Vol.2015-SE-187 No.8
2015/3/12

開発者の担当可能なタスク量を考慮したバグトリアージ手法の提案

柏　祐太郎[1,a)]　大平　雅雄[1,b)]

概要：本研究では，開発者の担当可能なタスク量を考慮したバグトリアージ手法を提案する．既存手法の多くは，不具合に対する開発者の適性のみを考慮するため，ごく一部の開発者に修正タスクが集中するという問題があった．既存手法に対し提案手法は，開発者の適性に加えて，開発者が一定期間内に修正作業に使える時間の上限を考慮している点に特徴がある．本研究では，不具合の割当問題を，開発者と不具合の組み合わせ問題として捉え，それぞれの開発者に割当てる不具合数に制約条件を課し，マルチナップサック問題として応用することで，最適な組合せを求める．Mozilla Firefox および Eclipse Platform，GNU Gcc 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 2 つの効果が確認できた．(1) 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こと (2) 現状のタスク割当て方法に比べ 36%から 43%の不具合修正時間を削減できること

1. はじめに

大規模・複雑化するシステム開発では，開発者が肥大化するソースコードのすべてを把握できないために，試験工程や運用工程で多数の不具合が検出される．多くのシステム開発では，報告される多数の不具合を管理するために，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一つ一つに対して，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 割の不具合に対して，担当者の再割当が行われており [3]，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，出来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盛んに行われている [1,3,7]．

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．一般的なソフトウェア開発では，リリース時期が定められており，優秀な開発者でも次のリリースまでに使える修正作業の時間は有限であるため，既存手法は現実的でないことが考えられる．

本研究では，開発者の負荷状況を考慮した不具合修正タスクの割当手法を提案する．本研究では不具合の割当問題を，開発者と不具合の組合せ問題として捉え，個々の開発者に割当てる不具合数に制約条件を課し，これをマルチナップサック問題 [6] として応用することで，最適な組合せを求める．制約条件を満たす組合せを求めることで，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．

2. バグトリアージ支援における課題

再割当による不具合修正の長期化は，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 [1,3,7]．以下では，既に提案されている手法を紹介する．

[1] 和歌山大学
Wakayama Uniersity
[a)] s141015@sys.wakayama-u.ac.jp
[b)] masao@sys.wakayama-u.ac.jp

2.1 既存手法

2.1.1 Content-Based Recommendation

Content-Based Recommendation (CBR) [1] の目的は，担当者の再割当が頻発しないようにあらかじめ適任と思われる開発者にタスクを割当てることである．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（Naive bayes や SVM，C4.5）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 [*1]）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．

2.1.2 CosTriage

CosTriage [7] の目的は，CBR より推薦精度をできるだけ落とさずに，修正時間を短縮できる開発者にタスクを割当てることである．具体的には，まず，CBR と同様の方法で，修正対象の不具合に対して各開発者が適任である確率を求める．次に，修正対象の不具合の修正に必要な時間を求める．その上で，精度と修正時間の短縮のどちらにどれだけの比率を置くかを決めた上で，最も適任となる開発者に不具合を推薦する．

CosTriage では，CBR と比べ，推薦精度は約 5%減少したものの，修正時間を 7-31%の削減に成功した．

2.2 本研究の着眼点

本研究では，既存手法の一部の開発者にタスクが集中する問題を受けて，開発者が修正作業に使える時間を考慮する．本研究では，開発者に割り当てるタスクに制約を課した上で，プロジェクト全体で最も効率が良くなる組み合わせを求めることを目標とする．

本研究では，この目標はナップサック問題の目標に近いと考えた．ナップサック問題は，重量制限のあるナップサックに利得と重量を持つアイテムを入れる問題である．つまり，制約の上で，利得を最大にする問題である．本研究では，ナップサック問題よりもナップサックが複数あるマルチナップサック問題の方が適しているため，マルチナップサック問題を応用する．

本手法が構築できれば，既存手法のように不具合と開発者の 1 対 1 の関係において，最も良い開発者を推薦できなくなるかもしれないが，プロジェクト全体としては最も良い組合せを求められることが期待できる．

3. マルチナップサック問題による定式化と解法

本研究では，不具合修正作業の効率化を目的として，開発者の修正作業に使える時間を考慮した新たなバグトリアージ手法を提案する．

3.1 マルチナップサック問題

マルチナップサック問題 (Multiple knapsack problem) 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複数のナップサックに入れる際，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 [6]．一般に知られているナップサック問題とは，ナップサックが複数ある点で異なる．マルチナップサック問題は，アイテムをナップサックに入れるか否かを求めるだけでなく，どのナップサックにいれるかも求める必要があるため，計算量はナップサック問題より比較的大きくなる．マルチナップサック問題は次のように定式化できる．

$$Maximize: \quad \sum_{i=1}^{m}\sum_{j=1}^{n} v_j x_{ij} \tag{1}$$

$$Subject\ to: \quad \sum_{j=1}^{n} w_j x_{ij} \le c_i \quad (i = 1, 2, ..., m) \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} \le 1 \quad (j = 1, 2, ..., n) \tag{3}$$

$$x_{ij} \in \{0, 1\} \quad (j = 1, 2, ..., n) \tag{4}$$

ここで v_j および w_j はそれぞれ j 番目のアイテムの利得と重みを表している．x_{ij} は目的変数と呼ばれ，マルチナップサック問題の解である．ここでは，i 番目のナップサックに j 番目のアイテムを入れるか否か（選択しない：0，選択する：1）を表している．(1) 式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．マルチ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，(2) 式は i 番目のナップサックの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_i）以下でなければならないことを表している．(3) 式はそれぞれのアイテムが一つしか存在しないことを表している．(4) 式は既に述べた x_{ij} 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，つまり，アイテムがナップサックに入っているか否かの状態以外は存在しないことを示している．

(2), (3), (4) 式の制約の下で (1) 式の値が最大となる x_{ij} の組み合わせを見つけ出すことがマルチナップサック問題の目的である．定式化された マルチナップサック問題は，lp_solve [*2] といったソルバーで容易に解くことができる．

3.2 マルチナップサック問題への応用

本論文ではバグトリアージを一種のマルチナップサック

[*1] 割当てる不具合のコンポーネントを過去に修正したことのある開発者に推薦できれば成功としている

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

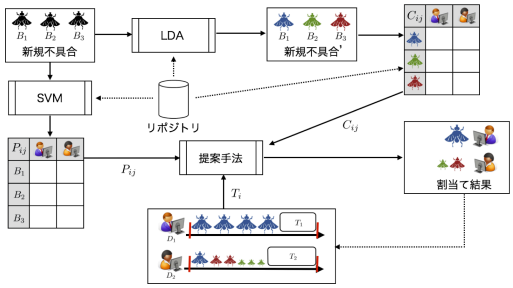

図 1 提案手法の全体像

表 1 本論文で用いる用語一覧

用語	記号	意味
カテゴリ	k	LDA で分類された不具合の種類
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当てるべきかを示す尺度．P_{ij} とは開発者 D_i が不具合修正タスク B_j の修正に適任である確率を示している．
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平均値とした．
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． $T_i = \text{上限 } L - \sum_{j=1}^{n} C_{ij} * x_{ij}$

問題として定式化し，これを解くことでタスク割当ての最適化を行う．マルチナップサック問題では，各ナップサックの重量制約のもと，利得が最大となるアイテムとナップサックの組合せを求めるが，本問題では，各開発者の時間制限のもと，プロジェクト全体で最も効率が良くなる開発者と不具合の組合せを求める．

本問題をマルチナップサック問題として応用する上で，プロジェクトをナップサックの集合，各開発者が**修正作業に使える時間の上限**を各ナップサックの最大重量，不具合をアイテム，不具合の修正に必要な時間（**コスト**）を重み，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**) を利得として考える．以降では本論文で用いる用語である，修正作業に使える時間，コスト，プリファレンスを定義する．また，表 1 に用語の一覧をまとめ，図 1 に本手法の全体像を示す．なお，本論文では開発者の一人一人の適性を反映できるよう，どの開発者がどの不具合を担当するかによってプリファレンスやコストが異なるというモデルになっている点に注意されたい．つまり，一般的なマルチナップサック問題（3.1 節参照）と本問題における変数の形が若干異なっている (v_j が P_{ij}, w_j が C_{ij} となっている)．

3.2.1 プリファレンス（開発者の適性）

マルチナップサック問題の目的関数では，目的変数に係数を設定する．本研究では，修正タスクをどの開発者に優先して割当てるべきかを示す係数として，**プリファレンス** P を用いる．プリファレンスとは，全開発者のうち開発者 D_i が不具合修正タスク B_j の修正に適任である確率（プリファレンス P_{ij}）として定義する．適任である確率を求める方法は，不具合報告の内容とその修正者を分類器で学習させることにより得る．本研究では，不具合報告の内容は様々であることが考えられるため，未知のパターンに対しても正しく識別する確率が高いこと（高い汎化能力）が期待できる SVM を用いる [5] ．

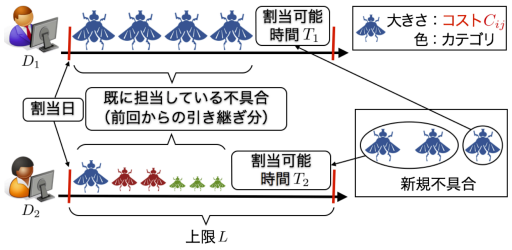

図 2 修正可能なタスク量（時間）の求め方

3.2.2 不具合の修正コスト

不具合修正に必要な時間はどの開発者がどの不具合を修正するかによって異なる．本研究では，開発者 D_i が不具合修正タスク B_j を修正する際に必要とされる時間を**不具合修正コスト** C_{ij} と定義する．本研究では，過去の修正履歴から開発者 D_i が不具合修正タスク B_j と類似した不具合の修正にかかった時間の平均を求め，不具合修正コスト C_{ij} として用いる．不具合修正タスク B_j と類似した種類の不具合であるかの判断するために，Latent Dirichlet Allocation (LDA) [4] を用いて分類する（本研究では，不具合の分類をそれぞれ**カテゴリ** k と呼ぶ)．なお，開発者によっては，修正したことのないカテゴリも存在する．そのままではコストを算出できないので，協調フィルタリング [?] を用いて，算出できないコストを推論して補う．

3.2.3 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 2 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**割当可能時間** T_i から求める．また，割当可能時間 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時点で既に開発者 D_i が担当している不具合のコスト C_{ij} から求める．

新規に割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．

3.3 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．

3.3.1 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 を割当てないことを，1 の場合は割当てることを意味する．

$$x_{ij} \in \{0, 1\} \tag{5}$$

3.3.2 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．

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

3.3.3 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．(7) 式は，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．(8) 式は，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．

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

$$\sum_{j=1}^{n} C_{ij}x_{ij} \leq T_i \qquad (i = 1, 2, ..., m) \tag{7}$$

- 1 つの不具合を担当する開発者は 1 人以下であること

$$\sum_{i=1}^{m} x_{ij} \leq 1 \qquad (j = 1, 2, ..., n) \tag{8}$$

3.4 提案手法の運用手順

提案手法の運用手順は以下の通りである．

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当てる．割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: 次の割当日に進む（Step.2 へ）

次の割当日（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，$n(> 0)$ の値はタスク割り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．

4. 評価実験

4.1 実験の概要と目的

本実験では，提案手法が適任の開発者にタスクの割当てを行い，開発者が修正作業に使える時間を考慮した上で，修正作業を効率化できるかを確認する目的で 3 つの実験を行なう．実験 I では，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当てる可能性があるため，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．実験 II では，不具合修正の長期化を解決するために，既存手法と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を確認する．実験 III では，不具合の割当て問題では正確な開発者にタスクを割当てなければ，再割り当てが起こる問題があるため，既存手法で提案手法で割当てが正確におこなわれているかを確認する．

4.2 データセット

本論文では，4 つの大規模 OSS プロジェクト (Mozilla Firefox , Eclipse Platform , GNU Gcc) を対象にしたケーススタディを行う．各プロジェクトは長期間の開発が続けられており，実験を行うためのデータを十分に取得可能である．また，既存研究の多くが分析対象としているため [1,3,7]，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 4 にデータセットの統計量を示す．

各プロジェクトから収集した不具合データの内，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（表 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はない．また，すべての開発者が活発に不具合修正を行って

表 2　データセット

プロジェクト	データセット	期間	対象不具合数 (件)
Firefox	学習データ	2010/7/5〜2011/7/4	1,043
	評価データ	2011/7/5〜2011/9/27	142
Platform	学習データ	2010/3/22〜2011/3/21	783
	評価データ	2011/3/22〜2011/6/22	168
Gcc	学習データ	2009/12/25〜2010/12/24	940
	評価データ	2010/12/25〜2011/3/25	250

表 3　データセットに含まれるアクティブな開発者

プロジェクト	全開発者	アクティブ開発者
Firefox	215	19
Platform	61	20
Gcc	97	23

表 4　データセットにおける不具合修正時間の統計量

プロジェクト	Firefox	Platform	Gcc
不具合数	1,185	951	1,190
修正時間が 2 日未満である不具合が占める割合（%）	19.2	49.2	51.2
修正時間の平均値（日）	12.5	6	4.4
修正時間の中央値（日）	7	2	1.9
修正時間の最小値（日）	1	1	1
修正時間の最大値（日）	59.9	38.5	27.5

いる訳ではないため，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を基準とし，半年以内に 6 回以上（一ヶ月に 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 3）．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．

4.3　評価方法

提案手法に対して 3 つの評価項目を設定し，それぞれの評価方法を以下に説明する．

- タスク集中の緩和効果
  既存手法および提案手法で実験して得られた各開発者のタスク量（修正時間）が，一部の開発者に偏っていないかを確認する．それぞれのプロジェクトにおける評価データの期間よりも修正時間が超過していないことが望ましいとする．
- プロジェクト全体における修正時間の削減効果
  既存手法および提案手法を用いることで不具合修正活動を効率化できるかを確認する．評価項目として，現状のタスク割当ての修正時間（修正履歴に記録されている修正時間）と比較し，既存手法と提案手法の修正時間が削減されているか，および，リリースまでに修正できた不具合数で比較する．

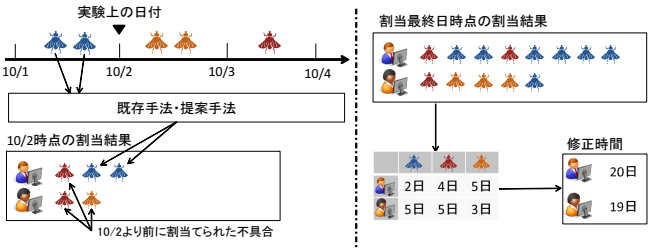

図 3　実験の概要

- 割当ての精度
  CBR はプリファレンスが最も大きい開発者に割り当てるが，提案手法と CosTriage は，コストと開発者が使える時間にも左右されるため，プリファレンスが最も大きい開発者に割当てるとは限らない．従って，提案手法と CosTriage は割当ての精度が下がると想定できる．3 つめの評価項目では，提案手法と CosTriage は，CBR と比較して，割当ての精度を落とさずに，それぞれの手法のメリットを享受できるかを確認する．割当ての精度とは，割当てを行った回数に対する成功した割合とし，割当ての成功は割当てる不具合が属するコンポーネントを担当したことがある開発者に推薦した場合とする．

4.4　比較対象

本実験の比較対象として，実際の修正時間（以下，現状の割当手法）と 2 つの既存手法（CBR，CosTriage）を比較する．CBR と CosTriage で用いられている機械学習アルゴリズムの内，最も精度の高い推薦を行える SVM ベースの手法 [1] を用いた．

4.5　実験手順

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当てる実験を行い，得られた割当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す．本実験では 365 日分の割当てを行う．

全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．

4.5.1 実験の設定

4.5.2 パラメータの設定

提案手法を適用するためには，あらかじめ上限 L と，次回の割当日までの間隔（3.4 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=14，Platform では L=7，Gcc では L=6 と設定した．なお，割当てを行う間隔 n は 1 日とした．また，LDA を適用するに当たって，あらかじめ何種類に分類するかを決める必要がある．本研究では，Arun らの手法 [2] を用いて，最適なトピック数（Firefox：7，Platform：12，Gcc：11）を求めた．

4.5.3 実験の設定

提案手法の適用手順（3.4 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．

先行研究 [7] と同様に，過去の修正タスク個々の修正時間は，以下の式から求めた．割当日時は不具合修正を完了させた開発者に割当てられた日時を指す．本研究の修正時間にはその開発者に割当てられる以前の修正時間（再割当の時間）は含めないとする．

$$\text{修正日数} = \text{修正完了日時} - \text{割当日時} + 1\text{ 日} \quad (9)$$

*ただし，小数点以下は切り捨てる

5. 実験結果

5.1 前実験：修正時間算出方法の評価

本実験において，不具合を実際に修正した開発者以外にタスクを割当てた場合，修正時間を確認することができない．そのため，本実験ではコストを修正時間として代用する．コストを修正時間として代用するにあたり，コストが修正時間として代用することが妥当であるかを前実験として確認する．

確認方法は，まず修正履歴から 2 つの情報（誰がどの不具合を修正したかの情報とその修正に要した時間）を用意する．そして，前者の情報とコストを用いて，実験と同様の方法で修正時間を算出する．そして，得られた修正時間と後者の情報（実際の修正時間）を比較する．この 2 つの修正時間の差が小さいほど，本実験における修正時間の算出は妥当なものであると考える．

前実験の結果を表 5 に示す．Firefox では 110.0 日（6%），Platform では 68.0 日（6%），Gcc では 123.7 日（11%）の誤差が見られた．不具合 1 件あたりの誤差では，全プロジェクトにおいて 1 日未満であり，概ね妥当であるといえる．

表 5 コスト算出方法の妥当性の検証

	現状の割当方法	シミュレーション	差
Firefox	1,702	1,812	110
Platform	1,003	935	-68
Gcc	1,089	1,213	124

5.2 実験 I：タスク集中の緩和

図 4 は，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 [*3]．

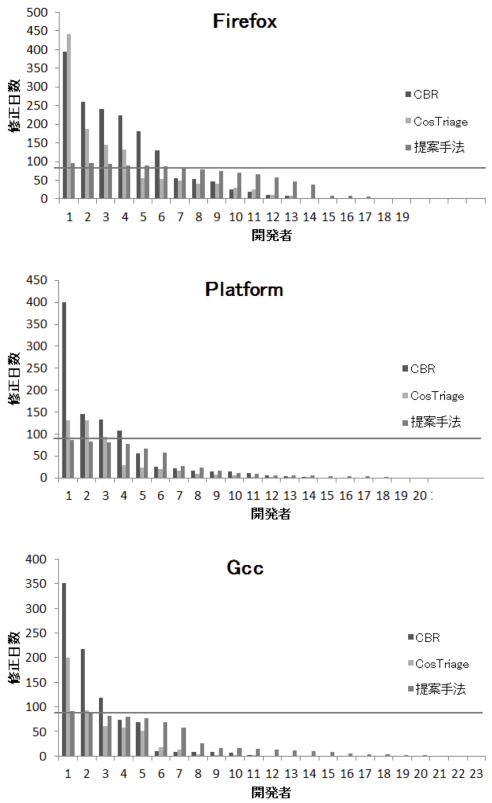

図 4 各開発者の修正日数（既存手法 vs. 提案手法）

CBR を用いた場合，リリースまでの時期である評価データの期間（Firefox：12 週，Platform および Gcc：3 ヶ月）以上を要するタスクが割当てられた開発者が，Firefox では 6 人，Platform では 4 人，Gcc では 3 人となっており，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，CosTriage を用いた場合でも，Firefox では 4 人，Platform では 3 人，Gcc では 2 人となっており，CBR よりは人数は少ないものの，依然としてタスクが集中していることがわかる．一方，提案手法を用いた場合，Firefox では 7 人，Platform では 0 人，Gcc では 1 人となっており，Platform と Gcc では，既存手法よりもタスクが集中した人数が少なくなったが，Firefox では逆に既存手法よりも人数が多くなった．この原因を確認すると，リリース

*3 横軸の番号は，開発者のタスク量（修正日数）を降順に並べた時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．

直前に報告された不具合を割り当てたために，設定期間よりも修正時間が少し大きくなったためである．また，提案手法と既存手法でタスクが集中した開発者の修正時間をみると，既存手法において負荷が大きい開発者ほど，提案手法における開発者の修正日数が大きく削減されていることが分かる．

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，一部の開発者へタスクが集中するのを緩和できるといえる．

5.3 実験 II：修正時間の削減

表 6 に，現状の割当方法による割当結果，CBR，CosTriage および提案手法をそれぞれ用いた場合のプロジェクトの修正時間を示す．また，表 7 には，各手法でリリースまでに修正した不具合数を示す．なお，表 7 の括弧内の数字はリリースまでに修正できなかった不具合数である．

Firefox では，プロジェクト全体としての合計修正日数は，現状の割当方法で 1,702 日，CBR で 1,647 日，Costriage で 1,215 日，提案手法で 1,092 日となり，CBR は現状の割当方法に比べて約 3%，Costriage は現状の割当方法に比べて約 29%，提案手法は現状の割当方法に比べて約 36%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，CBR に比べ約 34%，CosTriage に比べ約 10%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，現状の割当方法で 1,003 日，CBR で 957 日，Costriage で 477 日，提案手法で 576 日となり，CBR は現状の割当方法に比べて約 5%，Costriage は現状の割当方法に比べて約 52%，提案手法は現状の割当方法に比べて約 43%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，CBR に比べ約 40%の修正時間を削減でき，一方，CosTriage に比べ約 21%の修正時間が増加することが分かった．

Gcc ではプロジェクト全体としての合計修正日数は，現状の割当方法で 1,085 日，CBR で 875 日，Costriage で 501 日，提案手法で 676 日となり，CBR は現状の割当方法に比べて約 19%，Costriage は現状の割当方法に比べて約 54%，提案手法は現状の割当方法に比べて約 38%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，CBR に比べ約 23%の修正時間を削減でき，一方，CosTriage に比べ約 35%の修正時間が増加することが分かった．

次に，リリースまでに修正できた不具合数を確認すると，Firefox では，CBR が 53 件，CosTriage が 56 件，提案手法が 119 件であった．次に Platform では，CBR が 79 件，CosTriage が 123 件，提案手法が 158 件であった．最後に Gcc では，CBR が 108 件，CosTriage が 166 件，提案手法が 226 件であった．ここから，提案手法は修正時間の削減効果では，CosTriage と同等もしくはそれ以下であったが，リリースを考慮すると，プロジェクトの全体として効率化する効果が CosTriage よりもあるといえる．

表 6 現状の割当方法，既存手法および提案手法によるタスク割当結果の比較

プロジェクト	Firefox			
手法	現状の割当方法	CBR	Cos Triage	提案手法
割当てた不具合（件）		142		
割当てた開発者（人）	15	13	13	17
合計修正時間（日）	1,702	1,647	1,215	1,092

プロジェクト	Platform			
手法	現状の割当方法	CBR	Cos Triage	提案手法
割当てた不具合（件）		168		
割当てた開発者（人）	19	14	14	18
合計修正時間（日）	1,003	957	477	576

プロジェクト	Gcc			
手法	現状の割当方法	CBR	Cos Triage	提案手法
割当てた不具合（件）		250		
割当てた開発者（人）	19	11	11	20
合計修正時間（日）	1,085	875	501	676

表 7 リリースまでに修正できた不具合数

	CBR	CosTriage	提案手法
Firefox	53(89)	56(86)	119(23)
Eclipse	79(89)	123(45)	158(10)
Gcc	108(142)	166(84)	226(24)

5.4 実験 III：割当ての正確さ

表 8 に，CBR，CosTriage および提案手法をそれぞれ用いた場合の割り当ての精度を示す．Firefox における割当ての精度は，CBR が 88.7%，CosTriage が 93.4%，提案手法が 62.7%であった．次に Platform では，CBR が 64.9%，CosTriage が 48.8%，提案手法が 42.3%であった．最後に Gcc では，CBR が 82.8%，CosTriage が 66.4%，提案手法が 66.0%であった．3 つの手法ごとに精度の平均値をとると，CBR が 78.8%，CosTriage が 69.6%，提案手法が 57%であり，プリファレンスが最大である開発者に割当てる CBR が最も精度が高かった．また，CosTriage は CBR に比べて 13 %の精度が減少し，提案手法は CBR に比べて 38%の精度が減少した．

6. 考察

6.1 妥当性の検証

ここでは，本研究が用いた 3 つの設定（プリファレンス，上限 L，割当間隔 n）が妥当であったかを確認する．

表 8　タスク割当ての精度（%）

	CBR	CosTriage	提案手法
Firefox	88.7	93.7	62.7
Platform	64.9	48.8	42.3
Gcc	82.8	66.4	66.0

6.1.1 プリファレンス

本手法では，不具合報告の内容を入力として機械学習に与え，開発者が適任である確率を求め，プリファレンスとして開発者に割当てるべき尺度として用いた．

バグトリアージでは，再割当が頻発することによる不具合修正の長期化が問題とされている．本研究でも，再割当は無視できない問題である．しかしながら，本手法における再割当の防止方法は，機械学習を用いて，適任の開発者にタスクを割当てることのみである．現在，この方法がどの程度，再割当を防止できるかがわかっていないが，さらに再割当の発生を防ぐには，ペナルティとしてプリファレンス与えることができる．再割当が発生する確率を開発者ごとに算出し，再割当が発生する確率（正確には，1− 再割当が発生する確率）とプリファレンスの積をとることにより，適性が高く，再割当が発生しにくい開発者を中心に割当てを行うことができる．

6.1.2 上限 L

本実験では上限 L を，Firefox では 14，Platform では 7，Gcc では 6 と設定した．しかしながら，上限 L の大きさがプロジェクトに対して，どのような影響を与えるかわかっていない．ここでは，上限 L の大きさがプロジェクトにどのような影響を与えるかを確かめるために，上限 L の大きさごとにプリファレンスの合計と合計修正時間がどのように変化するかを確認する．図 5 は，各上限 L ごとのプリファレンスの合計と合計修正時間である．Firefox では，L が 5 から 16 ぐらいまで，プリファレンスの合計と合計修正時間が徐々に大きくなる．そして，L が 17 を超えたあたりから変化がなくなる．Platform と Gcc では，L が 2 から 5 までに，プリファレンスの合計と合計修正時間が急激に大きくなり，その後は大きな変動はなくなっている．

本実験で設定した上限 L は，プリファレンスの合計と合計修正日数の変化が少し小さくなり始めた頃であり，この前後であれば，実験で示したような結果が得られると思われる．そして，さらに上限 L を大きくすれば，CBR のような結果に近づいていく．リリースまで時間がある場合は，上限を大きくして，最も適任の開発者にタスクを割当てるということも可能である．一方，リリース直前など，数多く不具合を修正したい場合，上限 L を小さくすることで修正時間を大きく削減できるかもしれないが，割当ての精度も著しく下がると考えられ，再割当が発生する原因と成りうるため，注意が必要である．

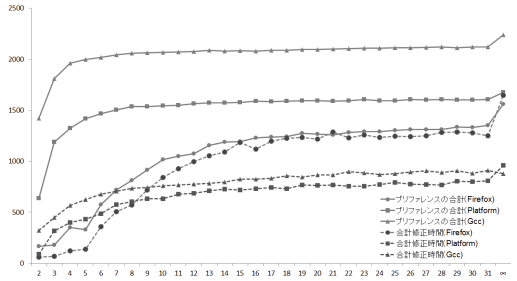

図 5　L の大きさとプリファレンスの合計および合計修正日数の関係

7. まとめと今後の課題

本研究では，開発者の担当可能なタスク量を考慮したバグトリアージ手法を提案した．本研究では不具合の割り当て問題を，開発者と不具合の組み合わせ問題として捉え，ここの開発者に割当てる不具合数に制約条件を課し，マルチナップサック問題として応用することで，最適な組合せを求めた．Mozilla Firefox および Eclipse Platform，GNU Gcc 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 2 つの効果が確認できた．(1) 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こと（2）現状のタスク割当て方法に比べ 36%から 43%の不具合修正時間を削減できること

謝辞　本研究の一部は，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. of ICSE 2006*, pp. 361–370 (2006).
[2] Arun, R., Suresh, V., Veni Madhavan, C. E. and Narasimha Murthy, M. N.: On Finding the Natural Number of Topics with Latent Dirichlet Allocation: Some Observations, *Proc. of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, Vol. I, pp. 391–402 (2010).
[3] Bhattacharya, P. and Neamtiu, I.: Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging, *Proc. of ICSM 2010*, pp. 1–10 (2010).
[4] Blei, D. M., Ng, A. Y. and Jordan, M. I.: Latent Dirichlet Allocation, *Machine Learning Research*, Vol. 3, pp. 993–1022 (2003).
[5] 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, University of Southampton (1998).
[6] Martello, S. and Toth, P.: *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, Inc., New York, NY, USA (1990).
[7] Park, J., Lee, M., Kim, Jinhan, H. S. and Kim, S.: COSTRIAGE: A Cost-Aware Triage Algorithm for Bug Reporting Systems, *Proc. of AAAI 2011* (2011).