアクラスタリング手法は，大規模なソフトウェアを理解するために有用である．しかし，システム内には，特定のサブシステムに属すべきでないライブラリのような部品も存在する．そのような部品を事前に特定することにより，クラスタリング結果の質が向上することが期待できる．本稿では，ソフトウェア部品の重要度を測定するコンポーネントランク法を用いることにより，特定のサブシステムに属すべきではない部品を特定する手法の提案を行う．提案する手法を用いたクラスタリングツールを用いて実験を行い，得られた結果に対する考察を行った．その結果，従来手法よりも優れたクラスタリング結果を得られることがわかった．

An Application of Component Rank Model to Software Clustering Algorithm

Gou Nakatsuka† Makoto Matsushita† Katsuro Inoue†

† Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan

Decomposing a large software system into some subsystems by grouping software modules helps us understand its structure and functions. In the system, however, there are special modules such as libraries that should not belong to a certain subsystem, and pre-detecting such modules can make the clustering results better. This paper proposes a new method of detecting such modules by Component Rank Model, that is used to measure mutual significance of software modules. We performed experiments with a new clustering tool combined with our detecting method and investigated the clustring results. As a result, our method turned out better than an existent method.

1 まえがき

ソフトウェア開発現場において，保守やソフトウェア部品再利用を行う際，あるソフトウェアシステム（以降，システム）の構造や機能を理解する必要がある [10]．しかし，仕様書等の不備や，ソフトウェアの巨大化，複雑化等の原因によりソフトウェア理解が非常に困難な場合が多い．

このような場合に，ソフトウェア理解を助ける手法の一つとしてソフトウェアクラスタリングという手法が研究されている．ソフトウェアクラスタリングとは，ソフトウェアシステムを構成する様々なソフトウェア部品（例えばJavaの場合，クラスやインターフェース）を何らかの方針に従って分類して，システムを複数のサブシステムへ分解する手法であり，これまでに様々な手法が提案されている [3,9]．以下，ソフトウェアクラスタリングにおいて，システムに含まれる分類の対象となる部品の単位を「モジュール」と呼ぶ．

ソフトウェアクラスタリングの方針の一つとして，各サブシステムが強凝集・低結合になるように分類を行う方法がある [5,7]．これにより，密に依存しあっているモジュールをクラスタとするこ

とによってサブシステムを構成し，サブシステム間の依存関係をできるだけ少なくするというものである．

しかし，ソフトウェアクラスタリングの結果，システム内の全てのモジュールをいずれか一つのサブシステムに属させないほうが良い場合がある．一般的にライブラリと呼ばれるようなモジュール，例えば文字列を操作するようなモジュールは様々なモジュールから利用されており，特定のサブシステムに含めることは適切ではない．また，例えば `java.awt` パッケージ内の `java.awt.Component` クラスのように，様々なクラスから参照もしくは継承されるような抽象データクラスも，特定のサブシステムに属するべきではない．このようなモジュールは遍在モジュール（omnipresent module）と呼ばれている [6]．遍在モジュールはソフトウェアクラスタリングを行う前に特定し，事前にクラスタリング対象から除去する必要がある．

従来のソフトウェアクラスタリング手法では，あるモジュールが依存関係を持つモジュールの数が，ある閾値以上のモジュールを遍在モジュールとするという単純な手法で特定している [5,6]．つまり，従来手法では直接的な依存関係のみを考慮しており，間接的な依存関係を考慮していない．特にオブジェクト指向言語の場合，継承やインターフェース実装といった依存関係も考慮するため，間接的な依存関係がより重要になると考えられる．しかし，これまでに遍在モジュールに関して詳細な考察を行っている例は少ない．

一方，Java ソフトウェア部品検索システム SPARS-J [1] では，システムに含まれるモジュールの重要度を計算するためにコンポーネントランク法と呼ばれる手法が用いられている．この手法は，あるシステム内のソフトウェア部品の直接的，間接的な相互利用関係を解析することによって，各ソフトウェア部品の重要度を計算する．このコンポーネントランク法によって上位にランク付けされたクラスは，ソフトウェア中の多くから利用される重要な部品であることがわかっており [12]，これは，遍在モジュールの満たすべき性質と似ている．

そこで本稿では，コンポーネントランク法を利用して遍在モジュールを特定する手法を，実験によって評価，決定する．その結果，本手法が従来のソフトウェアクラスタリングアルゴリズムを改良することを示す．

以降，2 節で研究の背景であるソフトウェアクラスタリングについて説明した上で，3 節にて本研究に関連する 2 つの手法について説明する．4 節で，コンポーネントランク法を用いた遍在モジュール特定手法を決定するための実験と結果について説明し，5 節でまとめと今後の課題について述べる．

2 ソフトウェアクラスタリング

2.1 概要

ソフトウェアクラスタリングは，システムを構成する多くのモジュールを，ある特定の基準に基づいて分類し，システムを複数のサブシステムに分解することによって，ソフトウェア理解を助けるための手法である．多くのクラスタリングアルゴリズムは，システムの構造に基づいて分類を行う．

システムは，複数のモジュールがお互いに依存しあって成立しており，その様子は図 1 の左図のように，モジュールを頂点，依存関係を有向辺とした有向グラフ（以下，依存関係グラフ）を用いて表される．ここで，モジュールとは，C 言語などの手続き型言語の場合ファイルを指し，Java などのオブジェクト指向言語の場合クラスあるいはインターフェースを指す．また依存関係とは，関数呼び出し，変数参照，継承，インターフェース実装を指し，場合によっては依存の程度を表す辺に重みをつけることもある．

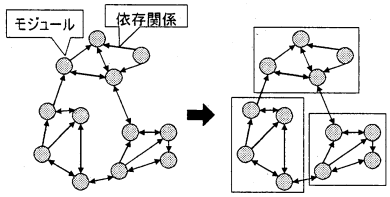

図 1: クラスタリングの例

このような依存関係グラフを分割する方針の一つとして，高凝集・低結合の原則に基づいて分割するというものがある [5,7]．具体的には，図 1 の左図で示されるような依存関係グラフで表されるシステムを，右図のように依存関係が密な部分をクラスタとしてまとめる方法である．この結果，クラスタ内の辺が多く，クラスタ間の辺が少なくなるような分割となる．このようにして作られたクラスタは，に着目することによってその動作を理解

することが容易となり，ソフトウェアを理解するための有用な方針として広く研究されている [5,7,8].

2.2 遍在モジュール

しかしながら，全てのモジュールを，高凝集・低結合となるようなサブシステムへ分類することが適切ではないことがある．その要因として，遍在モジュール（omnipresent module）と呼ばれるモジュールの存在が知られている．遍在モジュールは，一般的にユーティリティやライブラリと呼ばれるようなシステム内の多くのモジュールと依存関係のあるモジュールを表している．このようなモジュールは，特定のサブシステムに属すべきではない（図 2）.

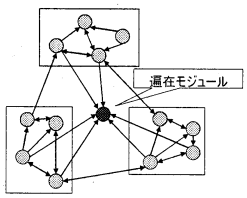

図 2: 遍在モジュールの例

そのため，ソフトウェアクラスタリングを行う前に，遍在モジュールを特定し，遍在モジュールとそれに関連する辺を除去したグラフに対して，クラスタリングを行うアルゴリズムが知られている．

従来の遍在モジュール特定手法は，ある閾値を決めて，次数が閾値以上のモジュールを遍在モジュールとしている．このとき，閾値の決め方はクラスタリングアルゴリズムよって異なり（例えば，頂点数の平方根を取ったもの [2] や，平均次数の 3 倍 [5] など），どのような条件が遍在モジュールを特定するために有効であるかという詳細な考察はほとんど行われていない．次数のみで決定する手法は，単純すぎるといえ，次数が閾値以下であっても，多くのモジュールに直接的あるいは間接的に依存されているモジュールは遍在モジュールとするべきである．

3 関連研究

この節では，本研究に関連する 2 つの研究について紹介する．

3.1 Bunch

高凝集・低結合の原則に基づくクラスタリングを行うシステムとして，現実的な実行時間で計算を行えるシステムが，Mancoridis，Mitchell らによって開発された Bunch [5] である．Bunch では，ソフトウェアクラスタリングを探索問題として定義することにより解を求める．つまり，分割結果を評価するための目的関数を定義して，考えられる分割全てに目的関数を適用して評価値が最も高いものを解とする．しかし，頂点数 n のグラフに対するクラスタリング結果の総数は $O(n!)$ となり，この中から最適なものを探す問題は NP 困難である [8]. そこで，局所探索法である山登り法（hill climbing）によって近似解を求めている．

Bunch では，分割を評価するための目的関数として，MQ（Modularization Quality）を定義している．これはサブシステムの凝集度と結合度のトレードオフを関数化したもので，以下のように与えられる．なお，評価する分割中のクラスタ数を k とする．

まず，クラスタ $i\,(1 \le i \le k)$ 内に含まれる辺（クラスタ i 内の 2 頂点間にある辺）の数を μ_i とする．また，クラスタ i からクラスタ j への辺（クラスタ i 内の頂点からクラスタ j 内の頂点への辺）の数を $\varepsilon_{i,j}$ とする．

次に，各クラスタ i に対して関数 CF_i（Cluster Factor）を次のように定義する．

$$CF_i = \begin{cases} 0 & (\mu_i = 0) \\ \dfrac{2\mu_i}{2\mu_i + \sum_{j=1, j\neq i}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & (\mu_i > 0) \end{cases}$$

これは，各クラスタに関連する辺のうち，クラスタ内部にある辺の割合を表している．また，全てのクラスタの CF の合計を MQ とする．

$$MQ = \sum_{i=1}^{k} CF_i$$

Bunch では，遍在モジュールとして，次の 4 種類を定め，次のような特定条件を与えている．

- **supplier**：入次数が平均次数の 3 倍以上
- **client**：出次数が平均次数の 3 倍以上
- **central**：supplier かつ client
- **library**：出次数が 0

Bunch の遍在モジュール特定手法の問題点として，2.2 節で述べたように，次数のみによる単純な評価であるという点に加えて，遍在モジュールの数が多過ぎることがあげられる．特に，出次数 0 のモジュールは多く，4 節で示す実験結果においては，

全モジュールの 10~30%が library として抽出されてしまう．この結果，全モジュールの 30~50%が遍在モジュールとして除去される．遍在モジュールはクラスタに属さず，個々に理解する必要があるため，遍在モジュールが多過ぎるとクラスタリング結果は見にくくなる．

指定した遍在モジュールによって，出力されるクラスタリング結果は当然変化するが，本稿では，適切な遍在モジュールを特定することによって，Bunch の出力する解を改善することを目的とする．

3.2 コンポーネントランク法

Java ソフトウェア部品の再利用を助けるための，部品検索システムとして SPARS-J が開発されている [1]．このシステムは，Java で書かれたソースコードを解析し，クラスやインターフェースをソフトウェア部品の単位としてデータベースに情報を記録し，与えられた検索単語に対して関連するソフトウェア部品を提示する．その際，ソフトウェア部品を提示する順番を決定する手法として，ソフトウェア部品の重要度を数値化するコンポーネントランク法（Component Rank 法，以下 CR 法）が用いられる．

CR 法では，依存関係グラフ上の各頂点に総和が 1 となるように，それぞれ均等に重みを与え，各頂点がその頂点を始点とする有向辺に重みを均等に分配することによって，その辺の終点の新たな重みが決定する．これを各頂点の重みが収束するまで繰り返し，収束した値をコンポーネントランク値（Component Rank Value，以下 CRV）とする．CRV の計算の例を図 3 に示す．

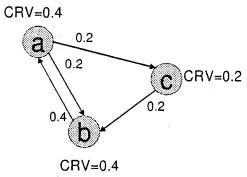

図 3: CRV 計算の例

2 つの部品から使用されている b の CRV は高くなっており，また CRV の高い b から使用されている a の CRV も高くなっている．このように，たくさんの部品から使用されている部品の CRV は高くなり，また，CRV の高い部品から使用されている部品の CRV も高くなる．

一般的に CRV の分布は，一部の部品が非常に高い CRV を持ち，ほとんどの部品の CRV は低くなることがわかっている．この性質を利用して，CRV が非常に高い部品に着目することによりソフトウェアの理解を容易にするための研究も行われている [12]．

本稿では，「CRV の高い部品は，多くの部品に直接的あるいは間接的に使用されている部品であり，遍在モジュールの候補ではないか」，という点に注目して，CRV を用いた遍在モジュール特定手法を提案する．

4 CRV を用いた遍在モジュール特定手法

本節では，本稿で提案する，CRV を用いた遍在モジュール特定手法について述べる．

Bunch で定義されている 4 種類の遍在モジュールを CRV，次数によって特定することを考える．その際，どのような閾値条件を用いれば，解を改良できるか検証するための実験を行った．

まず，考えられる特定条件として 40 の条件を定義し，その 40 の条件を比較するために，2 種類の実験を行い，実験結果より最適な条件を決定する．以下にその詳細を述べる．

4.1 実験対象となる特定条件

本実験では，表 1 に示すように，C1~C40 の遍在モジュール特定条件を実験対象として定めた．ここで in，out，CRV の列はそれぞれ入次数，出次数，CRV の条件を表し，数字は平均値からの倍率を表している．数字が正の場合，例えば 3 の場合は，平均値の 3 倍以上ならば該当とするが，数字が 0 の場合は，値が 0 でないと該当しない．

C1 は Bunch でデフォルトで用いられている特定条件であり，これよりも良い条件を探すことを目的とする．C2 は，遍在モジュールを全く考慮しない条件であり，C3~C5 は C1 の倍率を変更したものである．C6 は，遍在モジュールの数を増やし過ぎる原因であると思われる library を考慮しない条件である．

C7 以降は，supplier，central，library の条件に CRV を使用している．この際，CRV が平均を超えるモジュールは全体の約 20%と少ないため，CRV の条件としては平均以上のみを考慮する．supplier の特定条件として，CRV 平均以上を固定して，入次数条件を変化させたものでまとめてある．client

表 1: 遍在モジュール特定条件（実験対象，数字は平均からの倍率）

No.	supplier		client	central			library		
	in	CVR	out	in	out	CVR	in	out	CVR
C1	3		3	3	3			0	
C2									
C3	2		2	2	2			0	
C4	2.5		2.5	2.5	2.5			0	
C5	3.5		3.5	3.5	3.5			0	
C6	3		3	3	3				
C7		1	3		3	1		0	
C8		1	3		3	1			
C9		1	3		3	1		0	1
C10		1	3		3	1	1	0	1
C11		1	3	3	3	1	1	0	1
C12	1	1	3	1	3	1		0	
C13	1	1	3	1	3	1			
C14	1	1	3	1	3	1		0	1
C15	1	1	3	1	3	1	1	0	1
C16	1	1	3		3	1	1	0	1
C17	1	1	3	3	3	1	1	0	1
C18	1.5	1	3	1.5	3	1		0	
C19	1.5	1	3	1.5	3	1			
C20	1.5	1	3	1.5	3	1		0	1
C21	1.5	1	3	1.5	3	1	1	0	1
C22	1.5	1	3		3	1	1	0	1
C23	1.5	1	3	3	3	1	1	0	1
C24	2	1	3	2	3	1		0	
C25	2	1	3	2	3	1			
C26	2	1	3	2	3	1		0	1
C27	2	1	3	2	3	1	1	0	1
C28	2	1	3		3	1	1	0	1
C29	2	1	3	3	3	1	1	0	1
C30	2.5	1	3	2.5	3	1		0	
C31	2.5	1	3	2.5	3	1			
C32	2.5	1	3	2.5	3	1		0	1
C33	2.5	1	3	2.5	3	1	1	0	1
C34	2.5	1	3		3	1	1	0	1
C35	2.5	1	3	3	3	1	1	0	1
C36	3	1	3	3	3	1		0	
C37	3	1	3	3	3	1			
C38	3	1	3	3	3	1		0	1
C39	3	1	3	3	3	1	1	0	1
C40	3	1	3		3	1	1	0	1

特定条件には，CRV は使用しにくいため，出次数平均 3 倍以上で固定している．central に関しては，supplier と client の両方の性質を持つものという意味から，2 つの条件を足したような条件とした．library に関しては，出次数 0 だけでは条件が緩すぎるため，CRV 平均以上や，入次数平均以上という条件を加えている．

4.2 実験 1：結果の理解しやすさによる評価

Bunch では，クラスタの凝集度と結合度のオーバーヘッドを CF，MQ という 2 つの関数で表す．このとき，その値が高いほどクラスタ間に辺が少なくなり，クラスタリング結果が理解しやすいものとなる．MQ は，値域が正規化されていないため，モジュール数が多いシステムほどその値は高くなる傾向があるが，CF は，0~1 の値を取る．そこで，本実験では，平均 CF 値（全クラスタの CF 値の平均値）に注目した．

ところが，平均 CF 値が高いほどクラスタリング結果として良いというわけではない．遍在モジュールを多く特定するほど，多くの辺が無視されることになり，平均 CF 値が高くなりやすいが，この場合，クラスタリングされたモジュールは理解しやすいといえるが，多く特定された遍在モジュールは個別に理解する必要があるため，全体として理解にかかるコストが増加してしまう場合がある．よって，クラスタリング対象モジュール数（遍在モジュールではないモジュール）の数と，平均 CF 値のバランスを考えて，両方ともある程度高い結果がが優れた条件であることとする．

方針　15 種類の Java で書かれたオープンソースシステムに対して 40 の特定条件を適用し，クラスタリング対象モジュール数と平均 CF 値が共に高い条件を求める．

結果と考察　15 システムに適用した結果の平均値を図 4 に示す．横軸は 40 の条件を表し，縦軸は，クラスタリング対象モジュール数の全モジュール数に対する割合と，平均 CF 値を示している．また，2 つの評価値の調和平均値も共に示してある．

左端の，C1 の結果が，Bunch オリジナル条件での結果であり，C2 と比べると，平均 CF 値が約 2 倍に増加しているが，遍在モジュール数が多過ぎることがわかる．

C3~C5 の結果は，閾値が増えるにつれ，クラスタリング対象モジュール数は増えており，逆に平均 CF 値は下がっていて，両者がトレードオフの関係となっていることがわかる．

C6 の結果は，平均 CF 値が下がり，クラスタリング対象モジュール数が約 40%の増加になっている．これは，出次数 0 という条件によって library が多く選ばれていることを示している．

C7 以降の結果は，C1 に比べて，2 つの評価値が共に増加している条件が多数存在している．図中に示した縦の破線で示したブロックごとに，supplier に対する条件が厳しくなるが，結果は，クラスタリング対象モジュール数は増加し，逆に平均 CF 値が減少している．つまり，トレードオフとなっているが，調和平均値が改善していることがわかる．

では，このクラスタリング対象モジュール数と平均 CF 値のどちらを優先すべきかという点を考えると，遍在モジュールはあくまでも特殊なモジュールであって，たくさんのモジュールを遍在モジュールとして特定して別扱いするよりも，少し見づらくなってもクラスタに入れた方が，理解しやすい

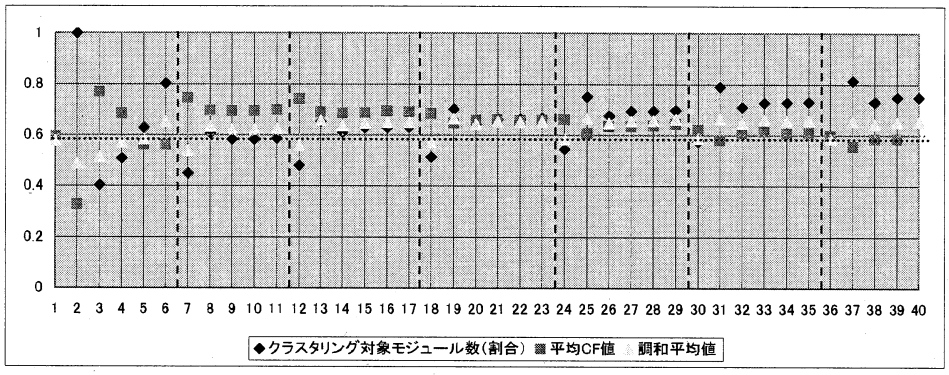

図 4: 実験 1 の結果

といえる．そこで，クラスタリング対象モジュール数を重視することにすると，C27~C29 の条件が優れていることがわかる．C27~C29 は，library の条件に CRV と入次数を加えたもので，C1 に比べて，library の数を程良く減らしている．

4.3 実験 2：ベンチマークによる評価

実験 1 では，クラスタリング結果に付随する表面的な評価値によって条件の比較を行った．次に，実際のクラスタリング結果を評価することを考える．

クラスタリングアルゴリズムを評価するためによく用いられる手法として，ベンチマークを作成してクラスタリング結果と比較するという手法がある [4, 11]．

ベンチマークと解を比較するために両者の距離を計る手法がいくつか提案されているが，ここでは EdgeSim と MeCl という 2 つのメトリクス [4] を用いた．EdgeSim（Edge Similarity）は，分解されたグラフ中で辺がクラスタ内に閉じているか，あるいはクラスタ間にまたがっているかという点に注目して，辺の類似度を測定するメトリクスである．一方 MeCl（Merge Clusters）は，片方のグラフを分割した後マージすることで，もう片方のグラフと一致させるために必要なコストを測定するメトリクスである．これらのメトリクスは共に，パーセント尺度で，値が高いほど 2 つのグラフが類似していることを意味する．ただし，この 2 つのメトリクスは，遍在モジュールを考慮しないため，遍在モジュールを含む結果に関して，正しい評価が行えない場合があるそのため，目視評価によって，その点を補う必要がある．

本実験では，実験 1 で用いたシステムのうちの一つである GNUJpdf に対してベンチマークを作成した．このシステムは，PDF ファイルを生成，印刷するための package で，25 モジュールから成る．作成したベンチマークを図 5 に示す．

方針 GNUJpdf のベンチマークと，40 条件におけるクラスタリング結果の類似度を EdgeSim，MeCl を用いて測定する．また目視による比較を行う．

結果と考察 実験 2 の結果を図 6 に示す．横軸は，40 の条件を表しているが，一部，下線を引いているものがある．これは，遍在モジュールの選び方によって，Bunch API の出力する解から一部のモジュールが削除されているというバグが存在し，そのため，Bunch API の解を一部書き換えて測定したことを表している．縦軸は，EdgeSim，MeCl を表している．両メトリクスの相関係数は 0.88 と非常に高い．

この結果から，CRV を用いることによって，EdgeSim，MeCl 共に値が改善している条件が多数存在することがわかる．実験 1 で結果の良かった C27~C29 は，実験 2 でも非常に良い結果を示している．ただし，C29 は，前述の Bunch API のバグのため，正しい解を出力できていない．

表 7 に C1 でのクラスタリング結果を，表 8 に C27，C28 でのクラスタリング結果（この 2 条件は同じ結果）を示す．C1 では，ベンチマークに比べて library が非常に多く，またクラスタ間の辺も多いことがわかるが，C27，C28 では，ベンチマークと異なるのは，PDFPage というモジュールが supplier か central かという違いのみで，ほとんど同じ結果

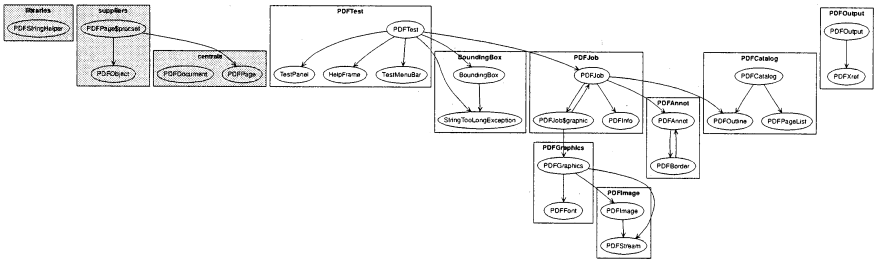

図 5: GNUJpdf のベンチマーク

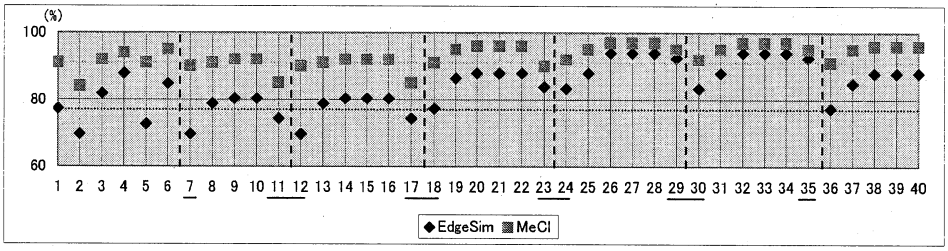

図 6: 実験 2 の結果

となっている.

4.4 実験のまとめ

GNUJpdf では，C27，C28 で同じ結果を示しているが，他システムでは結果は異なる．それらを調査したところ，Bunch API では，central の条件として，supplier かつ client としていない場合に正しくない結果を出力し得ることがわかった．C28 は，central の条件が supplier かつ client となっていないため，C27 が遍在モジュールを特定するための最も良い条件であることがわかった．その条件は具体的には以下の通りである．

- **supplier**：入次数が平均次数の 2 倍以上，かつ CRV 平均以上
- **client**：出次数が平均次数の 3 倍以上
- **central**：supplier かつ client
- **library**：入次数が平均以上，かつ出次数が 0，かつ CRV 平均以上

5 まとめ

本稿では，コンポーネントランク法によって測定されるコンポーネントランク値を用いて遍在モジュールを特定し，高凝集・低結合の原則に基づくソフトウェアクラスタリングを改良できることを確認した.

今後の課題としては，クラスタリング評価手法の拡張がまず挙げられる．現状では行っていない，遍在モジュールを考慮した上での評価を行えるように改良する必要がある．その上で，より大規模なシステムでのベンチマーク実験を行い，提案手法のさらなる評価を行いたい.

参考文献

[1] Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M. and Kusumoto, S.: Ranking Significance of Software Components Based on Use Relations, *IEEE Transactions on Sofware Engineering*, Vol. 31, No. 3, pp. 213–225 (2005).

[2] Luo, J., Zhang, L. and Sun, J.: A Hierarchical Decomposition Method for Object-Oriented Systems Based on Identifying Omnipresent Clusters, *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 647–650 (2005).

[3] Mitchell, B. S.: A Heuristic Search Approach to Solving the Software Clustering Problem, PhD Thesis, Drexel University, Philadelphia, PA (2002).

[4] Mitchell, B. S. and Mancoridis, S.: Comparing the Decompositions Produced by Software

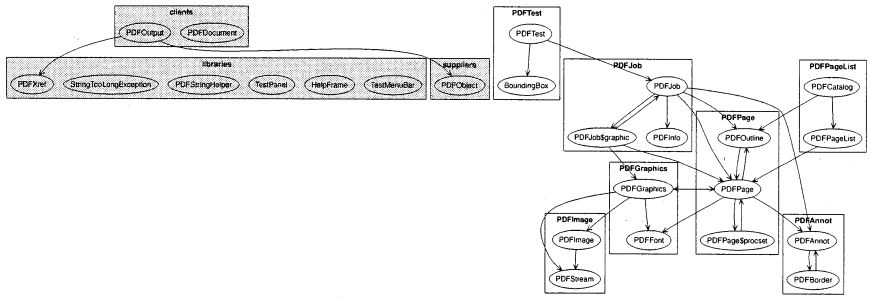

図 7: GNUJpdf における C1 でのクラスタリング結果

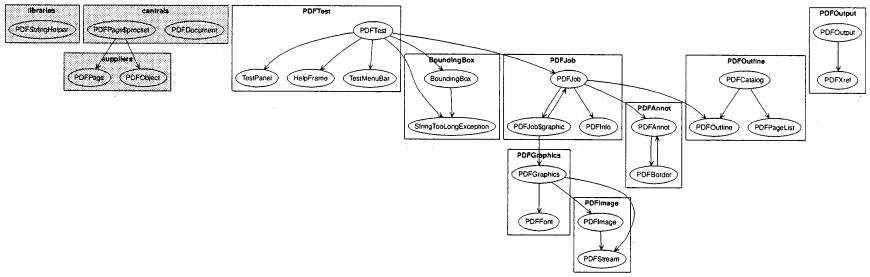

図 8: GNUJpdf における C27，C28 でのクラスタリング結果

Clustering Algorithms using Similarity Measurements, *Proceedings of International Conference on Software Maintenance*, pp. 744–753 (2001).

[5] Mitchell, B. S. and Mancoridis, S.: On the Automatic Modularization of Software Systems Using the Bunch Tool, *IEEE Transaction on Software Engineering*, Vol. 32, No. 3, pp. 193–208 (2006).

[6] Müller, H. A., Orgun, M. A., Tilley, S. R. and Uhl, J. S.: A Reverse Engineering Approach To Subsystem Structure Identification, *Journal of Software Maintenance: Research and Practice*, Vol. 5, No. 4, pp. 181–204 (1993).

[7] Schwanke, R. W.: An Intelligent Tool For Re-engineering Software Modularity, *Proceedings of the 13th International Conference on Software Engineering*, pp. 83–92 (1991).

[8] Shokoufandeh, A., Mancoridis, S., Denton, T. and Maycock, M.: Spectral and meta-heuristic algorithms for software clustering, *The Journal of Systems and Software*, Vol. 77, pp. 213–223 (2005).

[9] Tzserpos, V. and Holt, R. C.: ACDC : An Alogorithm for Comprehension-Driven Clustering, *Proceedings of the Seventh Working Conference on Reverse Engineering*, pp. 258–267 (2000).

[10] von Mayrhauser, A., Vans, A. M. and Lang, S.: Program Comprehension and enhancement of software, *IFIP World Computing Congress - Information Technology and Knowledge Engineering*, Vinna, Budapest (1998).

[11] Wen, Z. and Tzerpos, V.: An effectiveness measure for software clustering algorithms, *Proceedings of the International Workshop on Program Comprehension (IWPC '04)* (2004).

[12] 市井誠，横森励士，　松下誠，井上克郎：コンポーネントランクを用いたソフトウェアのクラス設計に関する分析手法の提案，電子情報通信学会技術研究報告，SS2005-37，Vol. 105, No. 229, pp. 25–30 (2005).