修士学位論文

題目

コンポーネントランク法を用いたJavaクラス分類手法の提案

指導教員
井上 克郎 教授

報告者
中塚 剛

平成19年2月13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 18 年度 修士学位論文

コンポーネントランク法を用いた Java クラス分類手法の提案

中塚 剛

内容梗概

大規模なソフトウェア・システムの構造・挙動の理解を容易にするために，システムに含まれる大量のソフトウェア部品を分類し，複数のサブシステムへ分解するソフトウェア・クラスタリングという手法が研究されている．例えば，密に依存しあっている部品群をサブシステムとしてまとめ，サブシステム間の依存関係を減らすことで，サブシステムごとの理解が容易になり，システム全体の構造がわかりやすくなる．しかし，システムの中には，一般的にライブラリと呼ばれるような，様々なソフトウェア部品と依存関係を持つ遍在モジュールと呼ばれる部品も存在する．このような部品は，特定のサブシステムに属するべきではないため，クラスタリングを行なう際に除去することが望ましいが，遍在モジュールを特定する方法に関して詳しい調査が行なわれていない．

そこで，Java を対象としたソフトウェア部品重要度ランキング手法であるコンポーネントランク法に注目した．この手法は，システム内の各クラスに対して，お互いの利用関係に基づいて，よく利用されているクラスに相対的に高い重要度をつけ，全部品をランク付けする．このランキングで上位に来ている部品は遍在モジュールの候補と考えられる．

本研究では，コンポーネントランク法によって得られる重要度を利用して遍在モジュールを特定する手法を実験により評価，決定し，その手法によって，従来のクラスタリング・アルゴリズムが改良されることを示す．また，その新たな遍在モジュール特定手法を組み合わせた，Java クラスに対するソフトウェア・クラスタリング・システムを提案し，そのシステムを試作する．

主な用語

ソフトウェア理解
ソフトウェア部品再利用
ソフトウェア・クラスタリング
コンポーネントランク法
遍在モジュール（omnipresent module）

目 次

1 まえがき

ソフトウェア保守を行なう際，ソフトウェアのシステム設計や各機能を理解することは必須である [33]．また近年，短期間で大規模かつ安定したソフトウェアを開発するための技術として注目されているソフトウェア部品の再利用の際にも，再利用したい部品の設計だけでなく，その部品がどのようにして用いられているか等，周辺の部品についての理解も必要となる [15]．この様に，ソフトウェア理解が必要となる場面は多々あるが，仕様書等の不備や，ソフトウェアの巨大化，複雑化等の原因でシステム全体を理解することは非常に困難な場合がある．

ソフトウェア理解を助ける上で，様々な手法が研究されているが，その一つにソフトウェア・クラスタリングがある．ソフトウェア・システムは，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）から構成されるが，これらの部品を意味のあるサブシステムに分類してシステムの大まかな構成を示すことによりソフトウェア理解を容易にする手法であり，ソースコードから自動的にソフトウェア・クラスリングを行なうための手法が過去に様々研究されている [19,31] ．以下，ソフトウェア・クラスタリングの際，分類の対象となる部品の単位を「モジュール」と呼ぶ．一般的には，手続き型言語の場合ファイル，オブジェクト指向の場合クラス・インターフェースの単位をモジュールとする．

サブシステム分解を行なう際，その分解の基準・方針は，各クラスタリング・アルゴリズムの細かい目的によって異なるが，その中の一つに，各サブシステムが強凝集・低結合になるようにサブシステム分解を行うというものがある．つまり，密に依存しあっているモジュールをサブシステムとしてまとめ，サブシステム間の依存関係はできるだけ少なくするというものである．これは，複数のモジュールが集まって一つのより抽象度の大きい機能を実装しており，その機能の集合としてシステム全体を構成しているという考え方からなるものである．一般的に，ソフトウェアにおける各モジュールの凝集度を高く，結合度を低くすることは，各モジュールに機能が閉じており，システムの信頼性を高めることになるが [24]，これをソフトウェア・クラスタリングに応用することにより，各サブシステムがより大きなモジュールとみなされ，サブシステムに機能が閉じることになり，システム全体が理解しやすくなることになる．

ところが，システム内の全てのモジュールがある特定のサブシステムに属するべきかというと，そうではない．一般的にライブラリと呼ばれるようなモジュールは様々なモジュールから呼び出されていて，特定のサブシステムに含めることが適切でない．言い替えると，このようなモジュールはあらゆるサブシステムに属するべきである．このようなモジュールは「遍在している（omnipresent）」と定義され [23]，遍在モジュール（omnipresent module）と呼ばれている．ソフトウェア・クラスタリングを行なう際に，遍在モジュールをある特定

のサブシステムに含めてしまうと誤解が生じるため，前もって遍在モジュールを特定し，除去することは重要である．従来手法では，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おり [16,17,21,23,35]，その方法を改善すべく，遍在モジュールを考慮せずクラスタリングを行なった後に依存関係を持つクラスタの数に対して閾値を定める手法 [35] も提案されたが，満足できる結果を得られていない．このように，遍在モジュール特定手法に関しては十分な研究が行なわれていない．

遍在モジュールは，様々なモジュールから直接，または間接的に依存されるモジュールである．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も依存関係に含まれるため，間接的な依存関係が増え，重要になってくる．そこで，Java ソフトウェア部品検索システム SPARS-J [13,39] で用いられている順位づけ手法コンポーネントランク法に着目した．コンポーネントランク法は，あるシステム内のソフトウェア部品（ここでは，クラス・インターフェースを指す）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 [12]．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り [36]，遍在モジュールの満たすべき性質と似ている．

そこで，本研究では，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，決定して，その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．また，その新たな遍在モジュール特定手法を用いた Java クラスに対するクラスタリング・システムを提案し，そのシステムを試作する．

以降，2 節で本研究の背景となるソフトウェア・クラスタリングについて説明し，3 節では，本研究に大きく関わる関連研究である，ソフトウェア・クラスタリング・システム Bunch とコンポーネントランク法について述べる．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を評価・決定するための実験を行ない，決定された新たな遍在モジュール特定手法を提案する．また，5 節で，その新たな遍在モジュール特定手法を組み合わせた，Java を対象としたソフトウェア・クラスタリング・システムを提案し，試作する．最後に 6 節で，本研究の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・システムは，多くのモジュール（関数単位，ファイル単位，クラス単位等，いろいろな単位の取り方が考えられる）から構成されている．ソフトウェア開発現場において，構造のわからないソフトウェア・システムの構造・挙動等を理解しなければならない場面は多々存在するが，巨大なソフトウェア・システム全体を理解することは非常に困難である．

例えば，図書館では大量の本が分野ごとに分類されて配置されているように，大量のものを分類してまとめること（クラスタリングと呼ばれる）は，全体の構造を把握したり，欲しいものを見つけるために非常に有用である．そこで，ソフトウェア・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て，ソフトウェアの理解を助けようという，ソフトウェア・クラスタリングという手法が研究されてきた．

どのような基準でモジュールを分類するかという点に関しては，例えば，ファイル名に基づいて分類するようなアルゴリズム [6] や，ファイルの所有者に注目したアルゴリズム [7] のように，知識に基づいて分類する手法も考案されたが，大部分は，システムの構造に基づいて分類するというものである．

2.2　システム構造に基づいたソフトウェア・クラスタリング

ソフトウェア・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を用いて表される．ここでは，一般的に，モジュールは 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・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，必要ならば辺に重みをつけることもできる．

このようなグラフは，モジュール依存グラフ（MDG: Module Dependency Graph）[21] や，部品グラフ（Component Graph）[13] などと呼ばれ（以下，依存関係グラフと呼ぶ），ソフトウェア理解の上で役に立つが，モジュールの数が多い場合には関係を追うのが大変になり，理解に役立たなくなる．そこで，この依存関係グラフを何か意味のある方針に従って複数の部分グラフ群に分割する．例えば，図 1 のグラフは図 2 のように 3 つのサブグラフ（クラスタと呼ぶ）に分割できる．

図 2 はあくまでも一例にすぎず，図 1 を分割する方法は他にもある．その中から，ソフトウェア理解を助けるような「良い分割」を提示することが，ソフトウェア・クラスタリング

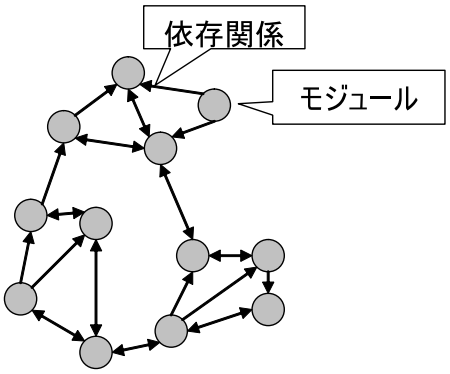


図 1: 依存関係のグラフ表現

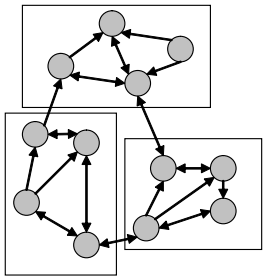


図 2: クラスタリングの一例

であるといえる．

一般的に，任意の頂点は，必ず一つのクラスタに含まれるように分割される．つまり，ある頂点が 2 つのクラスタに同時に含まれるような場合は考えない．この場合，頂点数 n のグラフ G を分割する方法は何通りあるかを考える．G を k $(1 \leq k \leq n)$ 個のクラスタに分割する場合の数を $S(n,k)$ とすると，この値は次のように再帰的に定義され，第 2 種スターリング数（Stirling numbers of the second kind）と呼ばれる．

$$G_{n,k} = \begin{cases} 1 & (k = 1 \text{ または } k = n) \\ S(n-1,k-1) + kS(n-1,k) & (1 < k < n) \end{cases}$$

よって，G の分割方法の場合の数を B_n とすると，次のように定義される．

$$B_n = \sum_{k=1}^{n} S(n,k)$$

この数は，n 番目のベル数（nth Bell number）と呼ばれる数で，n が増えるにつれて，$B_1 = 1, B_2 = 2, B_3 = 5, B_4 = 15, B_5 = 52, \ldots, B_{10} = 115975, \ldots, B_{15} = 1382958545$ というように指数的に

増加する（$O(n!)$ である [27]）．現実のソフトウェアのモジュール数は，これらの例よりはるかに大きい場合が多いため，可能な分割の数は天文学的数字になることがわかる．つまり，ソフトウェア・クラスタリングは，これだけ膨大な分割の中から「良い分割」を探し出す問題となり，一般的に，NP 困難である．

では次に，システム構造に基づいて，どのようにソフトウェア部品を分割するかという方針について，過去に提案されたものをいくつか紹介する．

まず，広く研究されている手法に，高凝集・低結合なクラスタリングがある．これは，モジュールにおける凝集度（cohesion）と結合度（coupling）という 2 つのトレードオフであるメトリクスをクラスタに応用した考え方であり，クラスタ内には依存関係がたくさんあり，クラスタ間には依存関係が少なくなるように，クラスタリングをするという手法である．Schwanke の提案した ARCH [25, 26] では計算量が多く，巨大なシステムには向かなかったが，Mancoridis らによって提案された Bunch というシステムでは，クラスタリングを最適化問題として扱い，山登り法や遺伝的アルゴリズムなどのメタヒューリスティックを用いることで，実用時間で解くことを可能としている [9, 18, 19, 21].

Tzerpos らが提案した ACDC（an Alogorithm for Comprehension-Driven Clustering）[31] では，巨大なシステムによく見られる 7 つのサブシステム・パターンを提案し，それらを実際のシステムに適用することでクラスタリングを行なっている．例をあげると，同じディレクトリにあるモジュールは同じサブシステム内にあるという “Directory structure pattern” や，多くのプログラミング言語が一つの手続きを 2 つのファイル（C の場合，.c ファイルと.h ファイル）に分けるような設計になっているので，それらは 1 つのサブシステムにまとめる “Body-header pattern” などである．

Periklis Andritsos らが提案した LIMBO（scaLable InforMation BOttleneck）[5] では，システム構造と，その他の情報（ファイルパスやファイルの開発者）を組み合わせて，クラスタリングを行なっている．そのアルゴリズムには，情報理論分野における情報ボトルネック法（Information Bottleneck Method）[28] を利用している．クラスタリング結果を C, システムの特徴（構造情報とその他の情報）を B とした時に，相互情報 $I(B;C) = H(B) - H(B|C)$ を最大化する．ここで，$H(B)$ は B の持つエントロピー（情報量），$H(B|C)$ は，C を条件とする B の条件つきエントロピー（C を知った上での，B の持つ情報量．$H(B)$ より少ない）を表す．つまり，$H(B) - H(B|C)$ は，C を知った時に，どれだけ B の持つ情報量（不確かさ）を減らせるかを表す [38]．この方法の場合，同モジュールへ依存関係があるモジュールや，同じ開発者やパスを持つソフトウェア部品がまとめられることになる．

Andreopoulos らは，静的情報・動的情報を組み合わせた MULICsoft を提案している [4]. このアルゴリズムでは，各モジュール（ファイル）の静的な依存関係とともに，その依存

関係（手続き呼び出し）が，実行中に何回起こったかを測定し，依存関係に重みづけを行なっている．さらに，各クラスタは内部に階層構造を持ち，そのクラスタを特徴付けるモジュールがクラスタの上位層に来るようなアルゴリズムになっている．クラスタのまとめ方は，LIMBO と同じく，依存しているモジュール集合が似ているものをクラスタとしてまとめる方針である．

このように，過去に様々な手法が提案されており，それぞれ，目的とするところが微妙に異なる．本研究では，ソフトウェア保守やソフトウェア再利用のためのソフトウェア理解を目的として，高凝集・低結合なクラスタリング手法に注目した．ソフトウェア保守の際には，大まかなシステム構造を知ると同時に，各モジュールの細かい挙動の知識も必要になる．ソースコードの挙動を目で追う場合，あるモジュールが様々な部品を呼び出していると，どこまで追えばいいのかわからなくなることがあるが，依存関係が密なサブシステムが提示されている場合，サブシステム単位で理解することが可能になる．また，ソフトウェア部品再利用に関しても，あるソフトウェア部品を再利用したいと思った時，その部品がどのように利用されているのか，どのような部品と組み合わせて使われているのかが理解しやすくなり，また，サブシステム単位での再利用というものも可能になる．

2.3 遍在モジュール

では，全てのモジュールに対して，高凝集・低結合なサブシステムへの分類を適用すればよいのかといえば，そうではない．この原則を当てはめるべきでない，遍在モジュール（omnipresent module）と呼ばれるモジュールが存在する Müller は，依存関係グラフにおけるモジュールの次数（つまり，そのモジュールが依存関係を持つモジュールの数）が，ある閾値以上である場合，そのモジュールは「遍在している」（omnipresent）と定義している [23]．遍在モジュールは，例えば，ユーティリティやライブラリと呼ばれるようなモジュールを表している．このようなモジュールは，図 3 の様に，様々なモジュールから使用されているため，特定のサブシステムに属さないと考えられる．言い換えると，あらゆるサブシステムで用いられるので，あらゆるサブシステムに属している（遍在している）ともいえる．

遍在モジュールを特定のサブシステムに入れてしまうと，他の多くのサブシステムの結合度が上がってしまうため，遍在モジュールは，サブシステムとは独立した存在として考える必要がある．そのため，ソフトウェア・クラスタリングを行なう前に，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おいて，クラスタリングを行なうアルゴリズムが多い [16, 17, 21, 23, 31] [1]．

ところが，遍在モジュールに対する厳密な定義は存在しないため，システムによって，閾

[1] Alan MacCormack らは遍在モジュールと同じ概念としてバス（bus）[32] という単語を用いている [17].

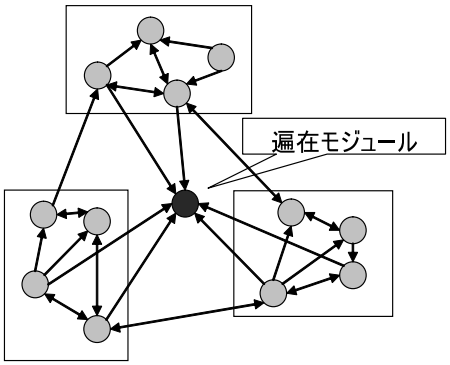


図 3: 遍在モジュールの例

値の定め方は異なる．全頂点数の何%（初期値 10%）という決め方 [17] や，頂点数の平方根を取ったもの [16] や，平均次数の何倍（初期値 3 倍）[21] という決め方がある．しかし，遍在モジュールの特定法に関して詳細な調査をしている例は少なく，現状の特定方法では問題点もある（詳しくは 3.1 節で述べる）．

Zhihua Wen らは，遍在モジュール特定法を改良することによって，クラスタリングの結果を改善する研究を行なっている [35]．彼らは，次数が多いモジュールを遍在モジュールとみなすのでは無く，さまざまなクラスタから使用されているモジュールこそが遍在モジュールだという論拠のもと，ソフトウェア・クラスタリング・アルゴリズム改良フレームワーク FICABOO （Framework for the Improvement of Clustering Algorithms Based on Omnipresent Objects）を提案している．このフレームワークは，次のような方法で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目指している．

1. 従来のソフトウェア・クラスタリング・アルゴリズム A を用いて，分割結果 d_0 を得る（この際，遍在モジュールは考慮しない）
2. d_0 より，各モジュールが依存関係のあるクラスタ数（クラスタ次数）を求める
3. クラスタ次数を元に，遍在モジュールを特定する（閾値の決定方法は数種類用意して実験している）
4. 遍在モジュールを除去したグラフを用いて，A を用いてクラスタリングを行ない，解 d_1 を得る
5. Orphan Adoption [30] を用いて遍在モジュールを d_1 に追加する

論文中では，FICABOO を前述した Bunch, ACDC, LIMBO に適用しているが，結果としては改良できた場合もあれば，できなかった場合もある．理由としては，最初に遍在モジュー

ルを考慮せずクラスタリングを行なう時点で，遍在モジュールが邪魔をして適切なクラスタリングが行なえていない場合があると考えられる．

このように，現状では，遍在モジュール特定に関しては，十分な研究が行なわれておらず，改良の余地がある．そこで，本研究では，別のメトリクスを用いた新たな遍在モジュール特定方法を提案し，ソフトウェア・クラスタリングの質を向上することを目的とした．

3 関連研究

この節では，本研究に深く関連する 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なうシステムとして，現実的な実行時間で計算を行なえるシステムが，Mancoridis，Mitchell らによって開発された Bunch [21] である．Bunch では，フトウェア・クラスタリング問題を探索問題として解く．そのためには，まず，考えられる解（分割）を評価するための目的関数（objective/fitness function）を定義する必要がある．そして，考えられる解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ればよいが，2.2 節で述べたように，考えられる分割の数は膨大なため，全ての分割に対して目的関数値を求めることは現実的ではない．そこで，メタヒューリスティック（最適解を得られる保証は無いがそれに近い解を求めるためのアルゴリズムをヒューリスティック（heuristic）と呼び，その内，問題に依存せずに利用できるアルゴリズムをメタヒューリスティック（metaheuristic）と呼ぶ）を用いて近似解を求める．以下に詳細を述べる．

3.1.1 目的関数 *MQ*

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次のような定義である．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 $(1 \leq i \leq k)$ 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．

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

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．全てのクラスタの CF の合計が MQ となる．

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

MQ の定義に関しては，いくつか提案されており [19]，初期に用いられていたものは異なる定義であったが，現在ではこの定義が標準的に用いられている．また，この MQ の定義によって，実際のシステムに対して正しく評価できることが確認されている [10].

3.1.2 メタヒューリスティックの利用

近似解を求める手法として，まず，古典的な局所探索法である山登り法（hill climbing）によって実装された [18]．これは，まずランダムに初期状態（ランダムに分割したもの）を用意して，その近傍（ここでは，ある 1 つの頂点を別のクラスタに移動したもの）の中で MQ 値を改善する状態へ移り（MQ を改善する状態が複数ある場合，どの状態に移動するかはパラメータによる），MQ が改善されなくなると，その状態を解とする方法である．しかし，山登り法の欠点として，初期状態によっては局所解に陥る可能性がある．それをできるだけ回避する方法として，初期状態を複数用意する方法と，焼きなまし法（simulated annealing）[14] を利用する方法が実装されている [19, 21]．焼きなまし法は，目的関数の改善のみではなく，改悪も許すことで局所解への収束を防ぐ手法である．改善・改悪は確率的に決まり，時間とともに改悪する確率を下げることで，最適解への収束を目指す．

山登り法のほかに，局所探索ではない，遺伝的アルゴリズム（genetic algorithm）[22] を利用して実装されているが [9]，現状としては，山登り法での実装に比べて解のばらつきが大きかったり，実行時間の面でもばらつきが大きく，山登り法による実装が標準として用いられている．

また，Sholoufandeh らは，メタヒューリスティックではなく，数学的なアプローチであるスペクトル法（spectral methods）[29] を Bunch に適用した [27]．スペクトル法は，ラプラシアン行列で表したグラフを 2 分割するために用いられる手法で，MQ を最大化するような 2 分割を再帰的に繰り返すことでクラスタリングを実装している．結果としては，山登り法による実装と比較して，解の MQ はわずかに小さくなっているが（つまり，解の質が少し悪い），メタヒューリスティックでは無いため，解は安定している．実行時間に関しては，計算量が $\Theta(n^4)$ のため，大規模なシステムになると，山登り法と比べて非常に大きくなってしまっている．筆者らはまだ改善の余地があるとしているが，この結果は，山登り法による実装の有用性を示しているともいえる．

3.1.3 階層的なクラスタリング

Bunch では，まず，初期状態がランダムに選ばれる．つまり，ベル数の数だけある分割方法の中からランダムに選ばれた分割が初期状態になる．そして，その状態で，任意の頂点 1 つだけを別のクラスタに移動（新たなクラスタを作ることも考える）した分割がいくつか考えられるが，その分割群を近傍分割（Neighboring Partitions）と呼ぶ．近傍分割の中には，MQ を改善するものもあれば改悪するものもあるので，（焼きなまし法を適用していない場合には，）MQ を改善する状態へ移動する．ここで，MQ を改善する状態が複数ある場合に，どの状態へ移動するかは，パラメータによって調節できる．一番初めに見つけた MQ を改善

する状態へ移動する NAHC（Next Ascent Hill Climbing）と，全ての近傍分割を調べて最も MQ が改善される状態へ移動する SAHC（Steepest Ascent Hill Climbing）を両極端として，近傍分割のうち最低何%を調べるかという値を設定できる．デフォルトでは NAHC が採用されているが，これは，近傍分割の数は多いため，全てを調べていると時間がかかりすぎるからである．

あとは，同じようにして MQ が改善する移動を繰り返し，MQ が改善されなくなれば一つの段階が終了する．次に，この状態でできた各クラスタを一つのモジュールとみなして，同じようにクラスタリングを行なう．この作業を繰り返し，最終的にはクラスタ数が一つになるまで繰り返す（MQ の性質的にクラスタが一つしかなければ $MQ = 0$ となって改善されないので最後は MQ を気にせず併合することになる）．よって最終的には，階層的にクラスタリングされた結果が得られる．デフォルトでは，この複数層の内，ちょうど真ん中の層が出力されるが，全ての層を出力することや最下層を出力することも可能である．

3.1.4 Bunch における遍在モジュールの扱いと問題点

Bunch では，遍在モジュールを考慮することが可能である．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．

- omnipresent supplier：入次数が平均次数の 3 倍以上
- omnipresent client：出次数が平均次数の 3 倍以上
- omnipresent central：入次数・出次数ともに，平均次数の 3 倍以上
- library：出次数が 0

なお，厳密には，omnipresent supplier，omnipresent client，omnipresent central（以下，それぞれ単に supplier，client，central と表記）の 3 種を遍在モジュールとして定義し，library は別扱いであるが，本論文では，記述の簡略化のため，library も遍在モジュールの 1 つとして表現する．また，supplier，client，central の特定条件に含まれる倍率は，デフォルトとして 3 となっており，変更可能である．なお，遍在モジュールの特定される優先順位は，library>central>supplier=client であると思われる．

また，上に挙げた特定条件以外にも遍在モジュールとして特定される特殊な場合も存在する．それは，遍在モジュールにのみ依存関係があるモジュールは遍在モジュールとなる，というものである．例えば，図 4 のように，モジュール a が supplier であると特定されており，a にのみ依存するモジュール b が存在した場合，b が，出次数が平均次数の 3 倍以上という条件を満たしていない場合でも自動的に client とみなされる．

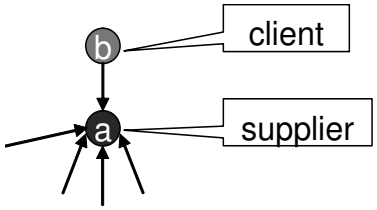


図 4: 遍在モジュールに特定される特殊な例

なお，現在の Bunch には，この処理に関してバグが含まれており，遍在モジュールの選び方（遍在モジュールはユーザが指定することもでき，supplier と client の和集合が central に一致しない場合）によっては，遍在モジュールが取り除かれた後に孤立したモジュールが遍在モジュールに選ばれず，クラスタリング対象にもならない（つまり，クラスタリング結果に現れない）例が存在するようである．

ここで，Bunch の遍在モジュール特定法に関して考えらる問題点をいくつか挙げる．まず，特定条件として使うメトリクスが次数のみで単純である．例えば，図 5 のような例を考える．

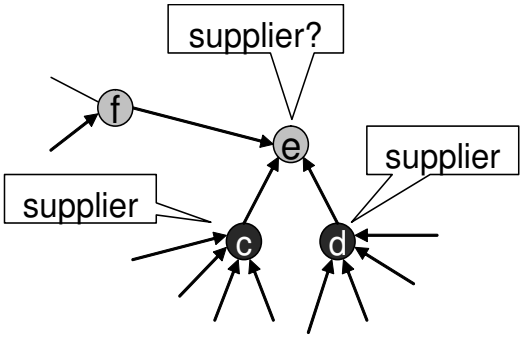


図 5: 遍在モジュールを次数のみで特定できない例

図 5 では，c, d が入次数によって supplier として特定されており，e, f は supplier として特定されてないとする．この場合，c, d は事前に除去されてからクラスタリングが行なわれる．よって，e は f と同じクラスタに入る可能性が高い．しかし，supplier である c, d から依存されている e は，特定のクラスタに入れるのではなく，supplier とした方が望ましい．特に，オブジェクト指向言語で c, d が e を継承している場合など，このような例が存在すると考えられる．

また，別の点として，libarary の条件が緩いという点がある．出次数が 0 というモジュールの数は多く，システムによっても異なるが，全モジュールの 10~30%が library として抽出され，結果的に全モジュールの 30~50%が遍在モジュールとして除去されている（詳しくは

4 節で述べる）．これだけのモジュールがクラスタリング対象から除かれていたのでは，ソフトウェア理解を助ける効果が弱くなってしまうと思われる．libarary の特定条件を考え直す（もしくは library を考慮しないようにする）必要がある．

3.1.5 フレームワークとしての Bunch

以上のような手法を実装している Bunch であるが，これさえあれば自動的にソフトウェア・クラスタリングができる，というアプリケーションの立場を取らずに，ソフトウェア・クラスタリングを行なうためのコアとなるフレームワークという立場で開発，公開されている [1].

例えば，Bunch への入力は，ソースコードではなく，ソースコードを分析して作成された依存関係グラフ（Bunch では MDG（Module Dependency Graph）という単語を使っている）を与える．こうすることによって，Bunch の言語依存を無くすとともに，ソフトウェア部品（モジュール）の単位が何であるか，依存関係が何であるかという定義もユーザが自由に決めることができる．一般的には，Acacia [8] などのソースコード分析ツールを使うと良い．

ソフトウェア・クラスタリングの実行部分に関しては，アルゴリズムの変更や，閾値・パラメータの変更ができ，また，遍在モジュールを自由に特定できる．また，このモジュールとこのモジュールは同じクラスタに入れるべきという，ユーザの知識を入力することも可能となっている．

出力に関しては，独自のテキスト・フォーマット，DOT フォーマット，GXL（Graph eXchange Language: XML によるグラフ・フォーマット）フォーマットが選択できる．DOT フォーマットに関しては，Graphviz [3] に含まれるプログラム `dotty` で表示でき，プログラム `dot` でフォーマット変換などの処理を行なえる．GXL フォーマットで出力した場合，Clusternav [2] というツールを使えば，クラスタを開いたり閉じたりしながら閲覧が可能である．図 6 は，GNUJpdf という Java で書かれたオープンソース・ソフトウェアに対して Bunch でクラスタリングを行なった結果である．DOT 形式で出力した後，パッケージ名（`gnu.jpdf`）を除去して，一部クラスタの場所を移動している．図の下部に，suppliers，clients，library が出力されているが，`PDFOutput` は，library と supplier にのみ依存関係を持つクラスのため，特殊な場合として client として特定されていることがわかる（普通の遍在モジュールに関しては，辺は全て省略されていうる）．上部に残りのクラスをクラスタリングした結果が表示されているが，全部で 3 層になっていることが確認できる（最上層はシステム全体を表すクラスタで，表示されていない）．

また，Bunch では API が `jar` ファイルという形で公開されており [1]，それを利用することでソフトウェア・クラスタリング・システムを構築することが可能になっている．API で

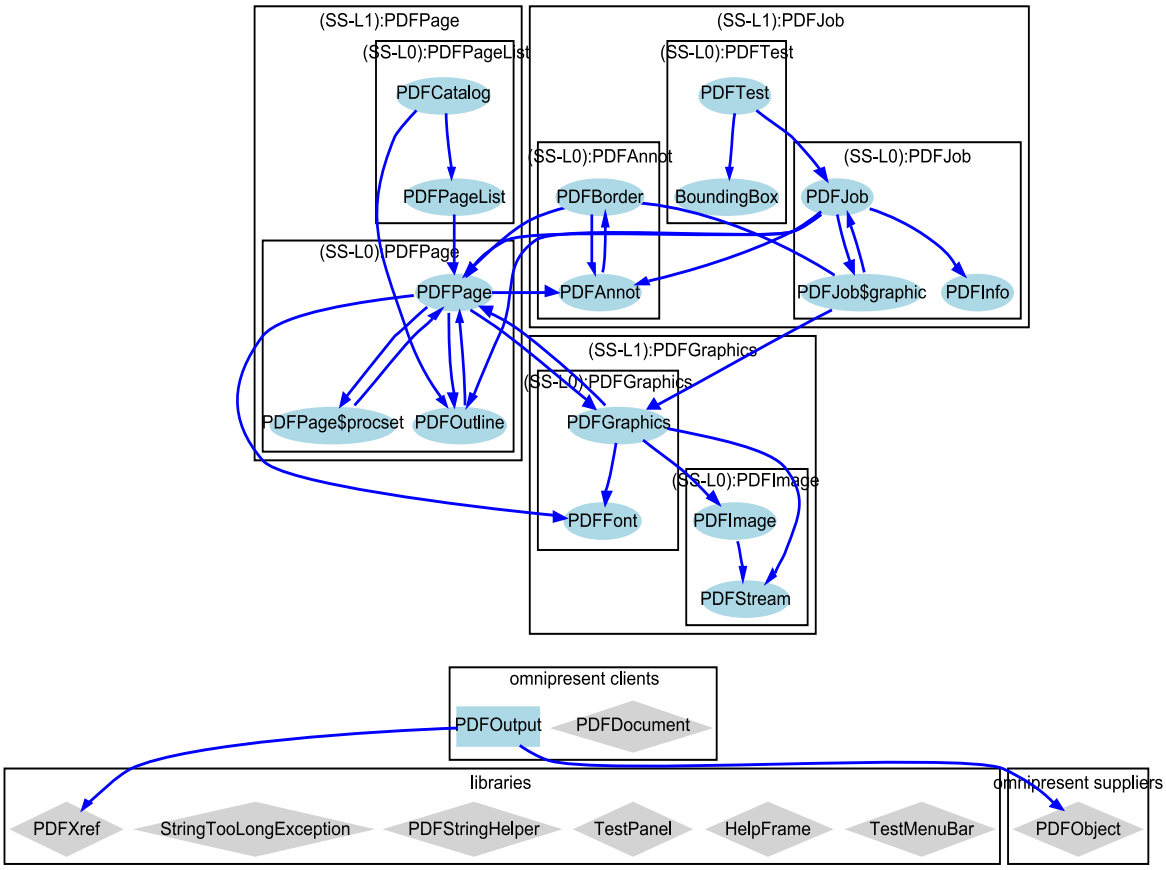


図 6: Bunch によるクラスタリング結果の例

は，各種パラメータ等を変更したり，遍在モジュールを指定できたり，結果を細かく分析することが可能である．本研究では，Bunch API を利用して，実験，システム構築を行なっている．

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（Software Product Archive, analysis and Retrieve System for Java）が開発されている [13, 39]．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・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（Component Graph と呼ばれる）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 7 に示す．

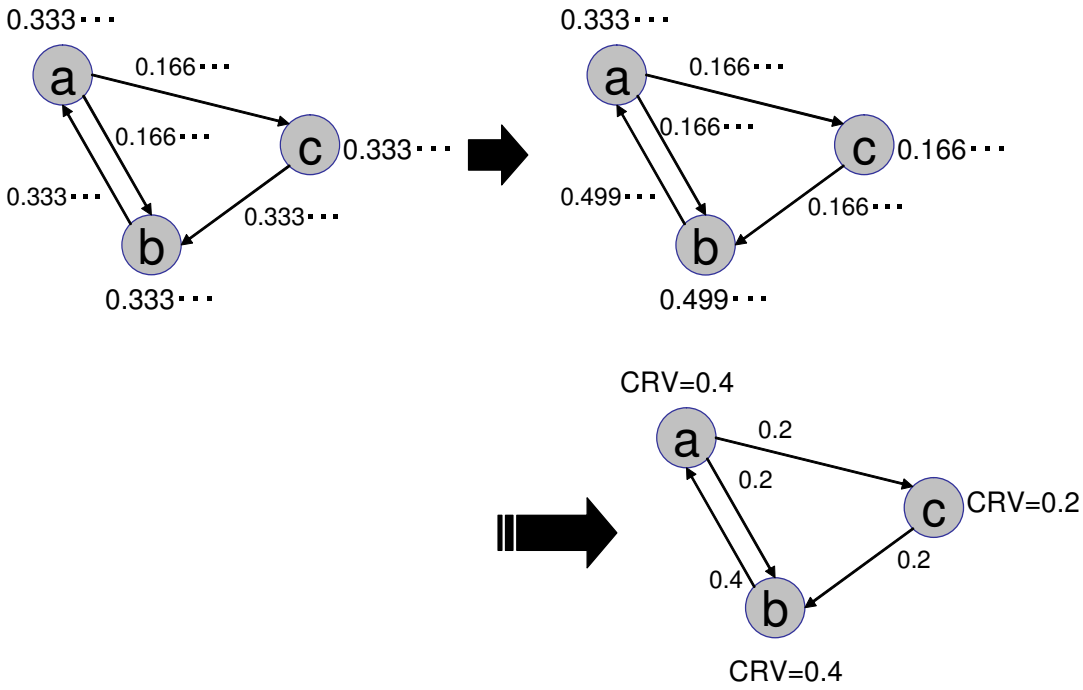


図 7: CRV 計算の例

この図では，頂点数が 3 のため，重みの初期値はそれぞれ 0.333･･･ となり，それを各辺に分配することで，各頂点の新たな重みが決定している．最終的には右下のような重みに収束し，この値を CRV とする．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．

一般的なソフトウェア・システムの依存関係グラフでは，多くの部品の次数は低く，一部の部品の次数が非常に大きいという性質（べき乗則）が成り立つことが経験的・実験的にわかっており [37]，そのため，一般的に CRV の分布も同様に，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 が非常に高い部品を見ることで，システム内で重要な部品が特定できることが調査されている [36]．

図 5 で示したモジュール e のように次数のみでは遍在モジュールと特定できないモジュールに関しても，CRV は高くなっているので，CRV を用いることで特定することが可能にな

る．本研究では，遍在モジュールのうち，supplier，central，client は，CRV が高いはずであると仮定し，遍在モジュールを特定する条件に CRV を使用することで，ソフトウェア・クラスタリングの質を向上させることを目指す.

4　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

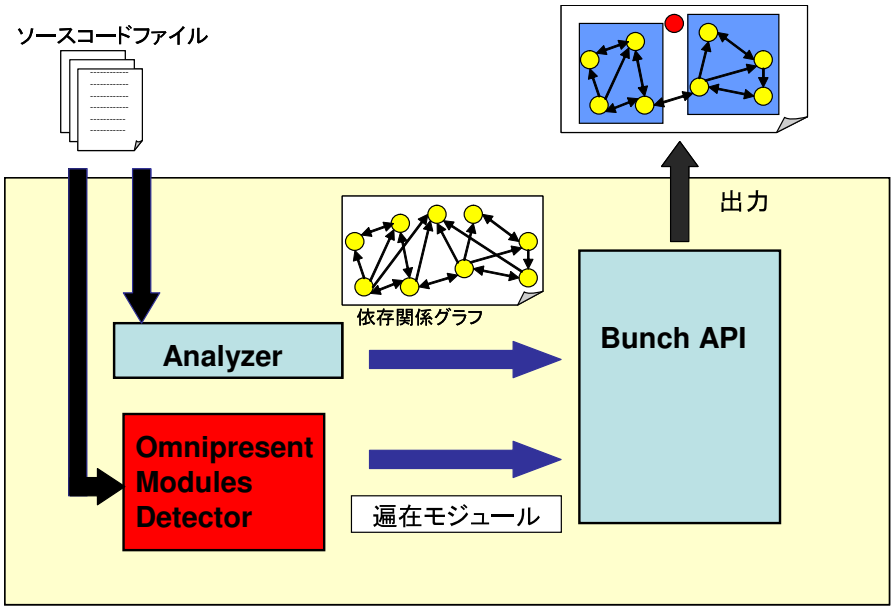


図 8: Bunch API を利用したクラスタリング・システムの構造

図 8 は，Bunch API を利用したクラスタリング・システムの一般的なシステム構造である．Bunch API は，依存関係グラフを入力として，クラスタリング結果を解として出力するためのアルゴリズムが実装されている．その際，遍在モジュールを特定して入力することが可能となっており，この入力を変えることで，当然，解も違ったものになる．すなわち，遍在モジュールを適切に選ぶことで，より優れた解を出力することが可能となる．

本節では，コンポーネントランク法によって得られる各モジュールの重要度 CRV を用いて遍在モジュールを特定することによって，解を改良できるか，また CRV を用いるといっても，どのように用いればよいか，という問題に対して，実験を行なうことによって解を示す．

まず，CRV と次数を用いた遍在モジュール特定条件として考えられるものを定義して，それらの条件のもと，解がどのように変化するかを 2 種類の実験によって評価し，最後に，実験の結果として得られた適切な条件を示す．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~C47 の遍在モジュール特定条件を実験対象として定める．各行が 1 つの条件を表し，supplier，client，central，library の特定条件をそれぞれ提示している．in，out，CRV はそれぞれ入次数，出次数，CRV を表し，数字は平均値からの

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	out	CVR	in	out	CVR	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	3			0	
C8			1		3		3	3	1		0	
C9			1		3		3	3	1			
C10			1		3		3	3	1		0	1
C11			1		3		3	3	1	1	0	1
C12			1		3		3	3	1	1	0	
C13			1		3			3	1	1	0	1
C14	1		1		3		1	3	1		0	
C15	1		1		3		1	3	1		0	
C16	1		1		3		1	3	1			
C17	1		1		3		1	3	1		0	1
C18	1		1		3		1	3	1	1	0	1
C19	1		1		3			3	1	1	0	1
C20	1		1		3		3	3	1	1	0	1
C21	1.5		1		3		1.5	3	1		0	
C22	1.5		1		3		1.5	3	1		0	
C23	1.5		1		3		1.5	3	1			
C24	1.5		1		3		1.5	3	1		0	1
C25	1.5		1		3		1.5	3	1	1	0	1
C26	1.5		1		3			3	1	1	0	1
C27	1.5		1		3		3	3	1	1	0	1
C28	2		1		3		2	3	1		0	
C29	2		1		3		2	3	1		0	
C30	2		1		3		2	3	1			
C31	2		1		3		2	3	1		0	1
C32	2		1		3		2	3	1	1	0	1
C33	2		1		3			3	1	1	0	1
C34	2		1		3		3	3	1	1	0	1
C35	2.5		1		3		2.5	3	1		0	
C36	2.5		1		3		2.5	3	1		0	
C37	2.5		1		3		2.5	3	1			
C38	2.5		1		3		2.5	3	1		0	1
C39	2.5		1		3		2.5	3	1	1	0	1
C40	2.5		1		3			3	1	1	0	1
C41	2.5		1		3		3	3	1	1	0	1
C42	3		1		3		3	3	1		0	
C43	3		1		3		3	3	1		0	
C44	3		1		3		3	3	1			
C45	3		1		3		3	3	1		0	1
C46	3		1		3		3	3	1	1	0	1
C47	3		1		3			3	1	1	0	1

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．つまり，C1 の場合，supplier である条件は入次数が 3 倍以上，library である条件は出次数が 0 であることを表す．C1 の central のように条件が複数あるものは，全てを満たしていなければいけない．また，例えば，C2 の supplier は条件が全く記述されていないが，これは supplier を全く特定しないことを示す．

C1 が 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例であり，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．5 節で述べたように，CRV が平均を超えるモジュールは少ないため，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 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，基本的に 2 つの条件を足したような条件にしている．library に関しては，出次数 0 だけでは条件が緩すぎると判断し，それに加えて，CRV 平均以上や，入次数平均以上という条件を加えている．なお，いずれの条件に関しても，各遍在モジュールの優先順位は，library，central，supplier，client とした．

本実験では，これら 47 個の特定条件を評価するために 2 種類の実験を行なった．以下にその詳細を述べる．

4.2 実験 1：平均 CF 値と遍在モジュール数による評価

4.2.1 方針

クラスタリング・ソフトウェア・アルゴリズムは，アルゴリズムによって目的が異なり，Bunch の目指すものは，各クラスタが高凝集・低結合になっていることであった．3.1.1 節で示したように，Bunch では，各クラスタの凝集度と結合度のトレードオフを CF という 0 ~ 1 の値をとるメトリクスで表し，各クラスタの CF の合計値を MQ というメトリクスで数値化している．つまり，同一システムの場合，MQ の値が高いほど，クラスタリング結果の質（理解しやすさ）が高いといえる．しかし，基本的には，モジュールの数が多い程，MQ 値は大きくなりえるので，異なるシステムに対するクラスタリング結果を MQ 値によって比較することはできない．つまり，例えば遍在モジュール特定条件 C1 と C2 の結果を比べる際に，MQ の値が高い方が優れているとはいえないということである．なぜなら，C1 と C2 では特定する遍在モジュールの数が異なるため，遍在モジュールに特定されずクラスタリン

グ対象となったモジュール（以下，このモジュールを「クラスタリング対象モジュール」と呼ぶ）の数が異なるからである．

そこで，本実験では，平均 CF 値に注目した．平均 CF 値は 0 ~ 1 の値を取り，この値が高い程，クラスタリング結果の質が高いことは明らかである[2]．ところが，平均 CF 値のみでは，正当に評価することは不可能である．例えば，クラス数 100 個のシステムで，依存関係のある 2 つのクラスを除いて，残り 98 個を遍在モジュールとみなした場合，遍在モジュールは CF の計算に含まれないため，2 つのクラスを 1 つのクラスタに入れて，平均 CF が 1.0 ということになってしまう．この問題は，3.1.4 節で示した，Bunch での遍在モジュール特定手法の問題点の 1 つである，遍在モジュールの数が多過ぎるという点にも関連する．つまり，遍在モジュールをたくさん特定した方が，平均 CF の値が大きくなってしまう場合が考えられるということである．よって，クラスタリング対象モジュールの数と，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優れているという判断基準を定める．

本実験では，表 2 に示す 15 個の Java で書かれたオープンソース・ソフトウェア・システムに対して C1~C47 の特定条件を適用した結果を比較する．なお，Bunch は階層的にクラスタリングを行い，各層によって平均 CF 値は異なるが，本実験では，最下層（つまり階層的にクラスタリングを行なわず，1 回だけクラスタリングを行なった結果）の平均 CF 値を使っている．また，局所解へ陥ることを避けるため，各条件に対して，山登り法における初期状態数を 30，さらに焼きなまし法を適用している（ただし，Azureus に関しては，クラス数が非常に多く，現実的な時間で終らなかったため，初期状態 5 で実行している）．

4.2.2 結果と考察

例として，GNUJpdf へ適用した結果を示す．表 3 は，各遍在モジュール特定条件に対する各遍在モジュールの特定された数である．「条件適合」の列が，表 1 の条件に適合した数で，それに，図 4 で示したような孤立したモジュールを遍在モジュールにする特殊な例を加えたものが「実際の数」である．

図 9 は，GNUJpdf において，各遍在モジュール特定手法に対する，クラスタリング対象モジュールの割合（全モジュールのうち，遍在モジュールに選ばれなかったモジュールの割合），平均 CF 値，そして参考としてその 2 つの調和平均[3]をプロットしてある．

図 10 は，表 2 に示した 15 のシステム全てに適用して，平均値をとった図である．図 9 に比べて振れ幅が小さいが，同じような形をしていることがわかる．

---

[2] しかし，Bunch の目的関数として平均 CF を使うことは適切ではない．なぜなら，クラスタ数が増えると平均 CF は下がりやすいので，クラスタ数が初期状態から増加しにくいと考えられるからである．

[3] クラスタリング対象モジュールの割合と平均 CF 値は対等なメトリクスとはいえないので，調和平均で評価をするわけではない

表 2: 実験に用いたシステム

名前	バージョン	クラス数	概要
Azureus	2.5.0.4	2870	BitTorrent クライアント
Dexter	0.4	41	スキャンしたグラフからデータを抽出
EJE (Everyone's Java Editor)	2.3	95	シンプルな Java エディタ
GNUJpdf	1.6	25	PDF を生成・印刷するための package
iText	1.4.8	586	PDF/HTML 文書を生成するためのクラス群
jpcap	0.01.16	115	ネットワーク・パケット・キャプチャ
jVi	0.7.2	127	javax.swing.text による vi/vim のクローン
jwma (Java Webmail)	0.9.7	95	JSP Model 2 によるウェブメール実装
MicroEmulator	1.0	297	Java 2 Micro Edition (J2ME) エミュレータ
PMD	3.9	823	Java ソースコード分析
pollo	0.4	313	XML エディタ
SAX (Simple API for XML)	2.0.2 final	41	XML パーサへのフロントエンド
StringTree Java utility package	2.0.4	325	テキスト変換・処理用 package
uitags	0.6.12	284	JSP 用カスタム・タグ・ライブラリ
Xdoclet	1.2.3	599	Javadoc Doclet 拡張エンジン

また，図 11 は，GNUJpdf において，図 9 で示した各値が Bunch オリジナル条件（C1）からどのくらい増加したかという増加率を示している．図 12 は同様に，15 システムでの増加率の平均値である．

以下，これらの表・グラフを見ながら考察を行なう．なお，3.1.4 節で述べた Bunch のバグ（遍在モジュールを取り除いた後に次数 0 になった頂点が消されてしまう）のため，値が正しくないと思われる箇所がいくつかある．

C1 の結果を見ると，GNUJpdf の場合，クラスタリング対象モジュールの割合は約 0.64，平均 CF 値は 0.51 となっており，15 システムの場合，それぞれ 0.58，0.59 となっている．この 2 つの値をバランスよく改善している条件が優れているといえる．

C3~C5 は Bunch の特定手法の閾値を 2，2.5，3.5 に変えたものだが，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，トレードオフとなっている．GNUJpdf の場合，C3 では平均 CF が最高値の 1.0 だが，クラスタリング対象モジュールが 25 個中わずか 7 個しか無く，適切なクラスタリングとはいえない．C1，C3~C5 を見比べる限りでは，閾値 3 という設定が一番適切であるといえる．

C6 は，C1 と同じ閾値 3 という設定だが，library を考慮しないようにしている．この場合，平均 CF 値は下がっているが，クラスタリング対象モジュール数は約 40%の増加になっており，出次数 0 という条件によっていかに library が多く選ばれているかがわかる．

表 3: GNUJpdf での遍在モジュールの数

No.	supplier		client		central		library	
	条件適合	実際の数	条件適合	実際の数	条件適合	実際の数	条件適合	実際の数
C1	1	1	1	2	0	0	6	6
C2	0	0	0	0	0	0	0	0
C3	1	3	3	6	2	3	6	6
C4	1	3	1	2	2	2	6	6
C5	1	1	0	0	0	0	6	6
C6	1	1	1	1	0	0	0	0
C7	5	6	0	0	0	0	6	6
C8	5	6	0	0	0	0	6	6
C9	6	6	0	0	0	0	0	0
C10	5	5	0	0	0	0	1	1
C11	5	5	0	0	0	0	1	1
C12	5	5	0	0	0	0	1	1
C13	4	4	0	0	1	2	1	1
C14	4	4	0	0	1	2	6	6
C15	4	4	0	0	1	2	6	6
C16	5	5	0	0	1	2	0	0
C17	4	4	0	0	1	2	1	1
C18	4	4	0	0	1	2	1	1
C19	4	4	0	0	1	2	1	1
C20	5	5	0	0	0	0	1	1
C21	3	3	0	0	1	2	6	6
C22	3	3	0	0	1	2	6	6
C23	4	4	0	0	1	2	0	0
C24	3	3	0	0	1	2	1	1
C25	3	3	0	0	1	2	1	1
C26	3	3	0	0	1	2	1	1
C27	4	4	0	0	0	0	1	1
C28	2	2	0	0	1	2	6	6
C29	2	2	0	0	1	2	6	6
C30	2	2	0	0	1	2	0	0
C31	2	2	0	0	1	2	1	1
C32	2	2	0	0	1	2	1	1
C33	2	2	0	0	1	2	1	1
C34	3	3	0	0	0	0	1	1
C35	2	2	0	0	1	2	6	6
C36	2	2	0	0	1	2	6	6
C37	2	2	0	0	1	2	0	0
C38	2	2	0	0	1	2	1	1
C39	2	2	0	0	1	2	1	1
C40	2	2	0	0	1	2	1	1
C41	3	3	0	0	0	0	1	1
C42	1	1	1	2	0	0	6	6
C43	1	1	1	2	0	0	6	6
C44	1	1	1	1	0	0	0	0
C45	1	1	1	1	0	0	1	1
C46	1	1	1	1	0	0	1	1
C47	1	1	0	0	1	1	1	1

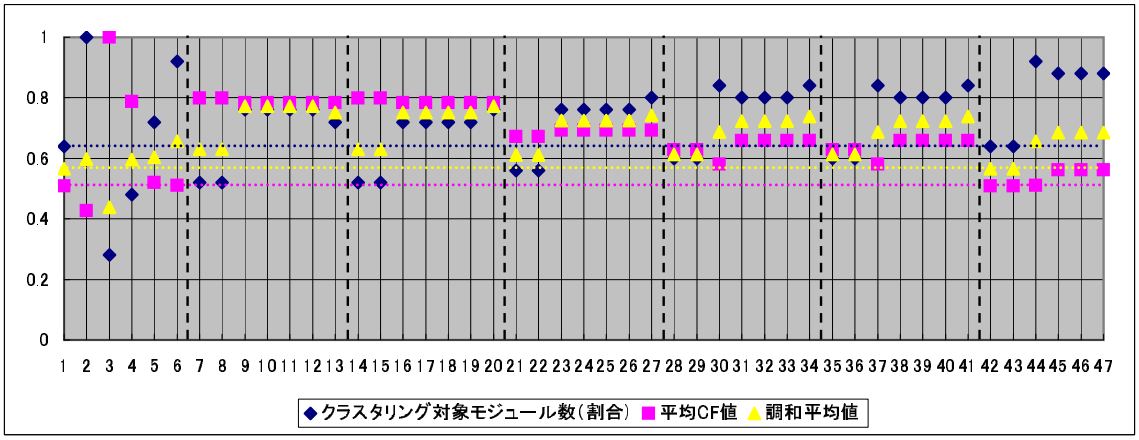


図 9: GNUJpdf における結果

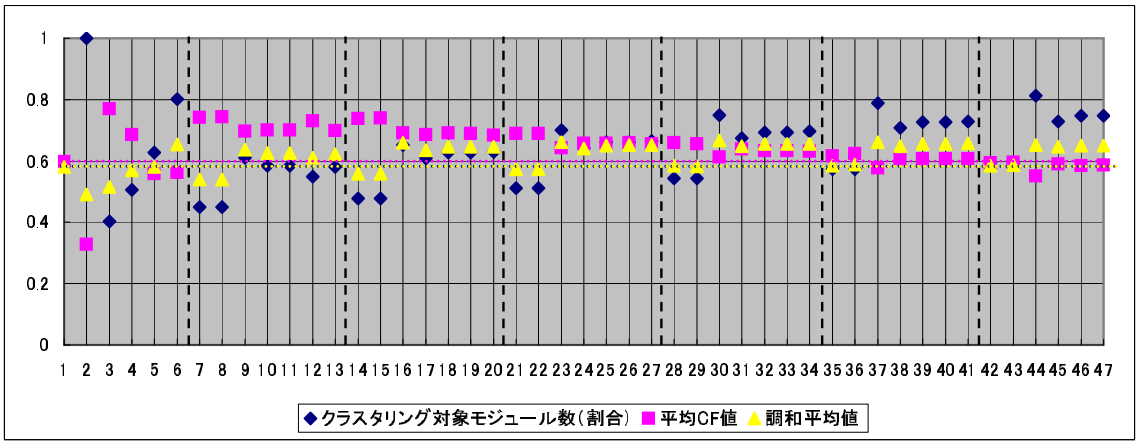


図 10: 15 システムにおける結果

C7~C13 は，supplier の条件として，CRV が平均以上のみを指定し，central や library の条件をいろいろと変えたものである．

C7 は，C1 と比べて supplier の条件を CRV 平均以上に変えただけであるが，クラスタリング対象モジュール数が約 20%減少し，平均 *CF* 値は GNUJpdf で 57%，15 システム平均で 24%増加している．これは，入次数平均 3 倍以上のモジュールよりも CRV 平均以上のモジュールが約 20%多いということであり，supplier の条件として CRV 平均以上のみでは不適切であることがわかる．

C8 は，C7 に比べて，central の条件を少し強くして，入次数平均 3 倍以上，出次数平均 3 倍以上，CRV 平均以上としたものであるが，これは，クラスタリング対象モジュール数，平

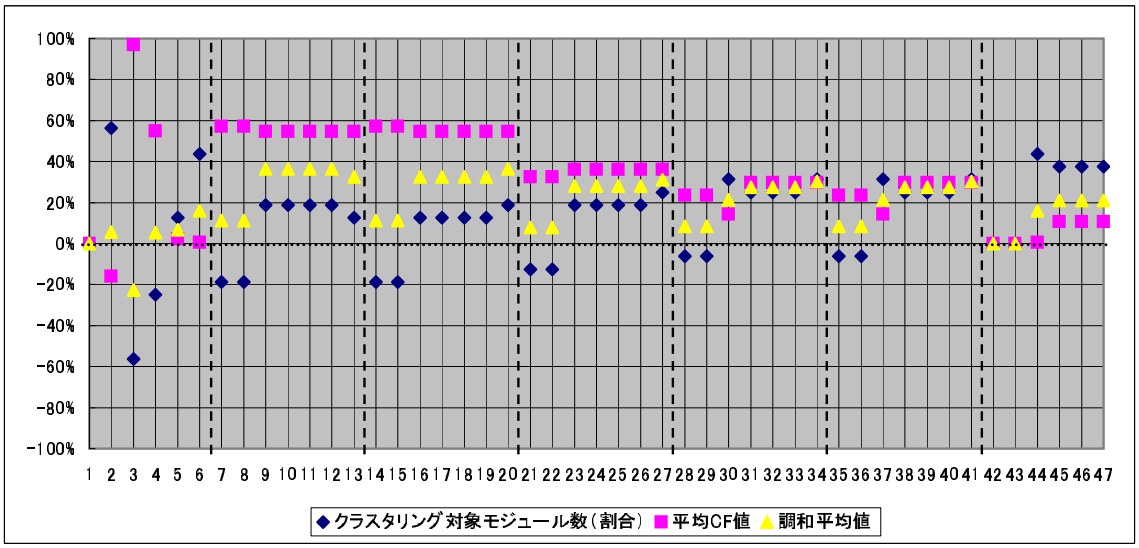


図 11: GNUJpdf における結果（オリジナルからの増加率）

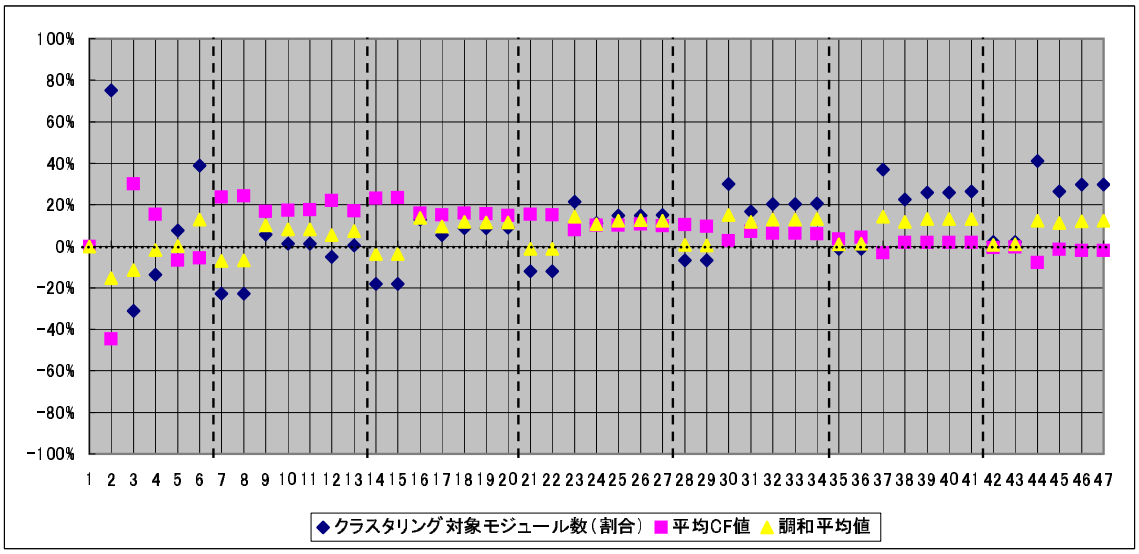


図 12: 15 システムにおける結果（オリジナルからの増加率）

均 *CF* 値ともに C7 と変わらないはずである．なぜなら，C7 と C8 は，入次数平均 3 倍以上，出次数平均 3 倍以上，CRV 平均**未満**のモジュールが central になるか，client になるかという違いしか無いためである．しかし，実際のデータを見ると，15 システムの平均 *CF* 値にわずかに 0.0026 という差が見られる．これは前述した Bunch のバグのためで，C7 も C8 も，central の条件が supplier かつ client になっていないからである．以下，同様の例が多々見られる．

C9 は，C8 から library を考慮しないように変更したものである．GNUJpdf の場合，C8 と比べると，library が 6 個減ったため，クラスタリング対象モジュールの割合が大きく増加し，平均 *CF* 値はほとんど減少していない．C1 と比べても共に増加しており，library を考慮しないことが有効であることを示している．15 システム平均の場合は，クラスタリング対象モジュール数は C1 と比べて 6%しか増加していない．やはり，supplier が多過ぎるようである．

C10~C12 は，library の数を減らすために，条件を新しいものに変えた例である．いずれも，library はある程度の部品から使われてなければならないという考え方を含めたものである．GNUJpdf では違いは見られないが，15 システムの場合，C11（library の条件は，入次数平均以上，出次数 0，CRV 平均以上）がわずかに優れている．

C13 は C11 に比べて，supplier かつ client であるものを central とするように変更したものであるが，C11 とほぼ結果は変わらない．C11 の結果には，やはり Bunch のバグを含んでいる可能性が高いので，正しい評価は難しい．

C14~C47 は，supplier の数を減らすために，supplier の条件に入次数を付け加えたものであり，条件を厳しくしていっている．central の条件は，supplier かつ client であるというものを基本にしている．library はの条件は，C7~C13 で使用した条件を各種試している．

supplier の条件ごとにブロック単位（C14~C20，C21~C27 というように，表 1 で 2 重線で区切ってある）で見ると，グラフの形はほぼ規則的な動きをしており，supplier の条件が厳しくなるにつれて，クラスタリング対象モジュール数が増加していき，平均 *CF* 値は若干減少していることがわかる．15 システムの結果を見ると，C14 以降の各ブロックの後半 4 条件（最後のブロックのみ後半 3 条件）では，必ずクラスタリング対象モジュール数，平均 *CF* 値ともに C1 に比べて改善されている．これらの調和平均値はほぼ同じだが，クラスタリング対象モジュール数と平均 *CF* 値にはそれぞれブロックごとに違いが見られる．C19，C20，C26，C27，C33，C34，C40，C41，C47 は Bunch のバグを含む例なので省き，library の条件は入次数平均以上，出次数 0，CRV 平均以上を採用すると，C18，C25，C32，C39 が残ることになる．

C18，C25，C32，C39 の 4 条件（supplier の入次数条件が順に平均の 1 倍，1.5 倍，2 倍，2.5 倍）を比べると，順に，クラスタリング対象モジュール数は増加していき，平均 *CF* 値は

減少していくことがわかる．ここで，クラスタリング対象モジュールと，平均 *CF* 値のどちらを優先するかを考えると，クラスタリング結果を与えられた時に，遍在モジュールがあまりに多いのでは，クラスタリングの意味が薄れてしまうため，少ない方が望ましい．また，きちんとモジュール化して作られたシステムであっても，クラスタ間に依存辺がほとんど無いという状況は考えにくいため，平均 *CF* 値は改善しにくいと考えられる．よって，クラスタリング対象モジュール数を重視すると考えると，C32，C39 が望ましい．C32 は，クラスタリング対象モジュール数の増加率 20%，平均 *CF* 値増加率 6.2% であるのに対し，C39 は，それぞれ 26%，1.8%と平均 *CF* 値の増加率が低いため，C32 の方が優れているといえる．

ここまで，15 システム平均で評価を行なってきて，C32 が最もよいという結論に至ったが，C32 において平均 *CF* 値が改悪しているシステムも存在する．その例として，uitags の結果を図 13 に示す．

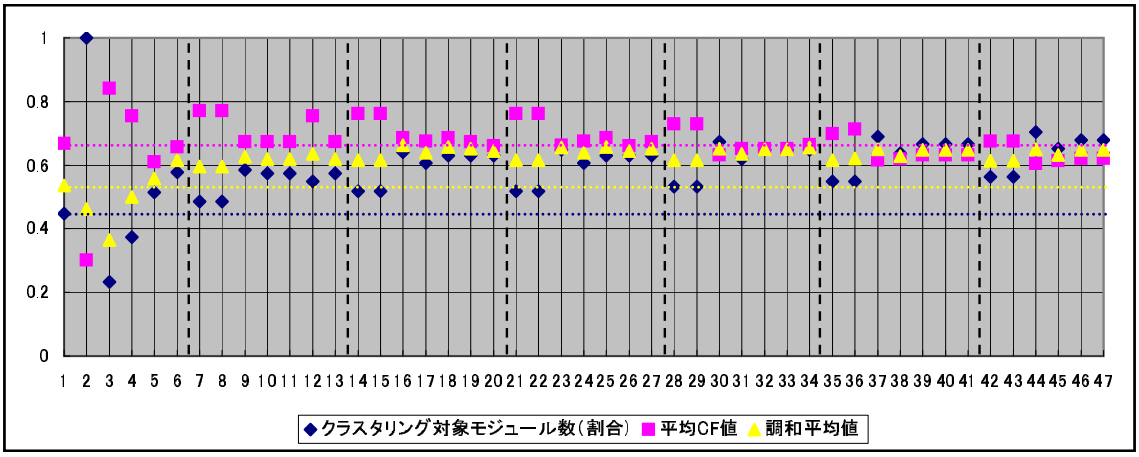


図 13: uitags における結果

uitags の場合，C32 での平均 *CF* 値が C1 に比べてわずかに減少（0.669→0.652）している．しかし，その代わりに，クラスタリング対象モジュール数の割合が C1 に比べて 45%増加（0.447→0.648）している．C1 では，過半数が遍在モジュールになるという非常に悪い結果であったため，平均 *CF* 値がわずかに減少していたとしても，クラスタリング結果の質が向上したといえる．なお，15 システムのうち，4 システムに関しては，平均 *CF* 値が最大 4.1%減少している．ここに提示しなかった残り 13 システムに対する個別の結果は付録という形で付す．

4.3 実験 2：ベンチマークの利用

4.3.1 方針

実験 1 では，クラスタリング結果の詳細を詳しく見ておらず，表面的な評価である．この実験では，クラスタリング結果の詳細に注目して評価を行なう．

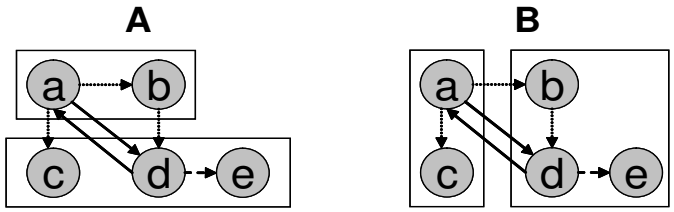
クラスタリング・アルゴリズムの評価方法は非常に難しい問題で，近年特に研究が行なわれている分野であるが，クラスタリング・アルゴリズムを評価するためによく用いられる手法に，ベンチマークを作成して解と比較するという手法がある [20,34]．つまり，あるシステムに対して，正しいと思われるサブシステム分解を作成して，それと，クラスタリング・アルゴリズムの解とを比較するという手法である．ここで，注意すべき点として，ベンチマークとはあくまでも人間が見て正しいと思われる解であり，基準となる一つの点でしかない．様々なクラスタリング・アルゴリズムな様々な目的・方針を持っているわけで，それをベンチマークで計るということは絶対的な正しい評価とはいえず，あくまでも一つの評価としかいえないという問題点がある．また，ベンチマークの作成の際に，主観が入るのは避けられない，大規模なシステムのベンチマークを作成するのは困難である，という問題点もある．

ベンチマークと解を比較するためには，その両者の距離を計る手法がいくつか提案されているが，今回は，Mitchell らによって提案された EdgeSim と MeCl という 2 つのメトリクス [20] を用いる．

EdgeSim は，ベンチマークと解の辺（Edge）の類次度（Similarity）を計る手法である．図 14 に例を挙げて説明する．ある依存グラフに対して，A と B の 2 つの分解がある．この図には辺は 6 本存在するが，それを 3 種類に分類する．1 種類目は，破線で示した $\langle d, e\rangle$ という辺で，A でも B でも，1 つのクラスタ内部に閉じている辺である．2 種類目は，実線で示した $\langle a, d\rangle$，$\langle d, a\rangle$ という 2 本の辺で，これらは，A でも B でも，異なる 2 つのクラスタ間にまたがっている．3 種類目は，点線で示した残り 3 本の辺で，一方では 1 つのクラスタ内に閉じているが，一方では異なる 2 つのクラスタにまたがっている辺である．この 3 種類のうち，1 種類目と 2 種類目の辺を，類似している辺と定義して，その辺の割合を EdgeSim と定義する．図 14 の場合，6 本中 3 本が類似しているため，EdgeSim は 50%となる．

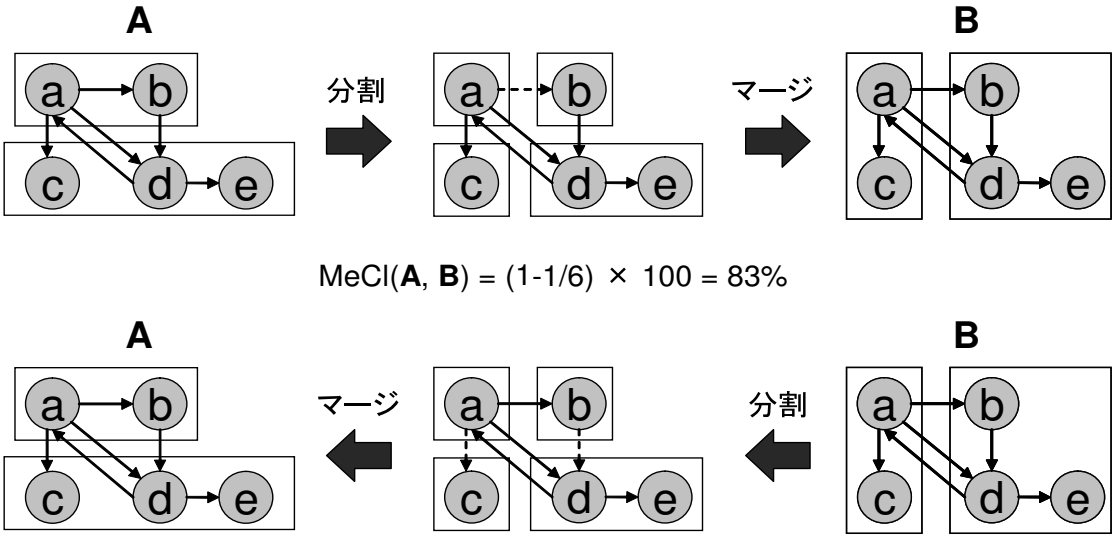
一方，MeCl は，ある分割中のクラスタ（Cluster）を，分割した後マージ（Merge）して，もう一方の分割に一致させるために必要なコストに注目した距離測定法である．図 15 は，図 14 と同じ 2 つの分割 A と B の距離 MeCl を測定している例である．図の上半分は，A を分割，マージして B と一致させようとしている．この時，MeCl では，分割する時に 2 つのクラスタに分かれてしまった辺に注目する[4]．この場合では，破線で示した $\langle a, b\rangle$ という辺である．そして，残りの辺の割合を MeCl(A, B) と定義する．この場合，残っている辺は 5/6

[4]分割・マージする回数を測定する MoJoFM [34] というメトリクスも存在する．



EdgeSim = 3/6 × 100 = 50%

図 14: EdgeSim 測定の例



MeCl(**A**, **B**) = (1-1/6) × 100 = 83%

MeCl(**B**, **A**) = (1-2/6) × 100 = 67%

MeCl = min(MeCl(**A**, **B**), MeCl(**B**, **A**)) = 67%

図 15: MeCl 測定の例

で 83%となる．また，今度は逆に B を A に一致させる MeCl(B, A) を測定する（図 15 の下半分）．そして，MeCl(A, B) と MeCl(B, A) の小さい方を MeCl として定義する．この例の場合，MeCl(B, A) の方が小さく，67%となっている．

なお，EdgeSim，MeCl は遍在モジュールを考慮するように実装[5]されていないため，現状では，suppliers，clients，centrals，libraries という 4 つのクラスタがあるとみなされてしまう．そのため評価値がおかしくなる場合があり，この点は改善すべき点である．

[5]Bunch の補助機能として測定機能が実装されている．

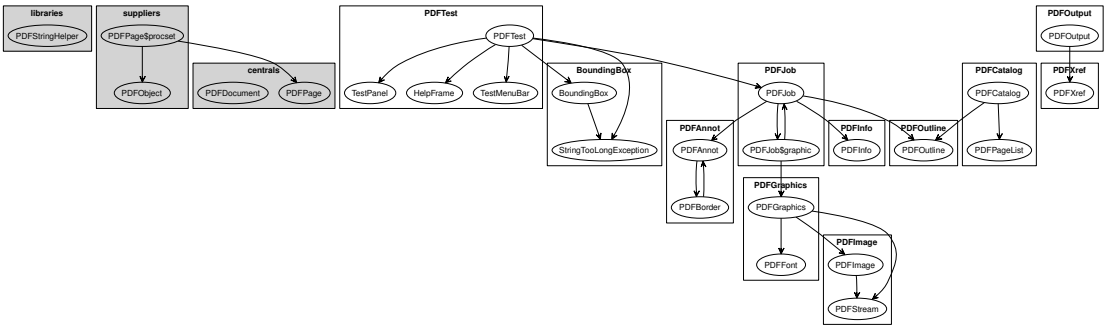


図 16: GNUJpdf のベンチマーク 1

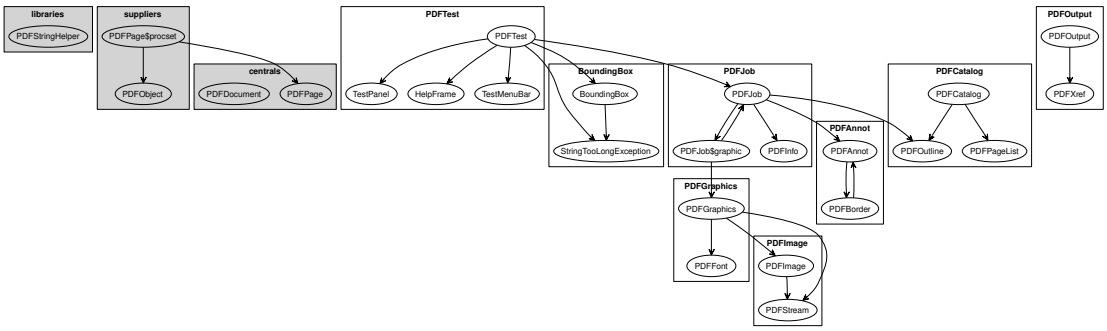


図 17: GNUJpdf のベンチマーク 2

本実験では，25 クラスから成る GNUJpdf のベンチマークを作成し，表 1 で示した 47 の遍在モジュール特定手法における解との比較を EdgeSim と MeCl を測定することで行なった．

ベンチマークの作成方法は，ソースコード等を読んで構造・挙動を理解して分割した．今回の実験は Bunch のクラスタリング結果の最下層（最も詳細な分割）を使用しているため，できるだけ詳細に分割した．GNUJpdf は，PDF を生成・印刷するための package であるため，PDF Reference [11] を参考に，PDF 文書構造ごとに分割することができた．作成したベンチマークを図 16 に示す．また，Bunch のアルゴリズムでは要素数 1 であるクラスタは $CF = 0$ となるため，つくり出すことはない．図 16 は，要素数 1 のクラスタを含んでいるが，Bunch に合わせて要素数 1 のクラスタを適当なクラスタにマージしたベンチマーク（図 17）も作成した．以降，前者をベンチマーク 1，後者をベンチマーク 2 と呼ぶ．

また，EdgeSim，MeCl は遍在モジュールを考慮できないため，結果が正しくないことがあるため，実際のクラスタリング結果とベンチマークを目で見比べて考察も行なう．

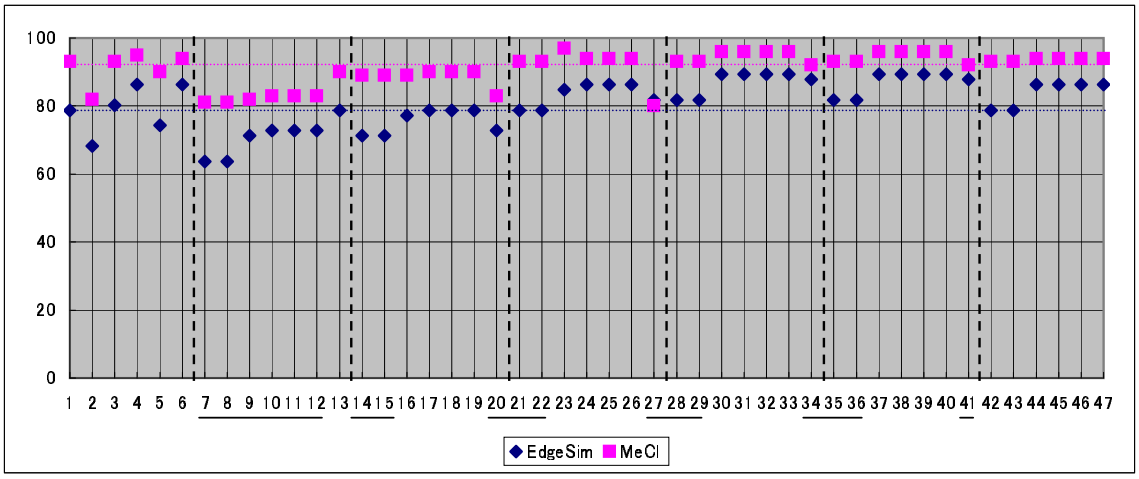


図 18: ベンチマーク 1 での結果

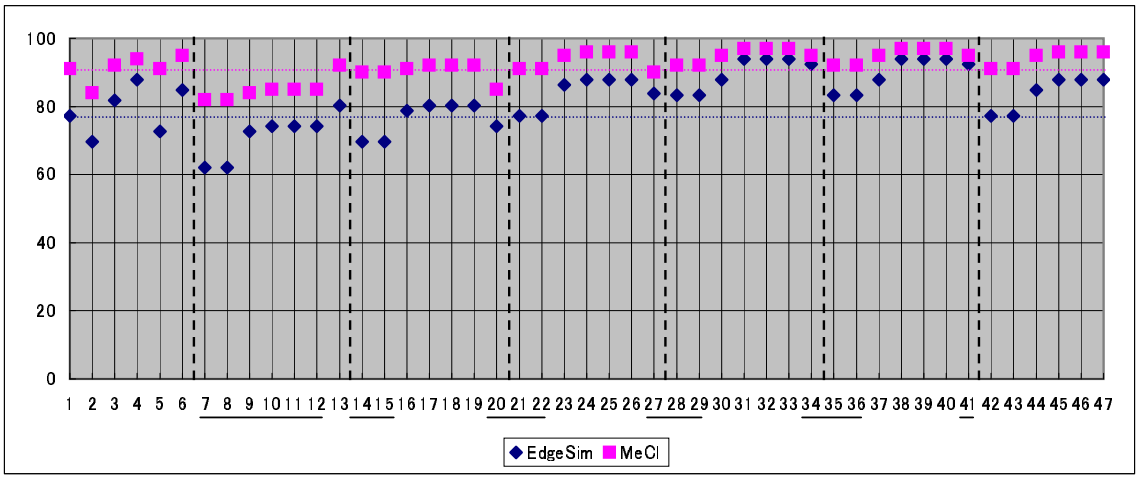


図 19: ベンチマーク 2 での結果

4.3.2 結果と考察

ベンチマーク 1，ベンチマーク 2 に対して 47 個の遍在モジュール特定条件を適用した解との距離を取った結果をそれぞれ図 18，図 19 に示す．なお，条件番号に下線が引かれているものは，Bunch の出力解にモジュールが欠損するバグを含んでいるもので，そのままでは距離が測定できないため，出力に手を加えて距離を測定している．

2 つの結果は比較的似ているが，ベンチマーク 2 での結果の方が若干数値が高い．EdgeSim と MeCl の相関係数は，ベンチマーク 1 で 0.844，ベンチマーク 2 で 0.908 となっており，2 つのメトリクスに高い相関があることがわかる．実験 1 の考察で優れていると判断された

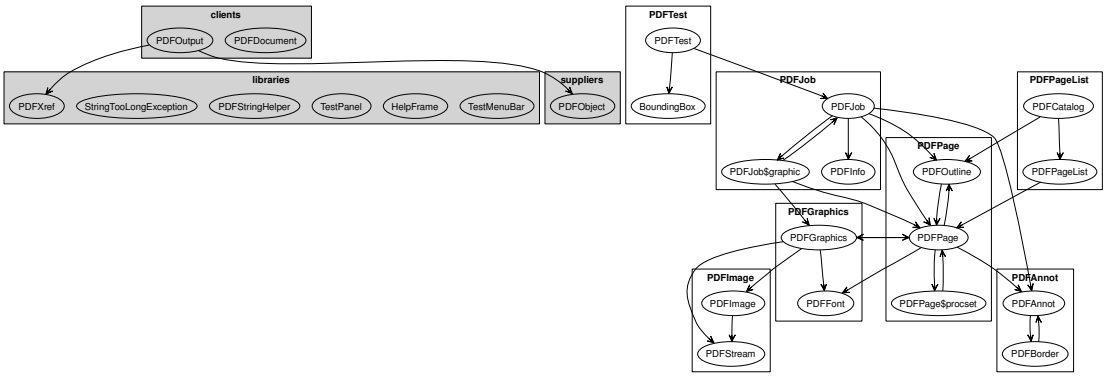


図 20: C1 でのクラスタリング結果

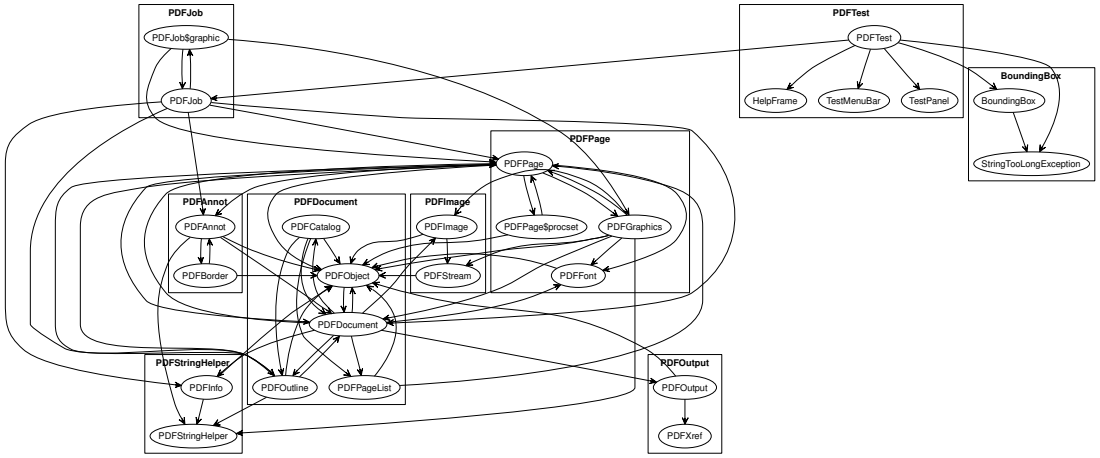


図 21: C2 でのクラスタリング結果

C18，C25，C32，C39 等は実験 2 でも優れた結果を示しており，特に C32，C39 の改善率は高い．

以下，特筆すべき点に関して，実際のクラスタリング結果とベンチマークへの距離を比べながら考察する．

まず，Bunch のオリジナル条件である C1 でのクラスタリング結果を図 20 に示す．ベンチマーク 2 と比べると，library が多過ぎる点と，クラスタ間に辺が多い点が目につく．特にベンチマークでは central だとみなした `PDFPage` がクラスタリング対象になっており，その周りの辺が多く邪魔になっている．遍在モジュールを考慮できないため，EdgeSim，MeCl は以外と高くなってしまっている．

図 21 は，C2，つまり遍在モジュールを全く特定しない場合の結果である．あまりにクラスタ間の辺が多くて，とても見づらい結果になっている．

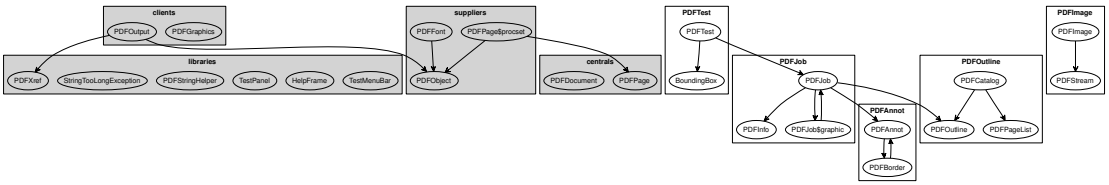


図 22: C4 でのクラスタリング結果

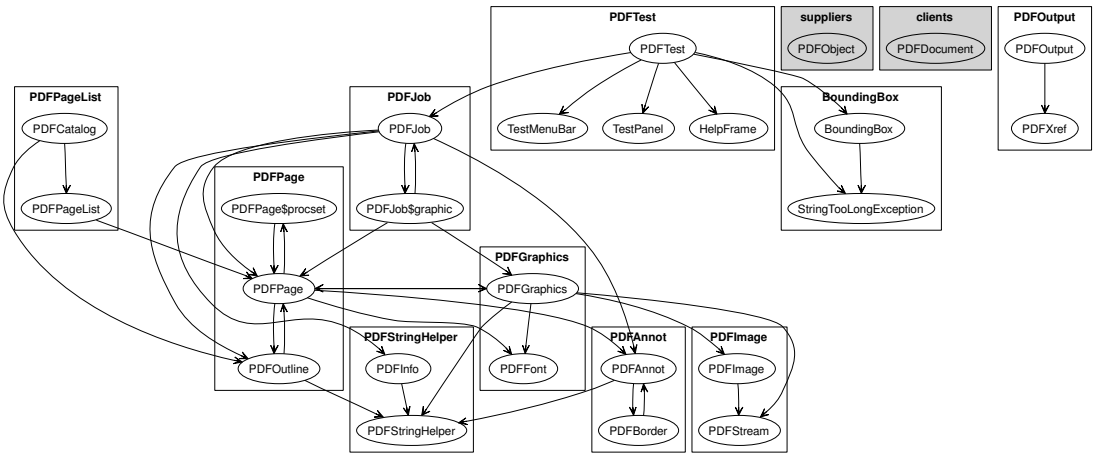


図 23: C6 でのクラスタリング結果

C4 での距離は非常に高い数値を示しているが，実際のクラスタリング結果は図 22 で，クラスタリング対象モジュールよりも遍在モジュールの方が多いという結果になっている．これは，EdgeSim，MeCl が遍在モジュールを考慮しないという問題点のためである．

C6 は，オリジナル条件から library を除いたものであるが，クラスタリング結果は図 23 である．ベンチマークに比べて遍在モジュールの数が少ないが，意外とクラスタ構造はベンチマークと似ている．ただし，若干クラスタ間の辺の数が多い．

図 24 は，supplier の条件を CRV 平均以上としたブロック（C7~C13）の中で唯一バグの無かった C13 のクラスタリング結果である．ベンチマークとの距離も比較的良い結果を得ており，実際の結果を見ても，遍在モジュールが多過ぎず，クラスタ間の辺も少なくてとても見やすい．クラスタ構造もベンチマークと比較的似ている．この例では，client は無いと判断されたため，遍在モジュールとして特定されたものはすべて CRV が平均以上のものとなっているが，CRV 平均以上のモジュールを取り除くだけで，クラスタリング結果がずいぶん見やすくなることがわかる．

図 24，25，26 は，それぞれ実験 1 で良いと判断された C18（C13 と同じ結果），C25，C32（C39 も同じ）のクラスタリング結果である．どの結果も非常にベンチマークに近い．この 3 つの違いとしては，supplier の条件が厳しくなるにつれて supplier が 1 つずつ減っていって

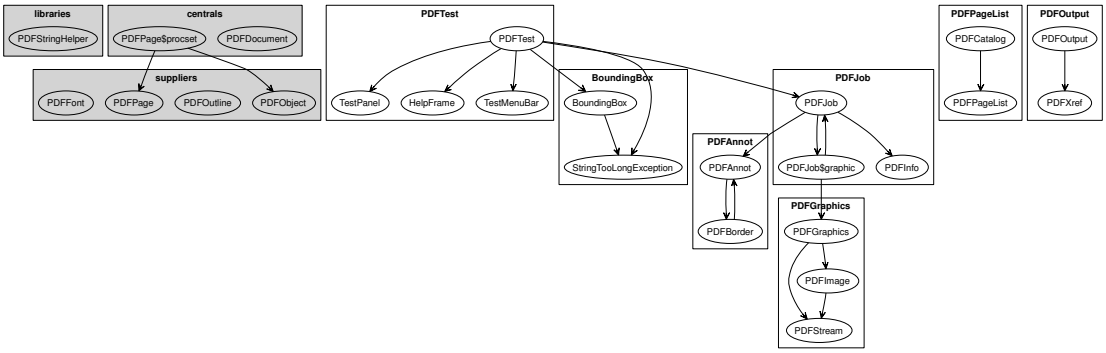


図 24: C13，C18 でのクラスタリング結果

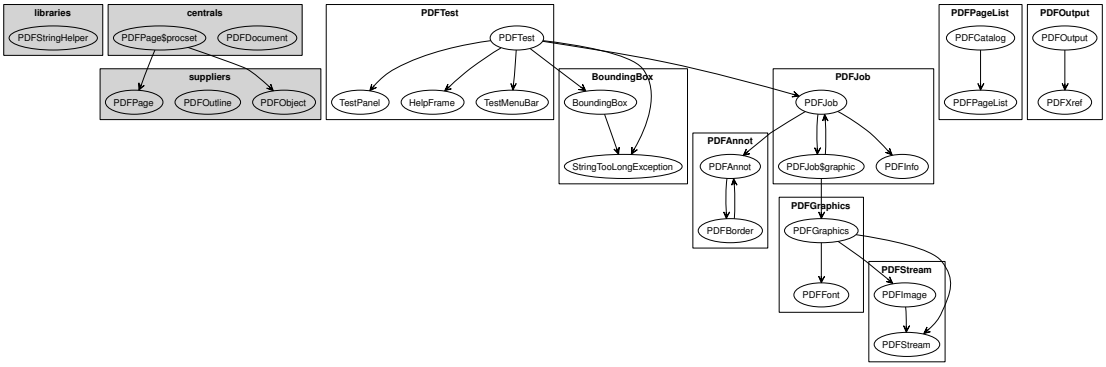


図 25: C25 でのクラスタリング結果

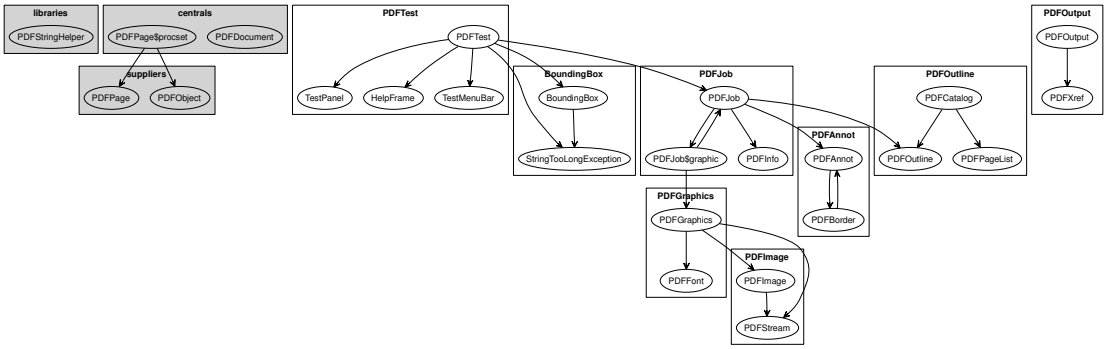


図 26: C32，C39 でのクラスタリング結果

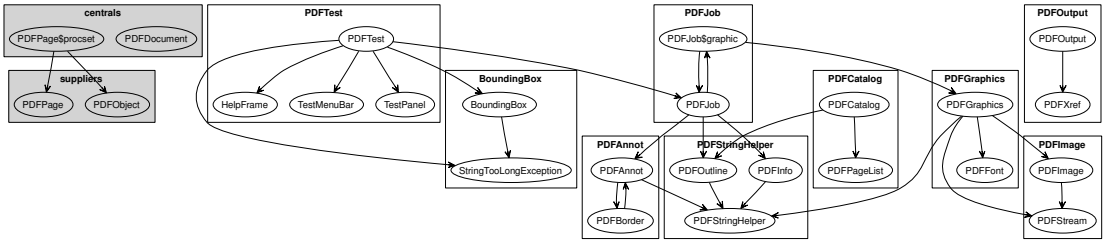


図 27: C30 でのクラスタリング結果

いるだけである．C32，C39 の結果はほぼベンチマーク 2 と同じと言ってよく，PDFPage が supplier か central かという違いしかない．非常に優れた条件であるといえる．

最後に，C30，つまり，C32 から library 条件を除いた条件の結果を図 27 に示す．図 26 で library となっている PDFStringHelper をクラスタリング対象モジュールに変わっているが，そのモジュールに対しての辺が増えていて若干見にくい．Helper という名前が示すように，このクラスはライブラリとして用いられているため，やはりクラスタリング対象から外した方が良いといえ，C30 よりも C32 の方が良いといえる．

4.4 実験 1 と実験 2 のまとめ

実験 1 と実験 2 を通して，遍在モジュール特定条件に CRV を使うことの有用性を評価してきた．まず，supplier に関しては，CRV が高く，さらに入次数もある程度高いモジュールが適している．central は，supplier と client の両方の条件を満たしている，と定めるのが無難である．library に関しては，出次数が 0 のモジュールのうち，あまり使われていないモジュールはクラスタに入れて，比較的よく使われている部品はライブラリとして設計された確率が高そうである．

特定条件としては C25，C32，C39 が優れており，特に C32 がバランスが良い．よって，次の条件を，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における適切な条件として提案する．

- omnipresent supplier：入次数が平均次数の 2 倍以上，かつ，コンポーネントランク値が平均以上
- omnipresent client：出次数が平均次数の 3 倍以上
- omnipresent central：入次数が平均次数の 2 倍以上，かつ，出次数が平均次数の 3 倍以上，かつ，コンポーネントランク値が平均以上

- library：入次数が平均以上，かつ，出次数が 0，かつ，コンポーネントランク値が平均以上

5　Java クラス分類手法

本節では，4 節で提案した新たな遍在モジュール特定手法を用いた Java クラス分類手法を提案し，試作を行なう．

5.1　提案するシステム

本研究で提案する Java クラス分類手法のシステム構造を図 28 に示す．

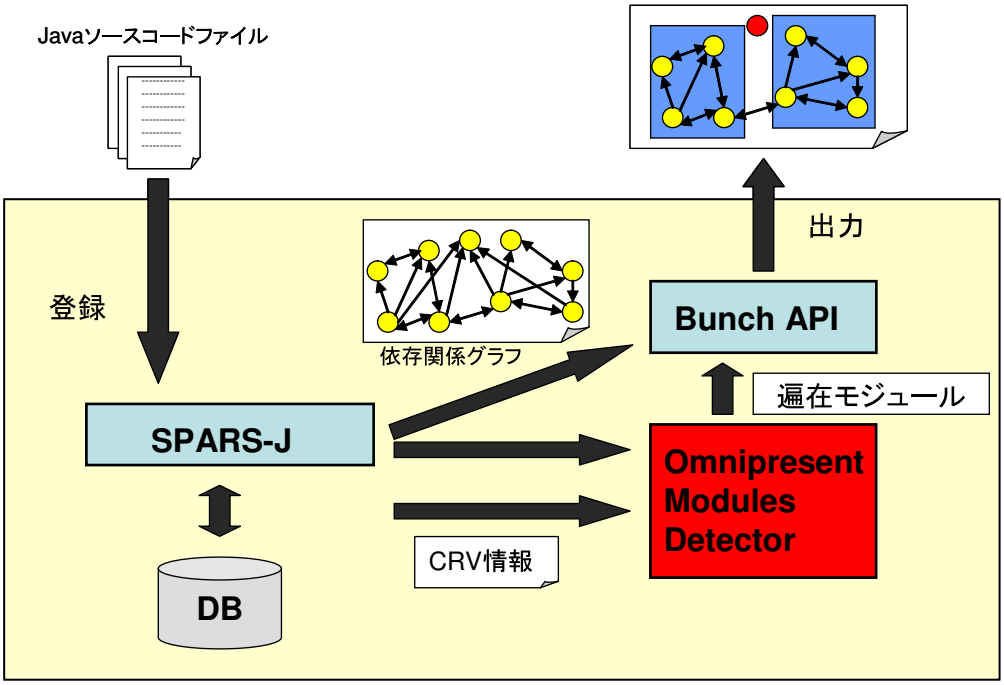


図 28: Java クラス分類手法のシステム構造

このシステムでは，ソフトウェア・クラスタリング・アルゴリズムには，前述の Bunch API を利用する．Bunch API の入力となる依存関係グラフの取得には SPARS-J を利用する．SPARS-J は，ソースコードから依存関係を解析し，その依存関係を元に CRV を計算してデータベースに格納するため，Bunch API の入力となる依存関係グラフと，Omnimodule Modules Detector の入力となる CRV 情報を出力することができる．

前述の通り，SPARS-J は，Java を対象としたシステムのため，今回提案するシステムは，Java を対象としたクラスタリング・システムとなっている．SPARS-J の解析部では，Java のクラス・内部クラス（inner class）・インターフェースをモジュールとして扱い，インスタンス作成，メソッド呼び出し，変数参照，継承，インターフェース実装を依存関係として扱う

ため，その方針に従った依存関係グラフが Bunch API の入力となる．なお，SPARS の拡張をすることによって他言語への対応も容易に可能となる．

Omnipresent Modules Detector では，SPARS-J から，依存関係グラフと CRV 情報を受け取る．依存関係グラフは，次数を計算するために用いる．そして，4 節で提案した，次の条件を用いて，遍在モジュールを特定する．

- omnipresent supplier：入次数が平均次数の 2 倍以上，かつ，コンポーネントランク値が平均以上
- omnipresent client：出次数が平均次数の 3 倍以上
- omnipresent central：入次数が平均次数の 2 倍以上，かつ，出次数が平均次数の 3 倍以上，かつ，コンポーネントランク値が平均以上
- library：入次数が平均以上，かつ，出次数が 0，かつ，コンポーネントランク値が平均以上

そして，特定した遍在モジュールと，依存関係グラフを Bunch API へ入力することで，ソフトウェア・クラスタリングを行ない，解を出力する．

以上が，本研究で提案する Java クラス分類手法であるが，さらなる拡張として，得られたソフトウェア・クラスタリング結果をデータベースに格納して，SPARS-J と組み合わせることにより，ソフトウェア部品再利用のための検索と理解を助けるような統合システムも構築可能である．

5.2 システムの試作

本節では，5.1 節で示したクラスタリング・システムの試作について述べる．

Bunch API は `jar` ファイルで提供されているため，システムは Java によって実装を行なった．一方，SPARS-J のシステムは C，C++で実装されているため，Java のシステムに組み込むのではなく，シェルを通して処理を行なっている．

SPARS-J では，`make` もしくは `gmake` を用いて，ソースコードの場所とデータベースの場所を指定して，システムの登録を行なう．その後，SPARS-J に含まれる `relation`，`packageinfo` というツール（プログラム）を用いて，依存関係グラフと，CRV 情報を取得する．

そして，`OmnipresentModulesDetector` というクラスが，依存関係グラフと CRV 情報を受け取り，次数を計算して，遍在モジュールを特定する．特定条件は 5.1 節等で示した通りで，library，central，supplier，client という優先順位をつけて特定を行なう．

Bunch APIでは，クラスタリング･アルゴリズムへのインターフェースとなるクラスBunchAPIと，クラスタリングの際の各種設定を行なうためのクラスBunchPropertiesが用意されており，その2つのクラスを用いてソフトウェア・クラスタリングを行なうように設計されている．

クラスBunchPropertiesに対しては，依存関係グラフのファイル名，出力形式，山登り法における初期状態数，焼きなまし法を適用するか，適用するなら初期温度と冷却率，各種遍在モジュール等の設定を行なう．現状では，試作段階ということで，ソースファイル上で指定を行なっているが，今後，GUIを介した設定，もしくは，Javaの`Properties`クラスを利用したプロパティファイルによる設定方法の実装を予定している．

そして，BunchAPIにBunchPropertiesを設定して，クラスタリングを実行することで，解が出力される．

図29に，SAXに対して，このシステムを用いてソフトウェア・クラスタリングを行なった結果を例として示す．出力する層（Bunchでは階層的にクラスタリングを行なうので，出力する層を指定できる）はメディアン（この場合はLevel 1）を指定している．なお，`dot`の描画エンジンは横長に描画するため，一部オブジェクトを移動している．また，本来は，パッケージ名も表示されているが，この図ではパッケージ名の共通部分（`org.xml.sax.`）を削除している．

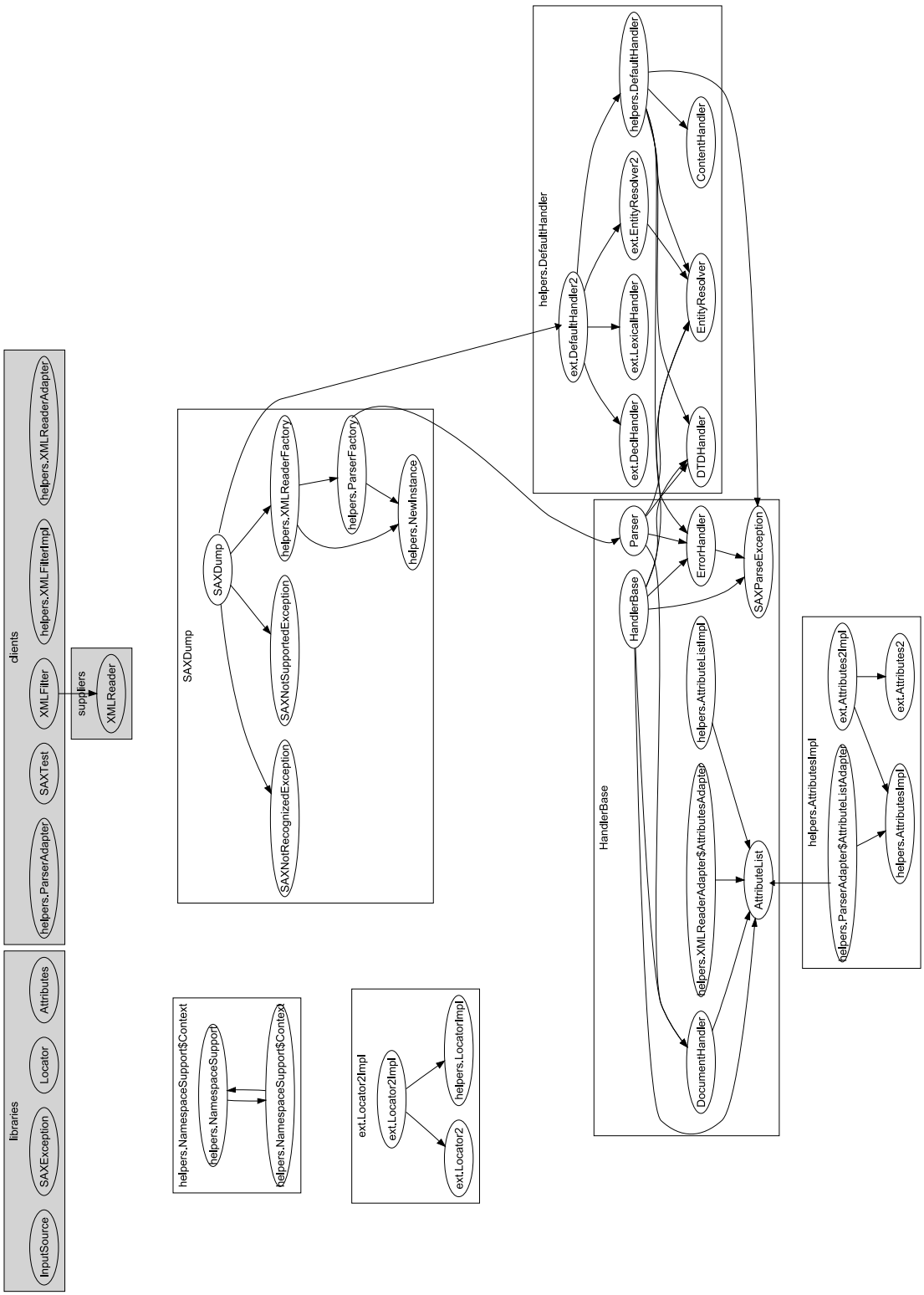


図 29: SAX に対して適用した例

6　まとめと今後の課題

本研究では，コンポーネントランク法によって測定される，Java クラスの重要度を表すメトリクス，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することによって，従来の手法より優れた，高凝集・低結合の原則に基づく Java クラス分類手法を実現できることを実験により確認した．そして，適切な遍在モジュール特定手法を決定し，その手法を組み合わせた Java クラス分類手法を提案した．

今後の課題としては，まず，クラスタリング評価手法の拡張が挙げられる．現状では行なえない，遍在モジュールを考慮した上での評価を行なえるように改良する必要がある．その上で，より大規模なソフトウェア・システムでのベンチマーク実験を行ない，提案手法のさらなる評価を行なう必要がある．

また，Java クラスを検索するシステムである SPARS-J と，本研究で提案した分類手法を組み合わせることで，ソフトウェア部品の検索，ソフトウェア理解を一体化したシステムを構築することが考えらる．

謝辞

本研究の全過程を通して，常に適切な御指導およ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.

本論文を作成するにあたり，適切な御指導およ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 松下 誠 助教授に心から感謝いたします.

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] Bunch and BunchAPI. `http://serg.cs.drexel.edu/projects/bunch/`.

[2] ClusterNav. `http://serg.cs.drexel.edu/projects/bunch/clusternav/`.

[3] Graphviz - Graph Visualization Software. `http://www.graphviz.org/`.

[4] Bill Andreopoulos, Aijun An, Vassilios Tzerpos, and Xiaogang Wang. Multiple Layer Clustering of Large Software Systems. In *Proceedings of the Working Conference on Reverse Engineering*, pp. 79–88, Pittsburgh, November 2005.

[5] Periklis Andritsos and Vassilios Tzerpos. Software Clustering based on Information Loss Minimization. In *Proceedings of the Working Conference on Reverse Engineering 2003*, pp. 334–344, Victoria, November 2003.

[6] Nicolas Anquetil and Timothy Lethbridge. File Clustering Using Naming Conventions for Legacy Systems. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 184–195, Toronto, Ontario, Canada, November 1997. IBM Press.

[7] Ivan T. Bowman and Richard C. Holt. Software Architecture Recovery Using Conway's Law. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 123–133. IBM Press, Nov 1998.

[8] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 1997.

[9] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic Clustering of Software Systems using a Genetic Algorithm. In *IEEE Procedings of the International Conference on Software Tools and Engineering Practice 1999*, 1999.

[10] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An Empirical Study of the Robustness of Two Module Clustering Fitness Functions. In *Proceedings of the 2005 conference on Genetic and evolutionary computing*, pp. 1029–1036, Washington DC, 2005. ACM Press.

[11] Adobe Systems Inc. PDF Reference sixth edition. `http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference.pdf`, 2006.

[12] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component Rank: Relative Significance Rank for Software Component Search. In *Proceedings of the 25th International Conference on Software Engineering*, pp. 14–24, May 2002.

[13] Katsuro Inoue, Reishi Yokomori, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Ranking Significance of Software Components Based on Use Relations. *IEEE Transactions on Sofware Engineering*, Vol. 31, No. 3, pp. 213–225, MARCH 2005.

[14] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, Vol. 220, No. 4598, pp. 671–680, May 1983.

[15] Charles W. Krueger. Software Reuse. *ACM Computing Surveys (CSUR)*, Vol. 24, No. 2, pp. 131–183, June 1992.

[16] Jing Luo, Lu Zhang, and Jiasu Sun. A Hierarchical Decomposition Method for Object-Oriented Systems Based on Identifying Omnipresent Clusters. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 647–650, September 2005.

[17] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science*, Vol. 52, No. 7, pp. 1015–1030, July 2006.

[18] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *IEEE Proceedings of the International Workshop on Program Comprehension 1998*. IEEE Press, 1998.

[19] Brian S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD thesis, Drexel University, Philadelphia, PA, 2002.

[20] Brian S. Mitchell and Spiros Mancoridis. Comparing the Decompositions Produced by Software Clustering algorithms using similarity measurements. In *Proceedings of International Conference on Software Maintenance*, pp. 744–753, 2001.

[21] Brian S. Mitchell and Spiros Mancoridis. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Transaction on Software Engineering*, Vol. 32, No. 3, pp. 193–208, MARCH 2006.

[22] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[23] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A Reverse Engineering Approach To Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, Vol. 5, No. 4, pp. 181–204, 1993.

[24] Glenford J. Myers. *Software Reliability: Principles and Practices*. John Wiley & Sons, Inc., New York, 1976.

[25] Robert W. Schwanke. An Intelligent Tool For Re-engineering Software Modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pp. 83–92, May 1991.

[26] Robert W. Schwanke, R. Z. Altucher, and M. A. Platoff. Discovering, Visualizing, and Controlling Software Structure. In *Proceedings of the 5th International Workshop on Software Specification and Design*, pp. 147–154, 1989.

[27] Ali Shokoufandeh, Spiros Mancoridis, Trip Denton, and Matthew Maycock. Spectral and meta-heuristic algorithms for software clustering. *The Journal of Systems and Software*, Vol. 77, pp. 213–223, 2005.

[28] Noam Slonim and Naftali Tishby. Agglomerative Information Bottleneck. In *Proceedings of the Neural Information Processing Systems (NIPS-12)*, pp. 617–623, 1999.

[29] Daniel A. Spielman and Shang-Hua Teng. Spectral Partitioning Works: Planar Graphs and Finite Element Meshes. In *IEEE Symposium on Foundations of Coumputer Science*, pp. 96–105, 1996.

[30] Vassilios Tzerpos and R.C. Holt. The Orphan Adoption Problem in Architecture Maintenance. In *Proceedings of the Working Conference on Reverse Engineering*, pp. 76–82, Amsterdam, October 1997.

[31] Vassilios Tzserpos and R. C. Holt. ACDC : An Alogorithm for Comprehension-driven Clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pp. 258–267, 2000.

[32] Karl Ulrich. The role of product architecture in the manufacturing firm. *Research Policy*, Vol. 24, No. 3, pp. 419–440, May 1995.

[33] Anneliese von Mayrhauser, A. Marie Vans, and Steve Lang. Program comprehension and enhancement of software. In *IFIP World Computing Congress - Information Technology and Knowledge Engineering*, Vinna, Budapest, Sept. 1998.

[34] Zhihua Wen and Vassilios Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings of the International Workshop on Program Comprehension (IWPC '04)*, 2004.

[35] Zhihua Wen and Vassilios Tzerpos. Software Clustering based on Omnipresent Object Detection. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension*, pp. 269–278, St. Loius, May 2005.

[36] 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. 技術研究報告 229, 電子情報通信学会, 2005.

[37] 市井誠, 松下誠, 井上克郎. ソフトウェア部品グラフの次数グラフの次数分布におけるべき乗則の調査. ソフトウェア信頼性研究会 第 3 回ワークショップ論文集, pp. 87–96, July 2006.

[38] 嵩忠雄. 情報工学入門選書 6 情報と符号の理論入門. 昭晃堂, 1989.

[39] 横森励士, 梅森文彰, 西秀雄, 山本哲男, 松下誠, 楠本真二, 井上克郎. Java ソフトウェア部品検索システム SPARS-J. 電子情報通信学会論文誌 D-I, Vol. J87-D-I, No. 12, pp. 1060–1068, 12 2004.

付録

4.2 節では，実験 1 の結果として GNUJpdf，uitags の個別の結果と，15 システム平均値を示したが，ここに，残り 13 システムに対する実験 1 の結果を示す.

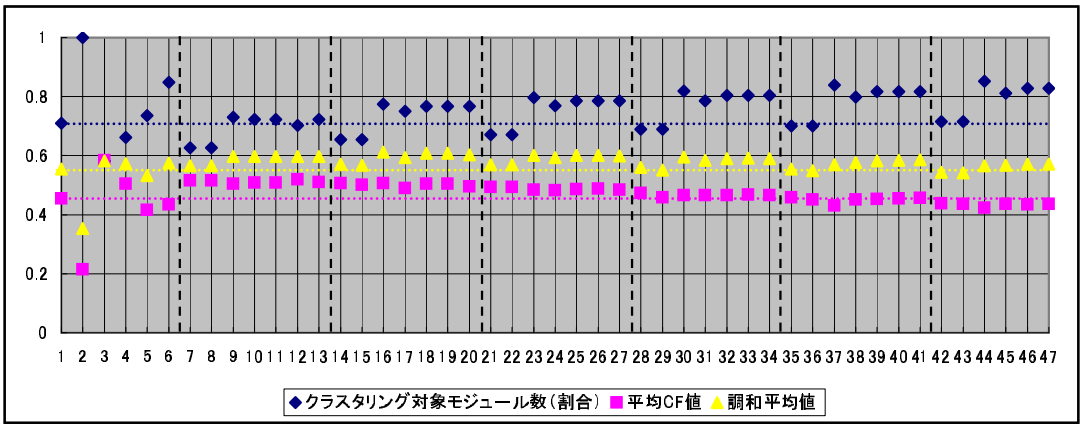


図 30: Azureus における結果

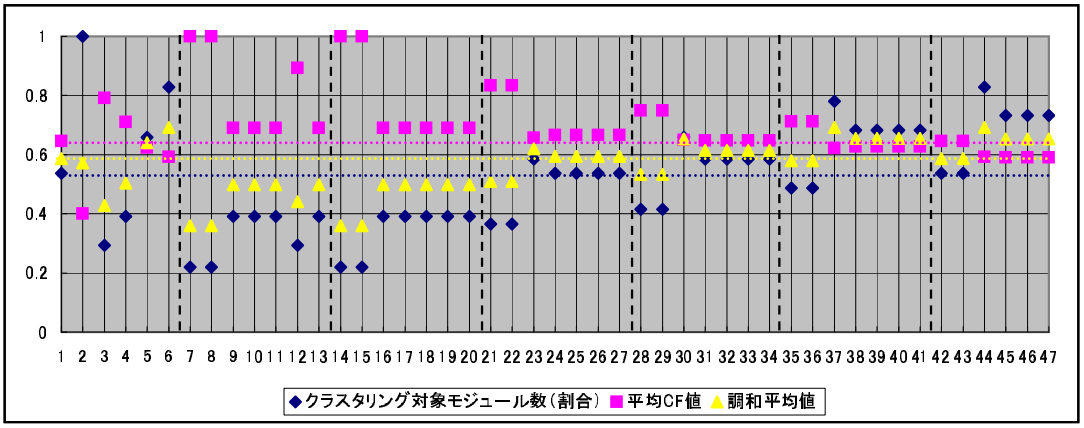


図 31: Dexter における結果

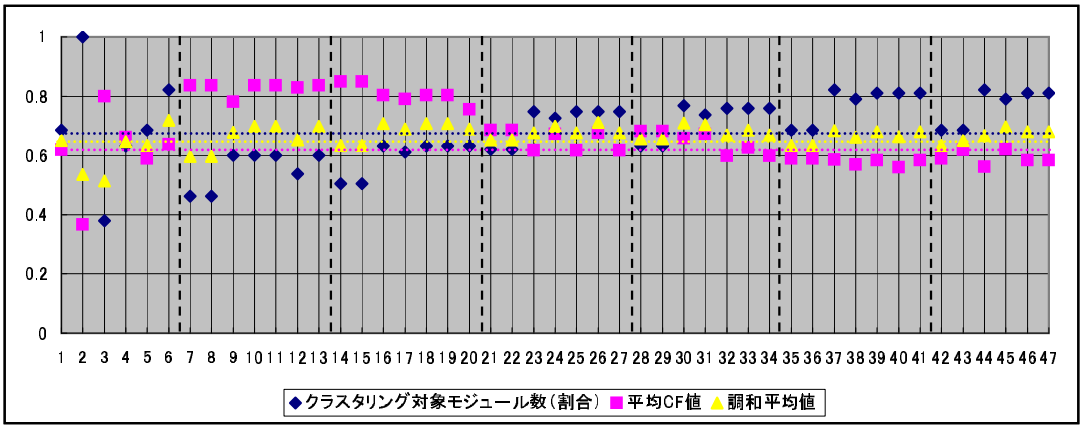


図 32: EJE における結果

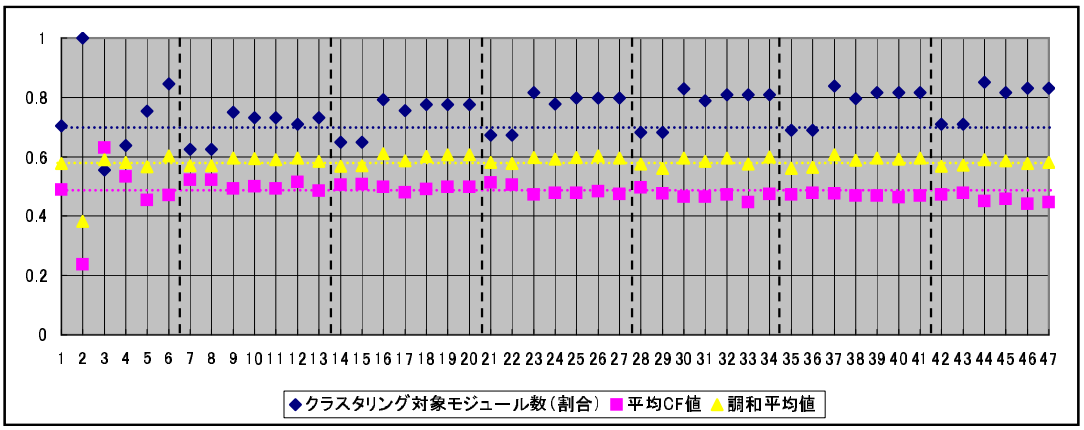


図 33: iText における結果

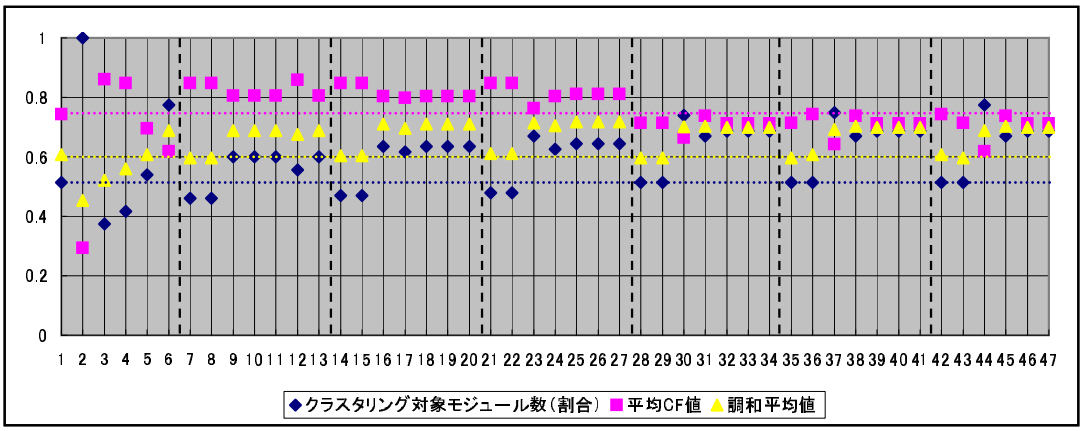


図 34: jpcap における結果

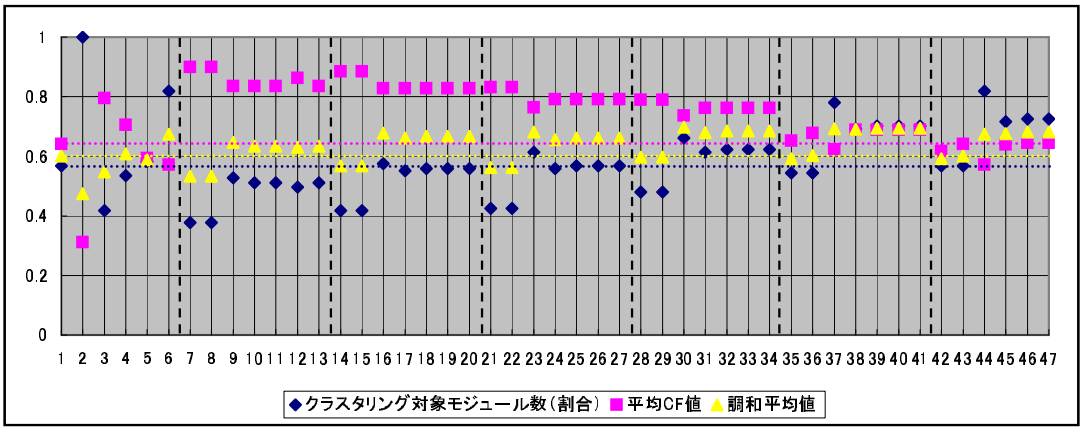


図 35: jVi における結果

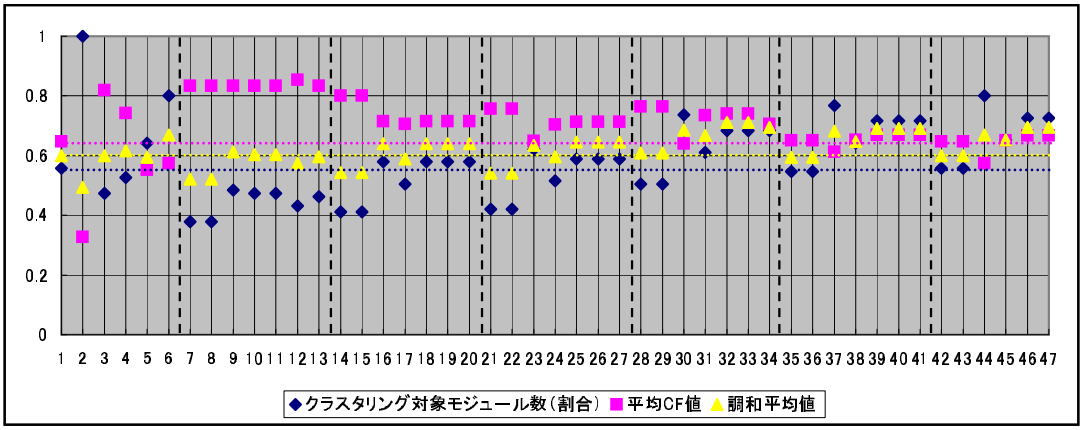


図 36: jwma における結果

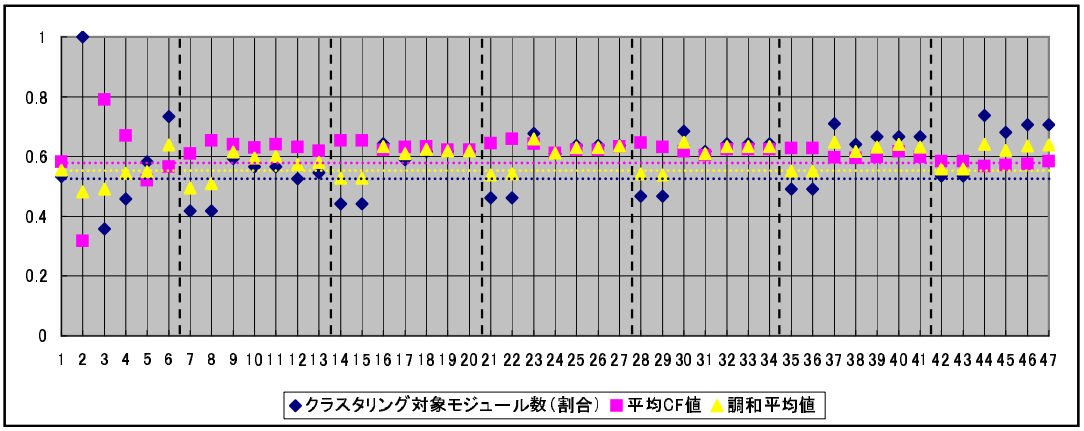


図 37: MicroEmulator における結果

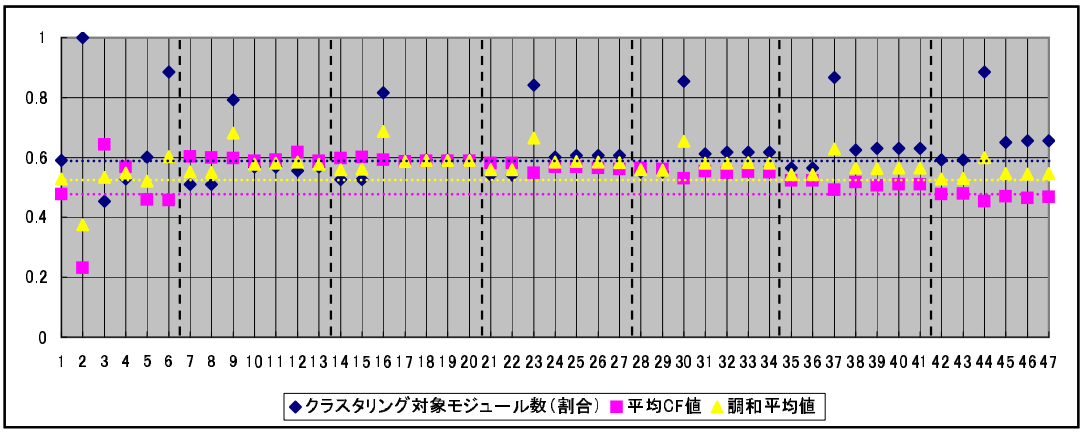


図 38: PMD における結果

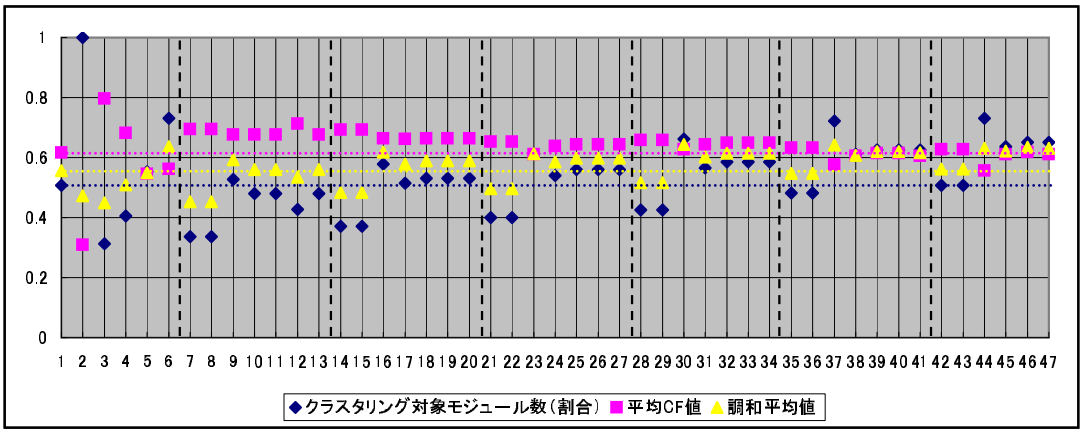


図 39: pollo における結果

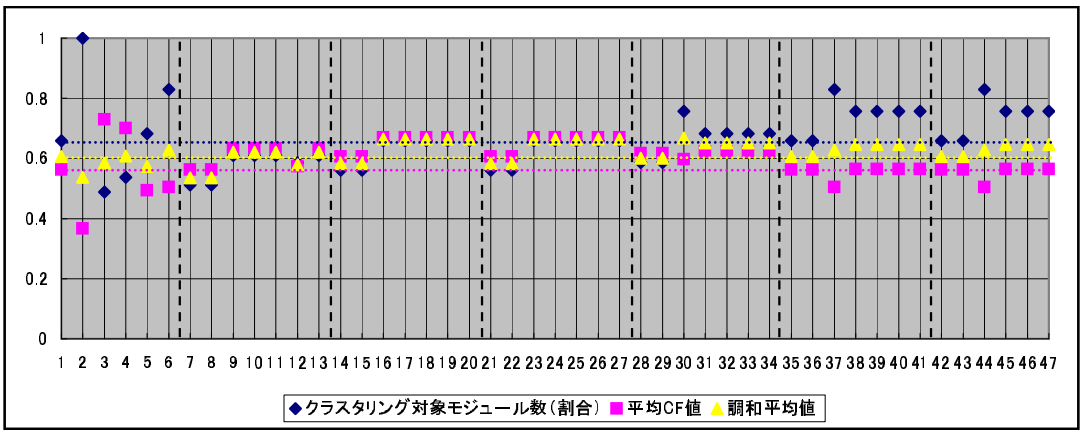


図 40: SAX における結果

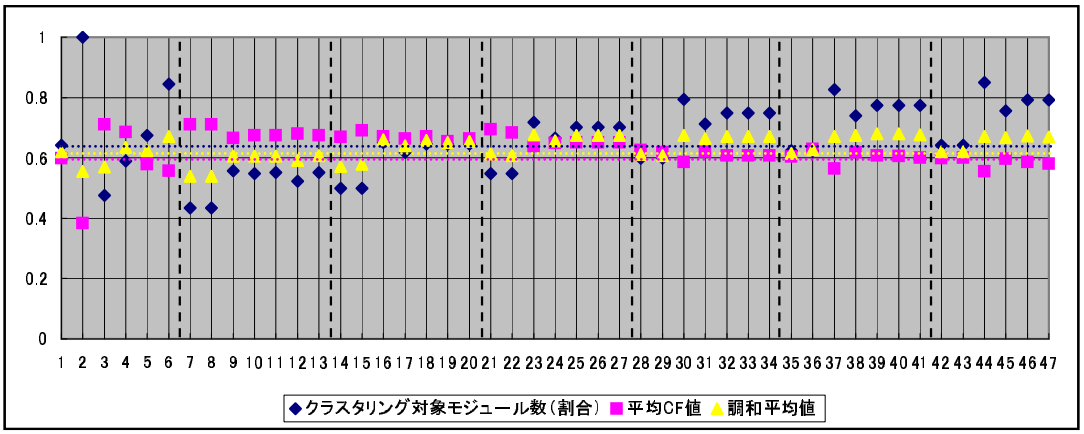


図 41: StringTree における結果

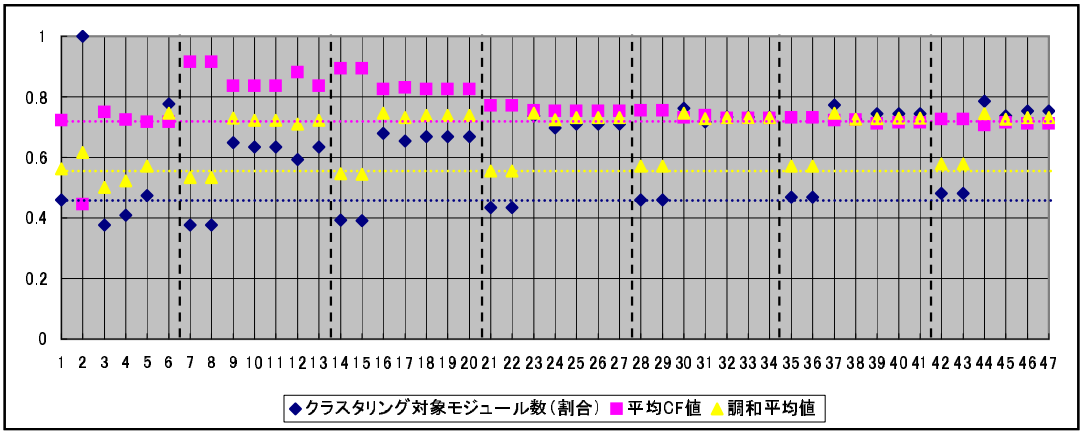


図 42: Xdoclet における結果