情報処理学会論文誌 Vol.56 No.2 669–681 (Feb. 2015)

大規模OSS開発における不具合修正時間の短縮化を目的としたバグトリアージ手法

柏 祐太郎[1,a)] 大平 雅雄[1,b)] 阿萬 裕久[2,c)] 亀井 靖高[3,d)]

受付日 2014年5月19日, 採録日 2014年11月10日

概要：本論文では，大規模 OSS 開発における不具合修正時間の短縮化を目的としたバグトリアージ手法を提案する．提案手法は，開発者の適性に加えて，開発者が一定期間に取り組めるタスク量の上限を考慮している点に特徴がある．Mozilla Firefox および Eclipse Platform プロジェクトを対象としたケーススタディを行った結果，提案手法について以下の 3 つの効果を確認した．(1) 一部の開発者へタスクが集中するという問題を緩和できること．(2) 現状のタスク割当て方法に比べ Firefox では 50%（Platform では誤差が大きすぎるため計測不能），既存手法に比べ Firefox では 34%，Platform では 38%の不具合修正時間を削減できること．(3) 提案手法で用いた 2 つの設定，プリファレンス（開発者の適性）と上限（開発者が取り組むことのできる時間の上限）が，タスクの分散効果にそれぞれ同程度寄与すること．

キーワード：バグトリアージ，大規模 OSS 開発，0-1 整数計画法

A Bug Triaging Method for Reducing the Time to Fix Bugs in Large-scale Open Source Software Development

Yutaro Kashiwa[1,a)] Masao Ohira[1,b)] Hirohisa Aman[2,c)] Yasutaka Kamei[3,d)]

Received: May 19, 2014, Accepted: November 10, 2014

Abstract: This paper proposes a bug triaging method to reduce the time to fix bugs in large-scale open source software development. Our method considers the upper limit of tasks which can be fixed by a developer in a certain period. In this paper, we conduct a case study of applying our method to Mozilla Firefox and Eclipse Platform projects and show the following findings: (1) using our method mitigates the situation where the majority of bug-fixing tasks are assigned to particular developers, (2) our method can reduce up to 50%–83% of time to fix bugs compared with the manual bug triaging method and up to 34%–38% compared with the existing method, and (3) the two factors, *Preference* (adequate for fixing a bug) and *Limit* (limits of developers' working hours), used in our method have an dispersion effect on the task assignment.

Keywords: bug triage, large-scale open source development, 0-1 integer programming

1. はじめに

近年の大規模システム開発では，試験工程だけでなく運用工程においても多数の不具合が検出される．多くの場合，不具合管理システムを用いて，不具合の再現方法や修正方法を詳細に記録し，不具合は漏れのないように管理される．不具合管理システムに報告された不具合１つ１つに対して，重要度や優先度を設定し，開発者に修正タスクを割り当てることをバグトリアージと呼ぶ [1].

しかしながら，大量に不具合が報告される現状では，個々の不具合に対して適切にバグトリアージを行うことは容易ではない．実際，大規模オープンソース開発プロジェクトの Eclipse や Mozilla では，約 4 割の不具合に

[1] 和歌山大学
Wakayama University, Wakayama 640–8510, Japan
[2] 愛媛大学
Ehime University, Matsuyama, Ehime 790–8577, Japan
[3] 九州大学
Kyushu University, Fukuoka 819–0395, Japan
[a)] s141015@sys.wakayama-u.ac.jp
[b)] masao@sys.wakayama-u.ac.jp
[c)] aman@cs.ehime-u.ac.jp
[d)] kamei@ait.kyushu-u.ac.jp

対して，担当者の再割当てが行われており [2]，人手によるバグトリアージには限界があることが知られている．担当者の再割当ては，人的リソースを浪費するだけではなく，不具合の修正作業を滞らせるため，できる限り生じないようにすることが望ましい．そのため現在，バグトリアージを支援するための研究がさかんに行われている [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11].

先行研究で提案されている手法のほとんどは，個々の不具合に対して確実かつ迅速に修正できる開発者を推薦することを目的としている．過去の不具合報告とその修正履歴に基づいて，新規に報告された不具合に対して適任の担当者を推薦することで，再割当てを起こしにくくすることが狙いである．しかし，既存手法は，個々の不具合修正の難易度や手間を考慮しないため，ごく一部の開発者にタスク割当てを集中させる傾向にある．優秀な開発者でも不具合修正に取り組める時間は有限であるため，既存手法は現実的でないと考えられる．

そこで本研究では，0-1 整数計画法に基づく不具合修正タスクの割当て手法を提案する．不具合の割当て問題を，開発者と不具合の組合せ問題としてとらえ，個々の開発者のタスク量に制約条件を課すことでタスク割当てを最適化し，プロジェクト全体としての不具合修正活動の効率化を目指す．本論文では，以下の Research Question に取り組み，提案手法の有用性について議論する．

RQ1：既存手法は，特定の開発者へ負荷を集中させる傾向があるか？　提案手法ではその問題を緩和できるか？　代表的な既存のバグトリアージ手法 [3] を用いて，既存手法が一部の開発者にタスクを集中して割り当てる可能性があることを確かめる．また，提案手法を用いることでタスク集中を緩和できることを確かめる．

RQ2：提案手法は，プロジェクト全体の不具合修正時間の短縮化に寄与するか？　0-1 整数計画法に基づくタスク割当ての最適化により，プロジェクト全体として不具合修正時間を短縮できるかどうかを実験的に検証する．すなわち，一部の開発者へのタスク集中を緩和することが，プロジェクト全体の不具合修正活動を効率化できることを示す．

RQ3：提案手法で用いる各種設定が，タスク割当ての最適化にどのように寄与するか？　提案手法の特徴は主に，(1) 不具合に対する開発者の適性（**プリファレンス**）を数値化し，適性の高い不具合をできるだけ多く割り当てることと，(2) 一部の開発者にタスクが集中するのを避けるために，一定期間内に修正可能な不具合の数に**上限**を設けることにある．プリファレンス有無，上限の有無により 4 つのモデルを構築し，修正時間短縮化の効果を詳細に調べる．

以降，2 章では，バグトリアージ支援に関する関連研究について紹介し，本研究との違いを明らかにする．3 章では，0-1 整数計画法に基づくバグトリアージ手法を提案する．4 章では，提案手法の有用性を確認するための実験について説明し，5 章で実験の結果を示す．6 章では，実験結果と追加実験の結果に基づいて提案手法の適用範囲および妥当性について考察する．最後に 7 章でまとめと今後の課題について述べ，本論文を結ぶ．

2. バグトリアージ

2.1 現状のバグトリアージにおける問題点

OSS 開発におけるバグトリアージは一般に，Bugzilla などの不具合管理システム（BTS: Bug Tracking System）を用いて行われる．BTS の管理者は，プロジェクトに所属する多数の開発者から当該不具合を修正するのに適任の開発者を決定し，不具合修正タスクを割り当てる必要がある．しかし，多くの場合，BTS 管理者が最初に割り当てた開発者では不具合を修正できずに，担当者を変更（再割当て）して不具合が修正されている．担当者の再割当ては不具合が最終的に修正されるまでの時間を大きく滞らせる [12]．そのため，近年の大規模 OSS 開発では，再割当ての頻発がプロジェクト全体としての不具合修正の長期化を引き起すという問題がある [2]．実際，Eclipse や Mozilla では，約 4 割の不具合に対して担当者の再割当てが行われており，1 度の担当者の変更で修正時間が平均約 50 日遅れるとされている [2]．再割当てによる不具合修正時間の長期化は，大規模 OSS プロジェクトにおいて解決すべき喫緊の課題であるとされており，バグトリアージを支援する手法がこれまで多数提案されてきた [1], [2], [3], [4], [5], [6], [7], [8], [9], [10].

2.2 既存のバグトリアージ手法とその問題点

既存のバグトリアージ手法の目的は，担当者の再割当てが頻発しないようにあらかじめ最も適任と思われる開発者にタスクを割り当てることである．既存手法は主に，機械学習に基づく方法を採用している [1], [3], [6]．具体的には，開発者が不具合報告に記述したタイトルと概要からなるテキストデータを入力として，各開発者が過去に用いた単語の出現頻度を算出し，機械学習のアルゴリズム（たとえば SVM [13]）を適用することで，各不具合に対して開発者を推薦するためのモデルを得る．構築したモデルに従うことで，比較的高精度（約 70–75%程度）に新規に報告された不具合の修正に対応可能な開発者を推薦できる [3].

しかしながら，1 章でも述べたように，既存手法は，一定期間内に開発者が不具合修正に取り組める時間を考慮しない．また，個々の不具合修正の難易度や手間を考慮しない．そのため，必ずしも有能な開発者が担当する必要のない軽微な不具合であっても，優先して有能な開発者に割り当てる可能性が高い．結果として，既存手法は一定期間内に取り組むことのできるタスク量を超えて一部の有能な開発者にタスク割当てを行う恐れがある．有能な開発者以外の開発者にも対応可能な軽微な不具合を，その他の開発者へ分散して割り当てることで，プロジェクト全体としての不具

合修正活動をさらに効率化できる余地があるといえる．

3. 不具合修正時間の効率化を目的としたバグトリアージ手法

本研究では，開発者が一定期間内に修正可能な不具合に関する上限を設けたうえで，プロジェクト全体の不具合修正効率にとって最適となる開発者と不具合の組合せを求める新たなバグトリアージ手法を提案する．そのアプローチとして，本研究では，不具合と開発者の組合せ問題をナップサック問題と見なす．ナップサック問題とは，重みと利得を持つ複数のアイテムを，最大重量が決まっているナップサックに入れる際に，ナップサックに入れたアイテムの総価値が最大となるようなアイテムの組合せを求める問題である．本研究では，ナップサックをプロジェクト全体の修正能力の上限，アイテムを不具合，重みを不具合の修正に要する時間，利得を不具合に対する開発者の適性を数値化したもの（プリファレンス）として考え，ナップサック問題の解法である 0-1 整数計画法 [14] を用いる．

3.1 0-1 整数計画法

0-1 整数計画法は，与えられた条件の下で目的を達成するためにより良い解を求める方法であり，ナップサック問題の解法として知られている．0-1 整数計画法に代表される数理計画法は，近年の計算機の発達により改めて注目されている最適化手法であり，生産問題やスケジューリング問題といったオペレーションズリサーチ分野をはじめとして，ソフトウェア工学の分野でも応用され始めている [15]．ナップサック問題は以下の 0-1 整数計画問題として定式化できる．

$$Maximize: \quad \sum_{i=1}^{n} v_i x_i \tag{1}$$

$$Subject\ to: \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{2}$$

$$x_i \in \{0,1\} \quad (i = 1, 2, \ldots, n) \tag{3}$$

ここで v_i および w_i はそれぞれ i 番目のアイテムの利得と重みを表している．x_i は目的変数と呼ばれ，ナップサック問題の解である．ここでは，そのアイテムを選択するか否か（選択しない：0，選択する：1）を表している．式 (1) は目的関数と呼ばれ，この値の大きさで前述の目的変数の組合せが他の組合せよりも良いものかどうかを判断できる．ナップサック問題における目的関数は選択されたアイテムにおける利得の総和を表し，この値を最大化することを目的とする．一方，式 (2) は重量制限を表した制約条件であり，選択されたアイテムの総重量が最大重量（c）以下でなければならないことを表している．式 (3) はすでに述べた x_i に関する制約であり，この値が 0 または 1 のいずれかしか許されないことを示している．

表 1 本論文で用いる用語一覧
Table 1 Terms used in this study.

用語	記号	意味
カテゴリ	k	不具合票の「コンポーネント」と「優先度」で分類したもの．
プリファレンス	P_{ij}	修正タスクをどの開発者に優先的に割り当てるべきかを示す尺度．P_{ij} とは開発者 D_i がカテゴリ k の不具合修正タスク B_j に対するプリファレンスを示している．$P_{ij} = \frac{\text{カテゴリ}k\text{ における開発者}D_i\text{の修正数}}{\text{カテゴリ}k\text{ における修正タスクの総数}}$
コスト	C_{ij}	開発者 D_i が不具合修正タスク B_j に要する時間．過去に開発者 D_i がカテゴリ k の不具合修正タスクを完了するのに要した修正時間の中央値とした．
上限	L	タスクの集中を防ぐために設定する値
割当て可能時間	T_i	一定期間内に修正可能なタスク量（時間）．T_i は開発者 D_i の担当可能時間を示す．$T_i = \text{上限}\ L - \sum_{j=1}^{n} C_{ij} * x_{ij}$

式 (2)，(3) の制約の下で式 (1) の値が最大となる x_i の組合せを見つけ出すことがナップサック問題の目的である．定式化された 0-1 整数計画問題は，lp_solve[*1] といったソルバで容易に解くことができる．

3.2 0-1 整数計画法に基づくタスク割当て

3.2.1 用語定義

まず，以降の議論を円滑に行うために，本論文で用いる用語を定義する．**表 1** には，用語の一覧をまとめる．

カテゴリ（タスクの分類）：本研究では，開発者 D_i（$i = 1, 2, \ldots, m$）には，不具合修正タスク B_j（$j = 1, 2, \ldots, n$）に対する適性が存在すると想定する．そこでまず，個々の修正タスクをカテゴリ k として分類する．カテゴリ k は，不具合票の「コンポーネント」と「優先度」で定義される．たとえば，対象コンポーネントを “UI” とする不具合票が 2 件あり，それぞれ優先度が “P1” と “P5” とされている場合は別々のカテゴリの修正タスクとして区別される．同様に，同じ優先度であってもコンポーネントが異なる場合は，別カテゴリとして区別される．修正タスクの分類にコンポーネントと優先度を用いる理由は，不具合修正時間がコンポーネントと優先度に依存することを示した先行研究の知見によるものである [16]．ただし，通常 5 段階で表現される優先度は，デフォルトの “P3” が用いられる場合が非常に多いため，本研究では，優先度が “P1” と “P2” を優先度 “高”，“P3” を優先度 “中”，“P4” と “P5” を優先度 “低” とし 3 段階に分けた．

プリファレンス（開発者の適性）：0-1 整数計画法の目的関数では，目的変数に係数を設定する．本研究では係数として，プリファレンス P を用いる．プリファレンスとは，修

[*1] lp_solve 5.5: http://lpsolve.sourceforge.net/5.5/

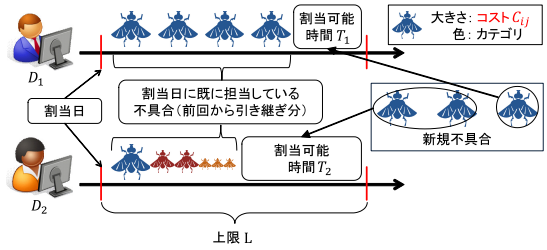

図 1 修正可能なタスク量（時間）の求め方
Fig. 1 The amount of available time for bug fixes.

正タスクをどの開発者に優先的に割り当てるべきかを示す尺度である．本研究では，カテゴリ k の修正タスクの総数に対する開発者 D_i の修正実績数の比をプリファレンスとする．たとえば，“UI” コンポーネントを対象とする優先度 “高” の修正タスクが過去に 10 件存在し，そのうち，開発者 D_i が 5 件を担当したことがあれば，開発者 D_i に対するカテゴリ k（$UI*$ 高）の不具合 B_j のプリファレンス P_{ij} は，0.5 として計算される．

上限：開発者が一定期間内に修正できるタスク量には限りがあると考えるのが自然である．本研究では，開発者 D_i が一定期間に修正可能なタスク量（時間）を考慮した不具合の割当てを行う．**図 1** は，修正可能なタスク量の求め方を示した概略図である．修正可能なタスク量は**割当て可能時間** T_i から求める．また，割当て可能時間 T_i は，あらかじめ設定する**上限** L（日）と，新規の修正タスク割当て時点ですでに開発者 D_i が担当している不具合の**コスト** C_{ij} から求める．コスト C_{ij} は，開発者 D_i が不具合修正タスク B_j に要する時間で，過去に開発者 D_i がカテゴリ k の不具合修正タスクに要した時間の中央値とした．

新規に割り当てる修正タスクのコストの合計が T_i を超えないようにすることで，特定の開発者へ修正タスクが極端に集中するのを防ぐ効果を期待できる．なお，上限 L はプロジェクトによって大きさを変えることができる．また，本論文の実験においては，全開発者に対して同じ上限 L を設定することを想定しているが，実際の運用においては，一定期間内に不具合修正取り組むことのできる時間は開発者ごとに異なるものと思われる．その場合は，上限を開発者ごとに設定（上限 L_i）することも可能である．

なお，先行研究 [6] と同様に，過去の修正タスクに要した個々の修正時間は，以下の式から求めた．

$$\text{修正日数} = \text{修正完了日時} - \text{割当て日時} + 1\text{ 日} \quad (4)$$
*ただし，小数点以下は切り捨てる

3.2.2 定式化

本研究では目的変数，目的関数，制約条件を次のように定義する．

目的変数：x_{ij} とは，開発者 D_i に不具合 B_j を担当させるかどうかを表し，0 の場合は開発者 D_i に不具合 B_j を割り当てないことを，1 の場合は割り当てることを意味する．

$$x_{ij} \in \{0, 1\} \quad (5)$$

目的関数：各不具合に対する各開発者のプリファレンスと目的変数の積の総和を最大化する．個々の開発者の適性に合うタスクがプロジェクト全体として最大となるような組合せを求めることを意味する．

$$Maximize : \sum_{i=1}^{m}\sum_{j=1}^{n} P_{ij}x_{ij} \quad (6)$$

制約条件：本研究では，目的関数に対する制約条件として，以下の 2 つを課する．式 (7) は，一部の有能な開発者にタスクが集中するのを防ぐための制約である．式 (8) は，同一の不具合を複数人の開発者が同時に修正することを避けるための制約である．

- 各開発者に割り当てるタスクに要するコストの合計は割当て可能時間を超えないこと

$$\sum_{j=1}^{n} C_{ij}x_{ij} \le T_i \qquad (i = 1, 2, \ldots, m) \quad (7)$$

- 1 つの不具合を担当する開発者は 1 人以下であること

$$\sum_{i=1}^{m} x_{ij} \le 1 \qquad (j = 1, 2, \ldots, n) \quad (8)$$

3.3 提案手法の適用手順

提案手法の適用手順は以下のとおりである．

Step 1：パラメータの設定 手法の適用前にあらかじめ上限 L を設定する．L の値で各開発者の修正可能時間 T_i を初期化する．

Step 2：プリファレンスとコストの算出 開発者 D_i がカテゴリ k の不具合修正タスク B_j に必要なコスト C_{ij} とプリファレンス P_{ij} をすべて算出する．

Step 3：割当て待ち不具合の追加 前回の割当てを行った日から今回の割当てを行う日までに報告された不具合を割当て待ち不具合として待機させる．

Step 4：0-1 整数計画法の適用 前節で述べた 0-1 整数計画法を用いて，割当て待ち不具合を開発者に割り当てる．割り当てられた不具合は割当て待ち不具合から外す．

Step 5：T_i の更新 Step 4 で割り当てられてた不具合のコスト分だけ各開発者の修正可能時間 T_i を減らす．

Step 6：次の割当て日に進む（Step.2 へ） 次の割当て日（n 日後）まで時間を進め，各開発者の T_i を n（日）増やす．ただし，n（> 0）の値はタスク割当ての状況やニーズによって決まるものであり，一意に定めることは難しい．本論文では議論の一般性を損なわないよう，n は本提案手法の利用者が任意に決定できる自然数であるとする．また，T_i は Step 1 で設定した L より大きくしない．次に Step 2 に戻り，Step 2 から Step 6 を繰り返す．

4. ケーススタディ

本章では，Research Question に取り組むために行った

表 2 データセット
Table 2 Data sets.

プロジェクト	データセット	期間	解決済み不具合（件）	対象不具合数（件）	コンポーネント数	カテゴリー数
Firefox	学習データ	2002/10/1～2011/9/30	10,165	2,536	28	60
	評価データ	2011/10/1～2012/9/30	1,648	659	31	39
Platform	学習データ	2002/10/1～2011/9/30	21,802	1,910	19	28
	評価データ	2011/10/1～2012/9/30	932	578	17	23

表 3 各フィルタリングにおける不具合数
Table 3 Number of bugs in each filtering step.

	フィルタリング	Firefox		Platform	
		学習データ	評価データ	学習データ	評価データ
A	各プロジェクトから収集した不具合	105,294	11,765	78,856	3948
B	A のうち修正時間および修正者が特定でき，かつ修正された不具合	10,165	1,648	21,802	932
C	B のうち，不具合の修正時間が外れ値になっていない不具合	8,699	1,496	18,370	871
D	C のうち，アクティブ開発者が修正した不具合	2,536	659	1,910	578

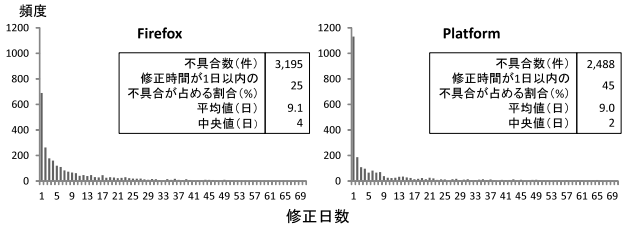

図 2 不具合修正時間のヒストグラムと統計量
Fig. 2 Histograms and statistic of the bug fixing-time.

ケーススタディについて述べる．

4.1 準備

4.1.1 データセット

本論文では，2 つの大規模 OSS プロジェクト（Mozilla Firefox，Eclipse Platform）を対象にしたケーススタディを行う．両プロジェクトは既存研究の多くが分析対象としており [1], [2], [3], [4], [6], [8], [16]，本ケーススタディで得られる結果の妥当性を確保できる．

表 2 にケーススタディで用いるデータセットの概要を，**表 3** にデータセット作成時に行ったフィルタリングの一覧を，**図 2** にデータセットの不具合の修正時間のヒストグラムと統計量を示す．各プロジェクトから収集した不具合データのうち，修正時間および修正者が特定でき，かつ，修正（FIXED）された不具合のみを用いている（図 2 では解決済み不具合に該当する）．なお，不具合報告の中には，数年間放置された後に修正されるものも存在するため，不具合の修正時間の分布を箱ひげ図で確認し，外れ値となる不具合はデータセットから除外している．

本ケーススタディでは，既存手法および提案手法により修正タスクを開発者に割り当てる実験を行うが，OSS プロジェクトの開発者は比較的短期間でプロジェクトを去ることが知られている [17] ため，各プロジェクトに在籍したすべての開発者を対象としてタスク割当てを行うのは現実的でない．また，すべての開発者が活発に不具合修正を行っているわけではない [16] ため，修正タスクを担当できる見込みのある開発者のみにタスクを割り当てる必要がある．そこで本研究では，評価データの最初の日，つまり 2011 年 10 月 1 日を基準とし，半年以内に 6 回以上（1 カ月に 1 回程度を想定）の修正タスクを完了させた開発者を「アクティブ開発者」と定義し，タスク割当ての対象とする（**表 4**）．なお，タスク割当ての精度を保証するために，対象の開発者以外が修正した不具合報告はデータセットから除外した．

表 4 データセットに含まれるアクティブな開発者
Table 4 The number of active developers in the data sets.

プロジェクト	全開発者	アクティブ開発者
Firefox	734	21
Platform	301	23

本ケーススタディでは，2002 年 10 月 1 日～2011 年 9 月 31 日の間に報告された不具合を学習データ，データセットの最後の 1 年である 2011 年 10 月 1 日～2012 年 9 月 30 日の間に報告された不具合を評価データとして，既存手法および提案手法を比較する．

4.1.2 実験の手順

本ケーススタディでは，既存手法および提案手法によりタスクを割り当てる実験を行い，得られた割当て結果を用いて修正時間を算出する．実験の概要を**図 3** に示す．

本実験では，実験上の日付を用意し，その日付に従って不具合報告を再現する．不具合データには報告日時が記されており，報告日時が実験上の日付と同じであれば報告されたと見なし，該当する不具合を提案手法および既存手法で割り当てていく．該当する不具合の割当てが済めば，実験上の日付を進め，再び該当する不具合の割当てを繰り返

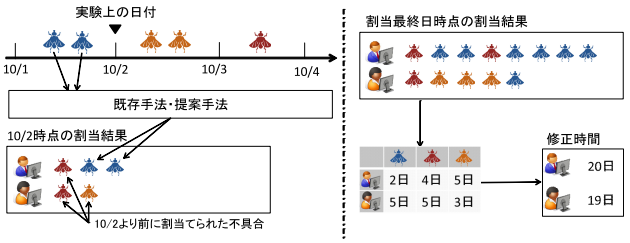

図 3 実験の概要
Fig. 3 An overview of the experiment.

す．本実験では 365 日分の割当てを行う．

すべての日数分の割当てが終了すると，次に修正時間の算出を行う（図 3 右）．提案手法および既存手法を用いると，実際に修正した開発者に割り当てられないことがある．その場合は実際の修正時間が算出できないため，学習データから個々の開発者における各カテゴリの不具合の修正時間の中央値を求め（すなわち，コスト C_{ij}），実験上の修正時間として使用する．

4.1.3 実験環境と設定

本ケーススタディでは，0-1 整数計画法を用いてタスク割当て問題の解を求めるために，オープンソースの数理計画ソフトウェアである lp_solve5.5.2.0 を用いた．また，CPU：Intel Core i7 2.30 GHz，OS：CentOS 6.2，メモリ：2 GB の計算機を用いて実行した．

提案手法を適用するためには，あらかじめ上限 L と，次回の割当て日までの間隔（3.3 節 Step 6 の n）を設定する必要がある．本研究ではデータセットに含まれる不具合の修正に要した時間の第 3 四分位値を求め，その値を切り上げた値とし，Firefox では $L = 10$，Platform では $L = 9$ と設定し，n は 1 日とした．また，既存手法には，Anvik らの機械学習に基づくバグトリアージ手法のうち，最も精度の高い推薦を行える SVM ベースの手法 [3] を用いた．

また，提案手法の適用手順（3.3 節）では Step 6 の後に Step 2 に戻り，コストとプリファレンスの再計算を行う．本実験において再計算を行うことは，評価データを学習データに追加して実験を続けていくことになる．これにより他の手法と実験環境が同じでなくなり，比較結果に影響を与える恐れがある．そこで本実験では Step 6 において，Step 2 に戻るのではなく Step 3 に戻ることとする．

4.2 実験と結果

RQ1：既存手法は，特定の開発者へ負荷を集中させる傾向があるか？ 提案手法ではその問題を緩和できるか？

動機：既存手法は一部の有能な開発者に集中して修正タスクを割り当てる可能性がある．既存手法と提案手法で一部の開発者に修正タスクが集中しすぎないかどうかを確認する．

アプローチ： テストデータにおける全期間（365 日間）と不具合報告の最も多かった月（繁忙期：31 日間）の 2 つの

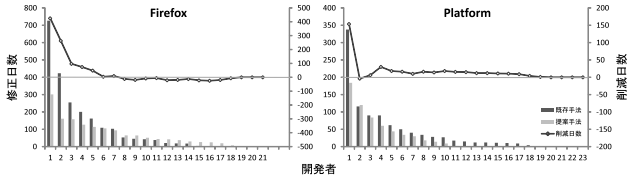

図 4 全期間における各開発者の修正日数（既存手法 vs. 提案手法）
Fig. 4 Distribution of the total cost of assignments by developer in the testing sets.

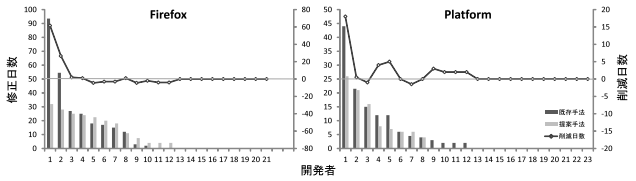

図 5 繁忙期における各開発者の修正日数（既存手法 vs. 提案手法）
Fig. 5 Distribution of the total cost of assignments by developer in a busy time of year.

期間を用いて，既存手法と提案手法によるタスク割当ての結果を比較する．繁忙期を設定した理由は，タスク量の上限を考慮しない既存手法と上限を考慮する提案手法とで，割当て結果に顕著な違いが見られると考えたためである．

結果：図 4 および図 5 はそれぞれ，テストデータにおける全期間および繁忙期に各アクティブ開発者が取り組んだタスク量（修正日数）を示している[*2]．また，図中の削減日数（折れ線グラフ）は提案手法が既存手法と比較し，削減できた修正日数を示している．なお，既存手法と提案手法とでは開発者に割り当てるタスク数が異なるため，図中に示した結果は，両手法で共通に割り当てられたタスクから修正時間を算出し比較したものである（割当て結果の詳細については**表 5** を参照されたい）．

図 4 の全期間での比較では，既存手法を用いた場合，Firefox では 2 人の開発者に合計で設定期間（365 日）以上を要するタスクが割り当てられており，一部の開発者に多くの負荷がかかっていることが見て取れる．また，Platform においても 1 人の開発者に設定期間に近いタスクが割り当てられている．図 5 の繁忙期では，Firefox で 2 人，Platform で 1 人の開発者に合計で設定期間（31 日）以上を要するタスクが割り当てられている．

一方，提案手法を用いた場合，全期間と繁忙期の両期間において，開発者間でタスク量のばらつきは見られるものの，設定期間以上のタスク割当ては生じていない．また，提案手法を用いることで，既存手法において負荷が大きい開発者ほど，修正日数が大きく削減されている（右側の縦軸）ことが分かる．

これらの結果から，提案手法は開発者が修正できるタスク量を考慮して不具合割当てを行っており，一部の開発者

[*2] 横軸の番号は，開発者のタスク量（修正日数）を降順に並べたときの順番を表すものであり，開発者を特定するものではないことに注意されたい．図 4 および図 5 以外の結果も同様である．

表 5 現状の割当て方法，既存手法および提案手法によるタスク割当て結果の比較
Table 5 Comparison of task assignments between the three methods.

プロジェクト		Firefox			Platform		
手法		現状の割当て方法	既存手法	提案手法	現状の割当て方法	既存手法	提案手法
割り当てた不具合（件）	共通化前	659	439	440	578	386	572
	共通化後	377			385		
割り当てた開発者（人）	共通化前	16	17	18	20	19	12
	共通化後	16	15	18	20	19	11
合計修正日数（日）	共通化前	5,816	3,091	1,685	4,893	995	901
	共通化後	2,920	2,220	1,471	3,433	967	599

へタスクが集中するのを緩和できるといえる．

RQ2：提案手法は，プロジェクト全体の不具合修正時間の短縮化に寄与するか？

動機：RQ1 の結果から，提案手法は既存手法に比べ，特定の開発者へのタスク集中を緩和できることが分かった．しかし，プロジェクト全体の不具合修正時間を短縮化できないのであれば，タスク割当てを分散させる意義が薄れてしまう．そこで，提案手法がプロジェクト全体の不具合修正時間を短縮化できるかどうかを検証する．

アプローチ：現状のタスク割当て（OSS プロジェクトで行われている実際のタスク割当て）の修正時間，既存手法と提案手法で実験して得られた修正時間をそれぞれ比較する．

結果：表 5 に，現状の割当て方法による割当て結果，既存手法および提案手法を用いた割当て結果を示す．前述したとおり，それぞれの方法で割り当てられるタスクの数が異なるため，3 つの方法で共通して割り当てたタスクのみにした結果を「共通化後」として示している．比較のため，以下では共通化後の結果について議論する．

Firefox では，プロジェクト全体としての合計修正日数は，現状の割当て方法で 2,920 日，既存手法で 2,220 日，提案手法で 1,471 日となり，既存手法は現状の割当て方法に比べて約 24%，提案手法は現状の割当て方法に比べて約 50%の修正時間を削減できることが分かった．また，提案手法を用いた場合，既存手法に比べ約 34%の修正時間を削減できることが分かった．

Platform ではプロジェクト全体としての合計修正日数は，現状の割当て方法で 3,433 日，既存手法で 967 日，提案手法で 599 日となり，既存手法は現状の割当て方法に比べて約 72%，提案手法は現状の割当て方法に比べて約 83%の修正時間を削減できることが分かった．また，提案手法を用いた場合，既存手法に比べ約 38%の修正時間を削減できることが分かった．

RQ3：提案手法で用いる設定（プリファレンスと上限）が，タスク割当ての最適化にどのように寄与するか？

動機：RQ2 より，提案手法がプロジェクト全体の不具合修正時間の短縮化に効果があることが分かった．しかし，修正タスクに対する開発者の適性と，一定期間内に開発者が取り組めるタスク量の上限を考慮するためにそれぞれ用いたプリファレンス P と上限 L が，タスク集中の回避にどのように寄与するかの詳細については不明なままである．

表 6 条件の有無による 4 つのモデル
Table 6 Four models with and without P and L.

	L：なし	L：あり
P：なし	**モデル A：**プリファレンスと上限を設定せず，修正時間の短い開発者に不具合を優先的に割り当てる	**モデル B：**上限を設定した条件下で，修正時間の短い開発者に不具合を優先的に割り当てる
P：あり	**モデル C：**プリファレンスのみ設定し，カテゴリごとのプリファレンスが高い開発者にタスクを割り当てる	**モデル D：**上限を設定した条件下で，プリファレンスの合計が最大となるように不具合を割り当てる（提案手法）

アプローチ：P の有無，L の有無により 4 種類のモデルを構築し，それぞれのモデルを比較することで，タスク割当て問題におけるプリファレンスおよび上限の効果を調べる．各モデルの特徴は，**表 6** のようにまとめることができる．

結果：表 7 に，各モデルを用いてタスク割当てを行った結果を示す．以下では，P と L がタスク集中の回避にどのように寄与したのかを，表 7 の結果から議論する．

モデル A とモデル B との比較：プリファレンスを設定せず上限の有無のみで比較する．モデル A（L なし）に比べモデル B（L あり）のタスク割当て人数は，両プロジェクトにおいて増加している（Firefox：1 人から 15 人，Platform：1 人から 14 人）．上限の設定はタスク割当てを分散させる効果があるといえる．

モデル A とモデル C との比較：上限を設定せずプリファレンスの有無のみで比較する．モデル A（P なし）に比べモデル C（P あり）のタスク割当て人数は，全プロジェクトにおいて増加している（Firefox：1 人から 9 人，Platform：1 人から 12 人）．プリファレンスの設定により，修正実績数だけではなく修正タスクへの開発者の適性が反映される．上限の設定と同様に，プリファレンスの設定はタスク割当てを分散させる効果がある．

モデル B とモデル D との比較：上限を設定したうえで，プリファレンスの有無で比較する．モデル B（P なし）に比べモデル D（P あり）のタスク割当て人数は，Firefox では増

表 7 モデルごとのタスク割当ての結果
Table 7 Comparison of task assignments between the four models.

プロジェクト		Firefox				Platform			
モデル		A	B	C	D	A	B	C	D
適性		無	無	有	有	無	無	有	有
上限		無	有	無	有	無	有	無	有
割り当てた不具合（件）	共通化前	659	659	559	440	578	578	575	572
	共通化後	440	440	440	440	572	572	572	572
割り当てた開発者（人）	共通化前	1	15	10	18	1	14	12	12
	共通化後	1	15	9	18	1	14	12	12
合計修正日数（日）	共通化前	4,284	3,066	4,897	1,685	5,780	1,186	997	901
	共通化後	2,860	2,022	2,115	1,685	5,720	1,174	910	901

加し，Platform では割当て人数が減少した．（Firefox：15 人から 18 人，Platform：14 人から 12 人）．上限が先に設定されている場合は，プリファレンスの設定はタスクの分散に必ずしも有効ではない．

モデル C とモデル D との比較：プリファレンスを設定し，上限の有無で比較する．モデル C（L なし）に比べモデル D（L あり）のタスク割当て人数は，Platform では変化はなかったが，Firefox では増加している（Firefox：9 人から 18 人，Platform：12 人から 12 人）．モデル D では，プリファレンスの設定に加え，上限の設定により，さらにタスク割当てを分散できたことを示している．プリファレンスを設定したうえで，上限を設定することはタスク分散の効果を高める可能性がある．

以上の結果から，プリファレンスおよび上限の設定は，タスク割当ての分散に効果がある．特に上限の設定がプリファレンスの設定よりタスクの分散に大きく寄与する．

5. 考察

5.1 提案手法の適用範囲

本研究ではプロジェクト全体での不具合修正時間の短縮化を目的とするバグトリアージ手法を提案した．しかし，OSS 開発では，時期ごとに不具合の報告量や開発者数，不具合修正に要する時間は異なるため，提案手法がプロジェクトのあらゆる状況に適しているとは限らない．本節では，提案手法がどのような状況に適しているかを，追加分析の結果とともに考察する．

5.1.1 不具合報告状況への対応

提案手法は，特定の開発者へタスクが集中するのを回避しつつ，可能な限り適任の開発者にタスクを割り当てる．プロジェクトへ報告される不具合が多い状況と少ない状況とで，開発者の負荷がどのように変化するかを確認する．

図 6 は，不具合報告の最も多かった月（繁忙期 31 日間）と，最も少なかった月（閑散期 31 日間）のそれぞれで，各開発者に割り当てられたタスクの量（修正日数）を示したグラフである．なお，提案手法によりタスクが割り当てられなかった一部の開発者は図から省略している．タスクを

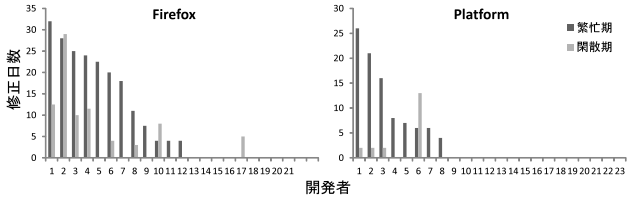

図 6 提案手法を用いた割当て結果（繁忙期 vs. 閑散期）
Fig. 6 Assignment results (busy vs. less busy period).

割り当てられる開発者の数は，両プロジェクトで共通して，繁忙期に比べて閑散期の方が少ないことが見て取れる．また，繁忙期に比べて閑散期では，修正日数のばらつきが大きいことが見て取れる．

図 6 のような結果になった理由は次のように考えられる．閑散期には，開発者に割り当てられるタスク量が上限 L に達しない．そのため，従来手法と同様，最も適任の開発者にのみタスクを割り当てることができる．一方，繁忙期には，開発者に割り当てられるタスク量が上限 L に達する開発者が多いため，上限 L を超える分のタスクが次に適任の開発者に分散して割り当てられる．

これらの結果から，閑散期に提案手法を適用する場合には，最も適任の開発者にのみタスクを割り当てるという，従来のバグトリアージ手法と同様の利点があり，繁忙期に提案手法を適用する場合には，プリファレンスが大きい開発者を中心にタスクを割り当てつつ残りのタスクをその他の開発者に分散させることで開発者の負荷状況をコントロールできるといえる．したがって，繁忙期と閑散期が存在するプロジェクト（たとえば，メジャーバージョンのリリースの前後で不具合報告数が大きく増減するプロジェクトなど）において，提案手法は特に有用であると考える．一方，いずれの開発者も上限 L に達することのないような小規模プロジェクトにおいては，開発者推薦の精度のみを追求した従来のバグトリアージ手法（文献 [3] など）の利用が適していると思われる．

5.1.2 開発者の負荷状況への対応

ケーススタディでは，上限 L の大きさをデータセットに含まれる不具合修正時間の第 3 四分位値（Firefox では 10

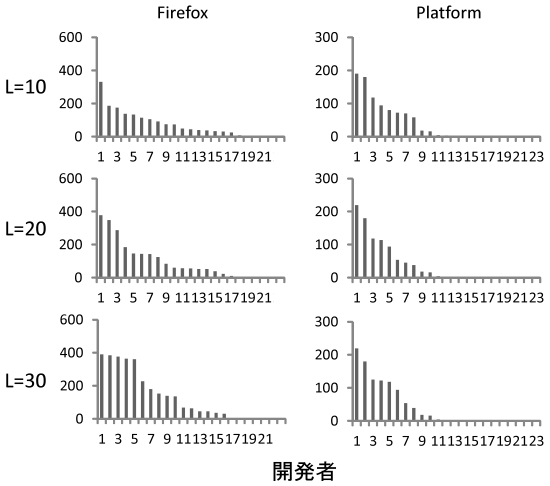

図 7 L の大きさ別割当て結果
Fig. 7 Assignment results of the size of L.

日，Platform では 9 日）として固定した．上限 L の大きさによって開発者の負荷は大きく変わると考えられるため，L の値を変化させてその効果を詳細に調べる．

図 7 は，上限 L の値を変化させたときの開発者のタスク量を示したものである．上限 L を大きくしたときの開発者のタスク量の変化は一様であったため，紙面の都合上，10，20，30（日）の値だけを示す．

Firefox では，L が大きくなるにつれて，タスク量の最も大きい開発者とその他の開発者とのタスク量との差が小さくなることが見て取れる．一方，Platform では，各開発者のタスク量は多少変化するものの，L の変化によりタスク量の分布は大きく変わらないという結果となった．

図 8 は，上限 L と割り当てた不具合数，合計修正日数の関係を示したものである．図から見て取れるように，Firefox では，上限 L が大きくなるにつれ，割り当てた不具合数および合計修正日数が増えるのに対し，Platform では，上限 L が大きくなっても，割り当てた不具合数にはほとんど変化はみられない．また，合計修正日数についても，Firefox ほどの大きな変動はみられない．Platform では，多くのタスクのコストは L より小さいため，L を変化させても一部の開発者に優先して割当てを行うという状況はあまり改善されないことが原因と考えられる．一方，Firefox では，Platform に比べてタスクのコストが L より大きい場合が多く，L を大きくすることで割当て可能な開発者が増えたと考えられる．

以上の結果から，上限 L は本来，開発者への負担を加味しながらプロジェクト管理者が決定すべきであるが，不具合修正に要するタスク量の分布によって，Firefox のように L を大きくすることが望ましいプロジェクトが存在することが分かった．また，Platform のように，不具合修正に要するタスク量が小さいものが多くを占める場合は，L を大きくしても負荷分散の効果が得られない場合もあり，実

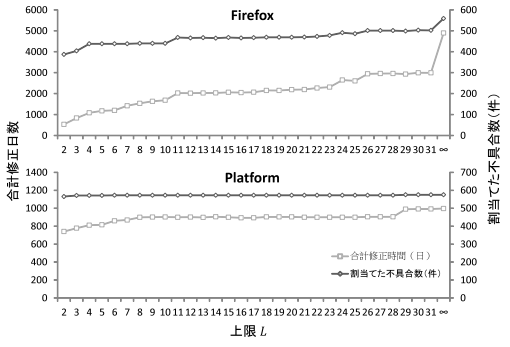

図 8 L の大きさと割り当てた不具合数および合計修正日数の関係
Fig. 8 Relationship among L, assignments, and fixing-time.

表 8 コスト算出方法の妥当性の検証
Table 8 The total costs (actual vs. simulated data).

	現状の割当て方法	実験	差
Firefox	4,451	4,660	−209
Platform	4,787	1,982	2,806

験で行ったような第 3 四分位を用いた機械的な設定ではなく，過去の不具合修正タスクを調査して L を設定するのが望ましいといえる．

5.2 修正時間算出方法の妥当性

まず，修正時間の算出方法の妥当性について検証する．**表 8** は，現状の割当て方法で実際に要した不具合修正時間の合計と，現状の割当て方法で割り当てられた修正タスクを提案手法の修正時間算出方法（不具合修正のコスト C_{ij}：開発者 D_i がカテゴリ k の不具合修正タスク B_j を完了するのに要した時間の中央値）で求めた不具合修正時間の合計とを比較したものである[*3]．

まず，Firefox では，提案手法の方が約 209 日とやや大きく修正時間を算出しているものの，約 4%の誤差であり修正時間の算出方法についてはおおむね妥当であったといえる．

一方，Platform では，提案手法の方が修正時間を 2,800 日以上小さく見積もっている．同じ修正時間算出方法を用いたケーススタディにおいての既存手法と提案手法とを比較した結果と結論自体には影響を与えないが，提案手法を実際に Platform に適用する際には，本論文における修正時間の算出方法は大きな問題があるといえる．したがって，RQ2 の結論は，“提案手法は，現状のタスク割当て方法に比べ Firefox では 50%の不具合修正時間を削減でき（Platform では誤差が大きすぎるため計測不能），既存手法に比べ Firefox では 34%，Platform では 38%の不具合修正時間を削減できることが分かった” とする．

[*3] 提案手法の修正時間算出方法に基づくため，現状の割当て方法で割り当てられたタスクの修正時間を計算できない場合がある．計算可能なタスクのみを対象として比較しているため，表 8 の比較結果は，表 5 の合計修正日数とは異なる数値が示されている．

Platform において修正時間の見積り誤差が大きくなった理由は，Platform では不具合修正時間の分散が大きかったことに起因するものと考えている．図 2 で示したように，Platform では，約 45%の不具合が 1 日以下で修正されている．一方，修正時間の平均値は 9.0 日（図 2）である．そのため，不具合修正のコストが 1 日とした開発者を多く想定することとなり，修正時間が過剰に短く算出されたものと思われる．前述した図 8 からも，コストの過小見積りが原因となって，L が小さな場合でもほとんどの不具合が割り当てられていることが見て取れる．

以上から，Platform のようなコストの小さな不具合が多くを占めるプロジェクトでは，本論文で用いた修正時間算出方法には限界があるといえる．実験の精度向上および実際の適用を考えるうえで，修正時間（コスト）算出方法は今後改良する必要がある．

5.3 目的関数の改良

5.3.1 優先度の重み付けの必要性

提案手法は，不具合修正タスクを分類するカテゴリの作成において，不具合の優先度を 3 段階（高・中・低）に分けて用いるが，個々のタスクの割当てにおいては優先度を考慮していない．そのため，優先度の低いタスクに対して高いプリファレンスを有する開発者にタスク量の上限に達する程度にまで多数割り当てられた場合，その開発者には優先度の高いタスクが割り当てられない場合が生じる可能性がある．タスク割当て自体にはタスクの優先度を考慮しないという点が，実験での評価にどれほどの影響を与えるかを調べるために，**表 9** にデータセットに含まれる不具合の優先度の分布を示す．過半数の不具合の優先度は “中” で，優先度が “高” と “低” の割合は小さいことが見て取れる．多くの大規模 OSS プロジェクトでは，優先度の偏りが顕著にみられることが知られており [16]，Firefox および Platform でも同様の傾向であった．特に，優先度 “低” の不具合の割合は非常に小さいため，前述したような状況はほとんど生じなかったものと考えられる．一方，Eclipse では Firefox に比べ，優先度 “高” の不具合の割合は大きい．優先度が “高” の不具合に対処した開発者は相対的に少ないため，同じ開発者であれば，優先度 “高” が属するカテゴリに対してのプリファレンスは，優先度が “中” のカテゴリに対するプリファレンスよりも高くなる．そのため，同程度のコストであれば，優先度 “高” の不具合は，「優先的に」プリファレンスの高い開発者に割り当てられる可能性が高い．これらのことから，本論文での実験結果に優先度は大きな影響を与えなかったと思われる．ただし，優先度の分布が均等なプロジェクトや優先度の低い不具合が大半を占めるようなプロジェクトでは，提案手法を適用することで前述の問題点が生じる恐れがある．提案手法をより一般性の高いものにするためには，今後，優先度に重み付けを考慮するなどして，優先度の高い不具合がプリファレンスの高い開発者に割り当てられることを保証する必要がある．

5.3.2 コストとプリファレンスのバランス

提案手法の目的は修正時間の削減であるが，修正時間を最小化するようにコストを用いるのではなく，プリファレンスを最大化するように目的関数を定めている．提案手法は，コストに関する制約条件を設けることにより，プリファレンスの総和を最大化するためにプリファレンスが大きくコストが小さい開発者に不具合を優先的に割り当てる．これにより，直接的な修正時間の削減ではないが，修正時間を短くする効果がある．一方，目的関数にコストを加味するようにした場合，直接的に修正時間を短くすることはできると考えられるが，比較的修正が容易な不具合のみを修正している開発者に，プリファレンスの小さな（適性の低い）不具合を多く割り当てる可能性が高まる．結果的に，再割当てを招く原因になると考えられる．実際の運用では，再割当てをある程度許容していても問題はないが，一般的に再割当ては修正時間を遅らせるため，本研究ではまず，再割当てができるだけ発生しないようにすることを前提として修正時間の削減を目指した．目的関数において，再割当てが起きないようなコストとプリファレンスのバランスがとれれば，さらなる修正時間の削減が見込めるため，目的関数を改良したバグトリアージ手法の構築を今後の課題としたい．

5.3.3 学習データの量

本実験では，学習データとして 9 年分のデータを用いたが，修正の回数を重ねるとともに各開発者の役割や能力が変わることも考えられる．ここでは，本実験で用いた学習データの期間が適切であったかを検証する．**表 10** は，表 8 のコスト算出方法の妥当性の検証結果に学習データのうち 2010 年 10 月 1 日から 2011 年 9 月 30 日の 1 年だ

表 9 各優先度における不具合の分布
Table 9 Distribution of bugs by priority.

優先度	Firefox		Platform	
	学習データ	評価データ	学習データ	評価データ
高	33 (2%)	10 (2%)	404 (16%)	176 (27%)
中	1,876 (98%)	568 (98%)	2,060 (81%)	481 (73%)
低	1 (0%)	0 (0%)	72 (3%)	2 (0%)

表 10 学習データの期間の違いによる算出コストの誤差
Table 10 Differences of estimated costs.

プロジェクト	Firefox		Platform	
期間	全期間	1 年	全期間	1 年
現状の割当て方法	4,451	4,451	4,787	4,787
実験	4,660	4,672	1,982	2,939
差	−209	−221	2,805	1,848

けを用いた場合を追加したものである．最後の 1 年だけを使った場合，全期間に比べて Firefox ではわずかながら誤差が大きくなるものの（−209 日（4%）→ −221 日（5%）），Platform では誤差が小さくなった（2,805 日（58%）→ 1,848 日（38%））．なお，割り当てた不具合数に変化はなかった．これらの結果から，提案手法を適用するプロジェクトに依存するものの，学習データは大きければ大きいほど良いとは限らないことが分かった．一方，直近の割当てトレンドを反映するために直近のデータのみを学習データ用いる場合，コストが少数の不具合によって影響を受けやすくなるため，必ずしも小さな学習データが良いとは限らない．学習データの量は，不具合の修正数やカテゴリ数に依存するため，今後の課題として，様々なプロジェクトを対象に，学習データとテストデータの量の関係を検証し，提案手法が最も効果的に適用できるよう学習データの期間を設定する方法を構築したい．なお，今回はコストのみの検証を行ったが，プリファレンスでも同様に，学習データとテストデータの量のバランスについても今後検証する必要がある．

5.4 制約

5.4.1 共通後の割当て結果

本論文では，Mozilla Firefox および Eclipse Platform プロジェクトを対象としたケーススタディを行い，提案手法の有用性を検証した．両プロジェクトにおいて提案手法は既存手法に比べて，タスク集中の緩和とプロジェクト全体としての不具合修正活動の効率化に効果があることが分かった．提案手法の一般性を保証するためには，今後さらにプロジェクトの種類を増やして検証する必要がある．また，本研究では主に，既存手法と提案手法の比較を目的としていたため，共通化後の不具合割当て結果に基づいて議論を行った．しかしながら，既存手法および提案手法はともに，現状の割当て方法に比べて割当て可能な（コストの算出が可能な）不具合の数自体は少なかった．今後は Park ら [6] のコスト算出方法を参考にするなどして，より実際に沿った割当てが行えるよう手法を改良する必要がある．

5.4.2 次回の割当て日までの間隔 n が与える実験への影響

実験では次回の割当て日までの間隔（3.3 節 Step 6 の n）を 1 日とした．n を小さい値にすると，有用な開発者には早く報告された不具合がより配分されやすく，有用な開発者が上限に達し，遅く報告された不具合はそれ以外の開発者に割り当てられやすいというデメリットがある．そのデメリットをふまえたうえ，実験で n を 1 日と設定した理由は，OSS 開発では管理者が毎日，不具合を開発者に割り当てている状況を最適化手法に置き換えることでどれくらい改善するのかを試すためである．一方，n の値が大きければ，割当て待ちとなる不具合が増え，前述のデメリットを改善できると考えられる．本紙では n を大きくした場合の実験を行っておらず，実験結果に与える影響は分かっていない．n の変化による実験結果に与える影響の測定および n の最適な大きさの議論は今後の課題とする．

5.4.3 不具合修正に求められるスキルと不具合修正時間

本論文では，各開発者は同じカテゴリの不具合を同程度の時間で修正できると仮定している．しかし，実際の不具合修正では，同じカテゴリであっても不具合の症状は様々であり，不具合の修正に必要とされる知識やスキルは同じと限らない．したがって，同じカテゴリの不具合でも少なからず修正時間が異なると考えられる．実際，5.2 節で示したように，Platform プロジェクトに対しては本研究で用いた不具合修正時間には，大きな誤差があることを確認した．本論文では，不具合修正時間の算出根拠として，不具合修正時間予測に関する先行研究 [12] から，予測モデルの構築において重要な変数である開発者とカテゴリ（コンポーネントと優先度）を用いた．しかしながら，文献 [12] を含め，現状の不具合修正時間予測モデル（たとえば，文献 [18] や [19]）は必ずしも十分な予測精度があるとはいえないため，不具合修正時間の見積り精度の改善は，提案手法の有用性を向上させるうえでも重要な課題である．

5.4.4 割当ての自動化に際しての新たな仕組みの必要性

本手法ではアクティブ開発者を定義し，アクティブ開発者のみにタスク割当てを行えるようにしている．実際の運用では，アクティブでない開発者にも，修正が可能と思われる不具合を修正してもらう方が効率的である．そのため，アクティブ開発者ではない開発者が，アクティブ開発者に自動割当てされたタスクの中から自分が貢献できそうなタスクを判断し，アクティブ開発者からタスクを譲り受けることを可能にする仕組みを作る必要がある．

6. おわりに

本研究では，大規模 OSS 開発における不具合修正時間の削減を目的としたバグトリアージ手法を提案した．提案手法は，0-1 整数計画法に基づいてタスク割当てを最適化する点に特徴がある．

Mozilla Firefox および Eclipse Platform プロジェクトを対象としたケーススタディを行った結果，提案手法について以下の 3 つの効果が確認できた．

- 特定の開発者へタスクが集中するという問題を緩和できる．
- 現状のタスク割当て方法に比べ Firefox では 50%（Platform では誤差が大きすぎるため計測不能），既存手法に比べ Firefox では 34%，Platform では 38%不具合修正時間を削減できる
- プリファレンスと上限が，タスクの分散効果にそれぞれ同程度寄与する

本研究の今後の課題は，実際のプロジェクトでの適用とより厳密な比較を行うために不具合修正時間（コスト）の

算出方法を改良すること，また，機械学習を用いてプリファレンスの精度を向上させることなどがあげられる．

謝辞 本研究の一部は，独立行政法人情報処理推進機構が実施した「2013 年度ソフトウェア工学分野の先導的研究支援事業」の支援および文部科学省科学研究補助金（基盤（C）：24500041）による助成を受けた．

参考文献

[1] Anvik, J., Hiew, L. and Murphy, G.C.: Who should fix this bug?, *Proc. ICSE 2006*, pp.361–370 (2006).
[2] Jeong, G., Kim, S. and Zimmermann, T.: Improving bug triage with bug tossing graphs, *Proc. ESEC/FSE 2009*, pp.111–120 (2009).
[3] Anvik, J. and Murphy, G.C.: Reducing the effort of bug report triage: Recommenders for development-oriented decisions, *ACM Trans. TOSEM 2011*, Vol.20, No.3, pp.10:1–10:35 (2011).
[4] Bhattacharya, P. and Neamtiu, I.: Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging, *Proc. ICSM 2010*, pp.1–10 (2010).
[5] Guo, P.J., Zimmermann, T., Nagappan, N. and Murphy, B.: "Not my bug!" and other reasons for software bug report reassignments, *Proc. CSCW 2011*, pp.395–404 (2011).
[6] Park, J., Lee, M., Kim, J., Hwang, S. and Kim, S.: COSTRIAGE: A Cost-Aware Triage Algorithm for Bug Reporting Systems, *Proc. AAAI 2011* (2011).
[7] Bortis, G. and van der Hoek, A.: PorchLight: A Tag-based Approach to Bug Triaging, *Proc. ICSE 2013*, pp.342–351 (2013).
[8] Tamrawi, A., Nguyen, T.T., Al-Kofahi, J.M. and Nguyen, T.N.: Fuzzy Set and Cache-based Approach for Bug Triaging, *Proc. ESEC/FSE 2011*, pp.365–375 (2011).
[9] Shokripour, R., Anvik, J., Kasirun, Z.M. and Zamani, S.: Why So Complicated? Simple Term Filtering and Weighting for Location-based Bug Report Assignment Recommendation, *Proc. MSR 2013*, pp.2–11 (2013).
[10] Naguib, H., Narayan, N., Brugge, B. and Helal, D.: Bug report assignee recommendation using activity profiles., *Proc. MSR 2013*, pp.22–30 (2013).
[11] Runeson, P., Alexandersson, M. and Nyholm, O.: Detection of Duplicate Defect Reports Using Natural Language Processing, *Proc. ICSE 2007*, pp.499–510 (2007).
[12] Ohira, M., Hassan, A.E., Osawa, N. and Matsumoto, K.: The Impact of Bug Management Patterns on Bug Fixing: A Case Study of Eclipse Projects, *Proc. ICSM 2012*, pp.264–273 (2012).
[13] Gunn, S.R.: Support Vector Machines for Classification and Regression, Technical report, University of Southampton, Faculty of Engineering, Science and Mathematics; School of Electronics and Computer Science (1998).
[14] 福島雅夫：数理計画法入門，朝倉書店，東京 (1996).
[15] 阿萬裕久：論理的制約条件付き 0-1 計画モデルを用いた重点レビュー計画法，コンピュータソフトウェア（日本ソフトウェア科学会誌），Vol.29, No.2, pp.612–621 (2012).
[16] Mockus, A., Fielding, R.T. and Herbsleb, J.D.: Two Case Studies of Open Source Software Development: Apache and Mozilla, *ACM Trans. TOSEM 2002*, Vol.11, No.3, pp.309–346 (2002).
[17] Bird, C., Gourley, A., Devanbu, P., Swaminathan, A. and Hsu, G.: Open Borders? Immigration in Open Source Projects, *Proc. MSR 2007*, p.6 (2007).
[18] Weiss, C., Premraj, R., Zimmermann, T. and Zeller, A.: How Long Will It Take to Fix This Bug?, *Proc. MSR 2007*, p.1 (2007).
[19] Hewett, R. and Kijsanayothin, P.: On modeling software defect repair time, *Empirical Software Engineering*, Vol.14, No.2, pp.165–186 (2009).

柏 祐太郎 （学生会員）

平成 25 年和歌山大学システム工学部情報通信システム学科卒業．現在，同大学大学院システム工学研究科博士前期課程在学中．学士（工学）．ソフトウェア工学，特にマイニングソフトウェアリポジトリの研究に従事．

大平 雅雄 （正会員）

平成 10 年京都工芸繊維大学工芸学部電子情報工学科卒業，平成 15 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．同大学情報科学研究科助教を経て，平成 24 年和歌山大学システム工学部准教授．博士（工学）．ソフトウェア工学，特にマイニングソフトウェアリポジトリの研究に従事．電子情報通信学会，ヒューマンインターフェース学会，ACM 各会員．

阿萬 裕久 （正会員）

平成 13 年九州工業大学大学院工学研究科電気工学専攻博士後期課程修了．同年愛媛大学工学部助手．平成 19 年同大学大学院理工学研究科講師．平成 25 年より同大学総合情報メディアセンター准教授．ソフトウェアメトリクス，エンピリカルソフトウェア工学に関する研究に従事．博士（工学）．平成 24 年電子情報通信学会情報・システムソサイエティ活動功労賞，平成 25 年情報処理学会山下記念研究賞受賞．電子情報通信学会，日本ソフトウェア科学会，日本知能情報ファジィ学会，IEEE 各会員．

亀井 靖高 （正会員）

平成 17 年関西大学総合情報学部卒業．平成 21 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．平成 22 年カナダ Queen's 大学博士研究員．平成 23 年九州大学大学院システム情報科学研究院助教．博士（工学）．ソフトウェアメトリクス，マイニングソフトウェアリポジトリの研究に従事．電子情報通信学会，IEEE 各会員．