CrowdMashup: Recommending Crowdsourcing Teams for Mashup Development

Faisal Binzagr and Brahim Medjahed(✉)

Department of Computer and Information Science,
University of Michigan - Dearborn, Dearborn, USA
{faisalb,brahim}@umich.edu

Abstract. Mashups involve the collaboration of multiple developers to build Web applications out of pre-existing APIs. A large body of research focused on recommending APIs for mashups. However, very few contributions looked at recommending developers. In this paper, we propose *CrowdMashup*, a crowdsourcing approach for mashup teams recommendation. We analyze online developer communities and API directories to infer developers' interests in APIs through natural language processing. We predict missing interest values using the alternating least square method for collaborative filtering. We also model interactions (comments and replies) among developers as a weighted undirected graph and introduce a sociometric to identify socially related developers. We propose an algorithm, based on the concept of cliques in graph theory, that combines developers' skills and sociometric to recommend efficient and balanced teams. We describe a prototype implementation and conduct extensive experiments on real-world data and APIs to evaluate our approach.

Keywords: Mashup · Crowdsourcing · Team recommendation
Sociometric · Skills

1 Introduction

The past decade has witnessed an increasing interest in *mashup* development [4]. For instance, the popular `programmableWeb`[1] API directory currently includes about 8,000 mashups. *Mashups* are Web applications that aggregate pre-existing APIs (or services) to create valuable services with added functionality [27]. Mashup development generally involves several APIs requiring a variety of technological skills such as REST, SOAP, JSON, XML, and security. This often calls for the collaboration of multiple developers to reduce the overall mashup cost (e.g., development time). A large body of research focused on recommending APIs for mashups [13]. However, very few contributions looked at recommending

[1] https://programmableweb.com.

https://doi.org/10.1007/978-3-030-03596-9_49

developers to be part of mashup development teams. With the substantial number of available APIs and programmers, finding skilled mashup developers is not straightforward. For instance, `programmableWeb` lists more than 19,000 APIs. The `StackOverflow`[2] and `GitHub`[3] developer community platforms report estimated 9 and 27 millions subscribers, respectively. Besides, the software industry has recently seen a new trend where *crowdsourcing* companies (e.g., `Topcoder`[4]) sell services to corporate, mid-size, and small-business clients, and pay community members (i.e., developers) for their work. These companies also organize open tournaments and programming challenges in which programmers are organized in teams to compete against each other. Therefore, it is important to form balanced teams with skilled developers.

Crowdsourcing is a powerful sourcing model to perform a broad range of hard tasks by splitting the work between workers [22]. It has been used in software development to perform vital activities such as implementation, design, coding, or testing [24]. Selecting appropriate developers should be performed carefully to improve productivity [20]. In the context of mashups, two factors contribute to successful developer recommendation. First, mashups involve various APIs that require a large array of *skills.* A recent study shows that the interest of project members toward specific tasks leads to better outcomes [15]. Hence, it is vital to pick developers that possess the right skills, demonstrate significant interest in the mashup, and have a good reputation among their peers. Second, it is necessary to form teams with members that can get along with each other. Studies have confirmed that strong social relationships among members increase team performance [10]. Most interactions among mashup and API developers take place via online communities such as `StackOverflow` and `GitHub`. Positive discussions between developers, through questions and answers, tend to increase their social ability and productivity.

In this paper, we propose *CrowdMashup*, a crowdsourcing-based approach for recommending teams of developers for mashups. We analyze `StackOverflow` and `programmableWeb` to generate teams that best statisfy mashup requirements. To the best of our knowledge, this is the first work to address recommendation in mashups from developer's perspective. The main contributions of the paper are summarized below:

– We use natural language processing [17] to assign *interest scores* to developers in using APIs. As developers may omit to comment on certain APIs, we predict *missing scores* using the alternating least square method for collaborative filtering [21]. We combine the computed interest scores and reputation values of developers in the community to quantify their *skills.*
– We define a *sociometric* to assess social relationships among developers in the community. *Sociometry* is a quantitative method in psychology for measuring social relationships [26]. We model interactions (comments and replies) among

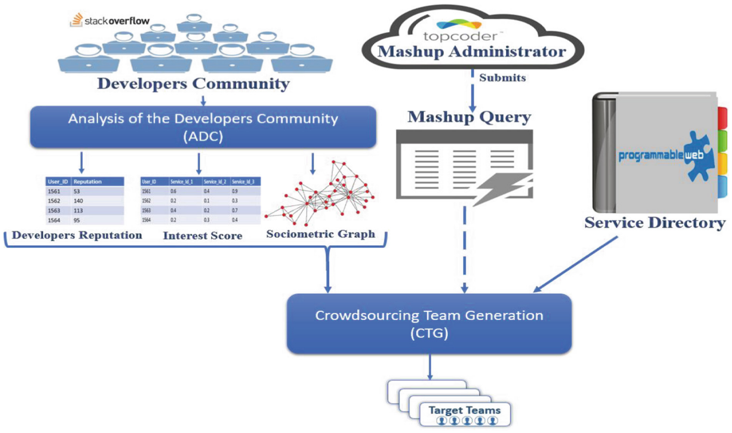
[2] https://stackoverflow.com.
[3] https://github.com.
[4] https://topcoder.com.

developers as a weighted undirected graph. The weight of each edge represents the number of interactions between developers modeled as nodes.
- We propose an algorithm to generate teams from *mashup queries*. The query is a specification of the mashup requirements. We adopt the concept of *cliques* from graph theory to identify strongly related developers [5]. A *clique* is a subset of vertices from the sociometric graph where every two distinct vertices are adjacent. We compare the skills of the developers in the clique along with their sociometric scores to recommend *top-t teams*. We also describe a prototype implementation and conduct experiments on real-world data and APIs to evaluate our algorithm.

The rest of this paper is organized as follows. We propose the *CrowdMashup* approach in Sect. 2. We describe the implementation and performance study in Sect. 3. In Sect. 4, we overview related work. We conclude in Sect. 5.

2 The CrowdMashup Approach

The *CrowdMashup* architecture (Fig. 1) is composed of two major components: *Analysis of the Developer Community* (ADC) and *Crowdsourcing Team Generation* (CTG).



User_ID	Reputation
1561	53
1562	140
1563	113
1564	95

User_ID	Service_Id_1	Service_Id_2	Service_Id_3
1561	0.6	0.4	0.9
1562	0.2	0.1	0.3
1563	0.4	0.2	0.7
1564	0.2	0.3	0.4

Fig. 1. CrowdMashup architecture

ADC runs *offline*, i.e., independently of any request to create mashup development teams. It analyzes the `StackOverflow` community to calculate and predict the interest of developers in adopting and using APIs. Nowadays developer communities become a troubleshooting manual, where many developers share experiences, issues, and solutions [18]. For instance, `StackOverflow` has more than 16 million questions and 24 million answers in 2018. The `LinkedIn` API uses

StackOverflow as a reference, at their official page, to support programmers in technical issues. Developer communities also showcase the level of affinity among developers. Many programmers may end up collaborating in projects as a result of their interactions in online communities [10].

CTG runs *online* at the reception of a *mashup query* from the *mashup administrator*. It returns efficient teams that best satisfy the mashup query requirements. The *mashup administrator* is a user or entity looking for teams of developers to collaborate on a mashup. Topcoder is an example of potential mashup administrator. It offers software development services to third party clients, contracting individual community programmers to work on specific tasks. It also holds design competition, thus offering design services to clients.

2.1 Analysis of the Developer Community (ADC)

ADC analyzes StackOverflow to generate three data structures (Fig. 1): *interests table* (U_I), *reputation table* ($\hat{U_R}$) and *sociometric graph* (SG).

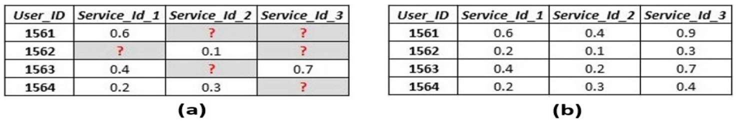
User Interests Table (U_I) - The initial step before analyzing the developer community is to prepare the list of APIs used in that community. To that end, we crawled all APIs from programmableWeb and extracted the *name* and *primary category* of each service using the Scrapy framework[5]. Since StackOverflow has about 66 millions comments (questions and answers), we focused on the ones that are related to APIs. We filtered StackOverflow comments using the API names retrieved from programmableWeb.

The next step is to analyze developers' comments and assign scores of interest in using APIs. For that purpose, we applied sentiment analysis to get the interest score $U_I(u_i)$ for each user u_i. We parsed comments using Stanford NLP[6] parser, which utilizes *recursive neural networks* (tree-structured models) for sentiment analysis [17]. For example, the comment "... Google Visualization API has several ways to do each task so it's important to know what you have already done and we could start there..." returns a positive interest value about Google Visualization API. An example of negative interest about Google Maps API is: "I simply have no experience with the Google Maps API ...".

Since certain APIs are not discussed by some developers, we ended-up with missing interest scores (Fig. 2). To solve this problem, we utilized the *Alternating Least Squares* (ALS) collaborative filtering technique [21]. In ALS, developers and their scores are described by a small set of latent factors used to predict the missing interest scores for all developers. Accordingly, we completed interest scores for all developers and APIs as shown in Fig. 2. If an API is listed on programmableWeb but unknown (i.e., not discussed) on StackOverflow, then ALS cannot complete the missing interest scores for this API. To deal with this issue, we average the interest scores of u_i for all APIs on StackOverflow that have the same category as the unknown API. Then, we assign the average score

[5] https://scrapy.org/.
[6] https://nlp.stanford.edu/.

as u_i's interest score for this API. If no API with similar category is commented by u_i, we average u_i's interest scores for all APIs discussed by u_i (Fig. 2).



User_ID	Service_Id_1	Service_Id_2	Service_Id_3
1561	0.6	?	?
1562	?	0.1	?
1563	0.4	?	0.7
1564	0.2	0.3	?

(a)

User_ID	Service_Id_1	Service_Id_2	Service_Id_3
1561	0.6	0.4	0.9
1562	0.2	0.1	0.3
1563	0.4	0.2	0.7
1564	0.2	0.3	0.4

(b)

Fig. 2. Interests table

User Reputation Table ($\hat{U_R}$) - `StackOverflow` has a reputation system which provides the level of expertise $U_R(u_i)$ for each user u_i. Since the extracted reputation has highly distributed values, we applied the *z-score* normalization to write reputation values into a standardized structure. The following formula shows the final reputation $\hat{U_R}$ for u_i, where μ and σ represent the mean and standard deviation of all reputation values, respectively:

$$\hat{U_R}(u_i) = \frac{U_R(u_i) - \mu}{\sigma}$$

Sociometric Graph (SG) - Another major aspect in teams formation is the social ability, or *sociometry*, among developers [26]. The idea is to make sure that members of the same team can actually work together. Studies showed that social relationships among members of the same team have a positive impact on improving the team productivity [3]. In our approach, we use interactions among developers via questions and replies in `StackOverflow` as a mean to estimate their social relationships. Developers that engage in more conversations with each other in online communities have more chances to successfully collaborate.

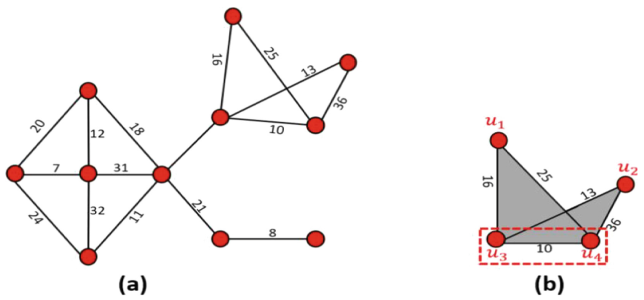


Fig. 3. Sociometric graph

We scanned the history of interactions among developers in `StackOverflow` regardless if the questions/replies are related to APIs or not. Then, we modeled those interactions as an undirected weighted graph, called *sociometric graph*

(SG). Each node in the graph represents a user (Fig. 3a). An edge (u_i, u_j) states an existing interaction (question or reply) between users u_i and u_j. Developers may interact at various levels, from few questions/replies to thousands. To capture this aspect, we label each edge (u_i, u_j) with a weight $W_e(u_i, u_j)$ that gives the number of interactions between users u_i and u_j:

$$W_e(u_i, u_j) = \#\ interactions\ between\ (u_i, u_j)$$

2.2 Mashup Query Specification

Mashup administrators interact with *CrowdMashup* through *mashup queries.* A *mashup query* Q defines the mashup requirements through a tuple $Q = (t, m, A)$ where:

- t: is the number of required teams.
- m: is the number of members within each team.
- A: is a list of APIs that compose the mashup.

Each element in the list A is defined as $<API_{ID}, API_w>$. API_{ID} is an ID that uniquely identifies the API. API_w is the weight (in the range 0 to 1) of the API. It represents the level of importance of the corresponding API in the mashup. For instance, a location-based mashup (e.g., transportation) may rely on a mapping API; the mapping API should be given a significant weight value to make sure the most skilled developers are recommended for this API. A small API_w implies that the API may not be mastered by all teams members; a big API_w indicates that the API should be mastered by most teams members.

Example 1. Assume we want to build 5 teams of 3 developers for a mashup that composes `GoogleMaps` (with ID 1 and weight 0.6), `Foursquare` (with ID 4 and weight 0.4) and `Last.fm` (with ID 5 and weight 0.1). The mashup query is specified by $t = 5$, $m = 3$, and $A =$ $[<1, 0.6>, <3, 0.4>, <5, 0.1>]$.

Mashup administrators may not want to limit mashups to specific APIs by providing the list of API categories instead of APIs. For instance, they may refer to "Social" as a required category instead of `Facebook` or `Twitter`. In this case, we automatically fetch all APIs that belong to the categories listed by the administrator from `programmableWeb` and replace each category by the matching APIs.

Example 2. Assume we want to build 5 teams of 3 developers for a mashup that composes APIs from the *Mapping* and *Social* categories with 0.6 and 0.4 weights, respectively. Assume that APIs with IDs 1, 30, and 47 belong to *Mapping* and APIs with IDs 3, 17, and 22 relate to *Social.* The query is specified by $t = 5$, $m = 3$, and $A =$ $[<1, 0.6>, <30, 0.6>, <47, 0.6>, <3, 0.4>, <17, 0.4>, <22, 0.4>]$.

2.3 Crowdsourcing Team Generation (CTG)

CTG generates teams that best satisfy the mashup query requirements. It uses as input the sociometric graph SG as well as interests and reputation tables, U_I and $\hat{U_R}$. Before describing the CTG algorithm, we introduce the metrics to calculate the performance of a team based on SG, U_I, and $\hat{U_R}$.

We evaluate the *skills* of each user (i.e., developer) u_i in the community based on u_i's reputation and interest in each $API^j \in A$ specified in the mashup. The user's interest in API^j is multiplied by API^j_w to take into account the weight (i.e., importance) assigned by the administrator to each API:

$$User_{skills}(u_i) = \hat{U_R}(u_i) * \sum_{API^j \in A} U_I(u_i, API^j) * API^j_w \tag{1}$$

Based on the skills of each user u_i given in formula (1), we define the skills of a team T composed of m members as the sum of the skills of all members:

$$Team_{skills}(T) = \sum_{u_i \in T} User_{skills}(u_i) \tag{2}$$

Using the sociometric graph SG, we also introduce the *sociometric score* of T to quantify the level of collaboration between members. The sociometric score $Team_{sociometric}(T)$ of T accumulates the weights of all edges that connect members in T and divide it by the number m of team members:

$$Team_{sociometric}(T) = \frac{\sum_{ui \in T, uj \in T, (ui,uj) \in SG} W_e(u_i, u_j)}{m} \tag{3}$$

From formulas (1) and (2), we define the overall performance of T by summing the skills and sociometric of the team:

$$Team_{Performance}(T) = Team_{skills}(T) + Team_{sociometric}(T) \tag{4}$$

The CTG algorithm (Algorithm 1) identifies strongly connected members in the sociometric graph SG using the concept of *cliques* in graph theory. A *clique* C is a subset of vertices of an undirected graph such that every two distinct vertices in C are adjacent [5]. We use the *Bron Kerbosch* algorithm [5] to return cliques in the *AllCliques* list (line 3). Another important data structure is *SharedCliques* (lines 1 and 16). Each element SC in this list contains *common vertices* between cliques as well as the remaining vertices (called *potential vertices*) in the cliques. For example, Fig. 3b depicts two adjacent cliques C_1 =$\{u_1, u_3, u_4\}$ and C_2 =$\{u_2, u_3, u_4\}$. The common and potential vertices are defined by *SC.common*=$\{u_3, u_4\}$ and *SC.potential*=$\{u_1, u_2\}$, respectively. Due to space limitation, we omit the algorithm for the *GetSharedCliques()* function.

CTG uses *AllCliques* and *SharedCliques* to recommend the *top-t* (t is the number of required teams). Each element in the returned *TeamsList* is composed of the team's members and performance of the team as defined in formula (4). The algorithm first looks for cliques of size m (i.e., cliques with required number of members). If more teams still need to be generated ($TeamsList.size() < t$), then CTG explores the shared cliques.

Algorithm 1. Crowdsourcing Team Generation (CTG)

```
   Input  : $\hat{U_R}$ $Table, U_I$ $Table$, Sociometric Graph $SG$, Mashup Query $Q$
   Output: $TeamsList$ (recommended teams)
1  $SharedCliques \leftarrow null$;
2  $TeamsList \leftarrow null$;
3  $AllCliques \leftarrow BronKerboschAlgorithm(SG)$;
4  foreach $C \in AllCliques$ do
5      if $(C.size() == m)$ then
6          T = All users from C;
7          Calculate Performance$_{Team}$(T) of team T;
8          $TeamsList$.add$(T, Performance(T))$;
9          $AllCliques = AllCliques - C$;
10     end
11 end
12 if $( TeamsList.size() >= t)$ then
13     $TeamsList \leftarrow sort()$;    //By team performance
14     Return Top-t teams from $TeamsList$;
15 end
16 $SharedCliques = GetSharedCliques(AllCliques)$;
17 foreach $SC \in SharedCliques$ do
18     if $(SC.common.size() >= m)$ then
19         T = Top users from SC.common; // By skills or sociometric
20         Calculate Performance$_{Team}$(T) of team T;
21         $TeamsList$.add$(T, Performance(T))$;
22         $SharedCliques = SharedCliques - SC$;
23     end
24 end
25 if $( TeamsList.size() >= t)$ then
26     $TeamsList \leftarrow sort()$;    //By team performance
27     Return Top-t teams from $TeamsList$;
28 end
29 foreach $SC \in SharedCliques$ do
30     if $(SC.common.size() < m)$ then
31         T = Users from SC.common + remaining top from SC.potential;
           // By skills or sociometric
32         Calculate $Performance_{Team}$(T) of team T;
33         $TeamsList$.add$(T, Performance(T))$;
34         $SharedCliques = SharedCliques - SC$;
35     end
36 end
37 $TeamsList \leftarrow$ sort();    //By team performance
38 Return Top-t teams from $TeamsList$;
```

We identify the following three cases during team recommendation:

Case 1: Cliques have m members (lines 4–15) - CTG first parses cliques with the exact number of members. If the size of a clique C is m, then all members of C are used to form a team. We calculate the performance of T, insert T and its performance to $TeamsList$, and remove C from *AllCliques*. If $TeamsList$ reaches the desired number t of teams (lines 12–15), $TeamList$ is sorted based on performance and the top-t teams are returned, hence ending the algorithm. Otherwise, we process shared cliques (Case 2).

Case 2: Shared cliques have al least m members (lines 16–28) - CTG processes shared cliques that have enough members in their common vertices. It picks the top-m members from common vertices using one of two selection options (line 19). (i) *CTG by Skills*: m members with the highest skills are selected; and (ii) *CTG by sociometric*: m members with the highest sociometric scores are selected. The corresponding teams are inserted into $TeamsList$ as described in Case 1; the shared cliques used to build the teams are removed from $SharedCliques$. If $TeamsList$ reaches the desired number t of teams (lines 25–28), $TeamList$ is sorted based on performance and the top-t teams are returned, hence ending the algorithm. Otherwise, we proceed to Case 3.

Case 3: Shared cliques have less than m members (lines 29–38) - CTG handles the shared cliques that do not have enough members in their common vertices. It picks the remaining members from the potential vertices in the shared cliques. The remaining members are selected using *CTG by Skills* or *CTG by Sociometric* as described in Case 2 (line 31). Teams along with their calculated performance are added to $TeamsList$ and the top-t teams are returned.

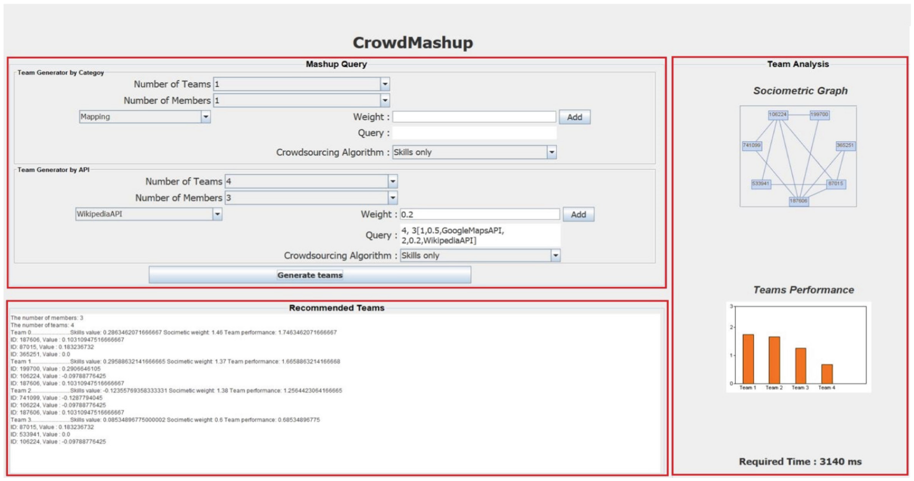
3 Implementation and Performance

In this section, we describe the *CrowdMashup* prototype implementation. Then, we evaluate the performance of our approach using real-world data and APIs.

3.1 CrowdMashup Prototype

We implemented a *CrowdMashup* prototype in Java. We used `Google BigQuery`[7] to retrieve comments about APIs from `StackOverflow`. We collected 8,617 comments related to 583 APIs. We used the `Jgrapgt` library[8] to handle graphs and identify cliques. We utilized `Stanford Natural Language Processing` library to calculate developers' attitude (interest) toward APIs. We used `Apache Spark's scalable machine learning` (MLlib) library[9] to deal with missing developers' interest values.

[7] https://cloud.google.com/bigquery/public-data/stackoverflow.
[8] http://jgrapht.org/.
[9] https://spark.apache.org/.



Number of Teams 1
Number of Members 1
Mapping
Weight :
Query :
Crowdsourcing Algorithm : Skills only

Number of Teams 4
Number of Members 3
WikipediaAPI
Weight : 0.2
Query : 4, 3[1,0.5,GoogleMapsAPI, 2,0.2,WikipediaAPI]
Crowdsourcing Algorithm : Skills only

```
The number of members: 3
The number of teams: 4
Team 0.......................Skills value: 0.2863462071666667 Socimetic weight: 1.46 Team performance: 1.7463462071666667
ID: 187606, Value : 0.10310947516666667
ID: 87015, Value : 0.183236732
ID: 365251, Value : 0.0
Team 1.......................Skills value: 0.29588632141666665 Socimetic weight: 1.37 Team performance: 1.6658863214166668
ID: 199700, Value : 0.2906646105
ID: 106224, Value : -0.09788776425
ID: 187606, Value : 0.10310947516666667
Team 2.......................Skills value: -0.12355769358333331 Socimetic weight: 1.38 Team performance: 1.2564423064166665
ID: 741099, Value : -0.1287794045
ID: 106224, Value : -0.09788776425
ID: 187606, Value : 0.10310947516666667
Team 3.......................Skills value: 0.08534896775000002 Socimetic weight: 0.6 Team performance: 0.68534896775
ID: 87015, Value : 0.183236732
ID: 533941, Value : 0.0
ID: 106224, Value : -0.09788776425
```

Required Time : 3140 ms

Fig. 4. The *CrowdMashup* user interface

Figure 4 shows *CrowdMashup*'s graphical interface. Mashup administrators specify their queries through the *Mashup Query* pane (top left). They assign the number of required teams and members in each team. Administrators enter either a list of specific APIs or generic API categories along with their weights. They also pick the algorithm to be used to generate teams: (1) *Skills Only*: members are selected based on skills only. (2) *Sociometric Only*: members are selected based on sociometric only. (3) *CTG-Skills*: uses both skills and sociometric but gives priority to skills in dealing with shared cliques (lines 19 and 31 in Algorithm 1). (4) *CTG-Sociometric*: uses both skills and sociometric but gives priority to sociometric in dealing with shared cliques (lines 19 and 31 in Algorithm 1). The generated teams are shown in the *Recommended Teams* pane (bottom left). The pane shows each recommended team as a list of developer IDs. It also displays the calculated performance of each team and orders the generated teams based on their performance. The *Team Analysis* pane (right) displays the two metrics for team recommendation: sociometric sub-graph and team performances illustrated in a bar graph to visualize the performance of different teams. The time to generate teams is also shown in this pane.

3.2 Experiments

The aim of the experiments is to assess the ability of CTG to select teams with the best performance. We ran our experiments on a 64-bit Windows 10 environment, in a machine equipped with an Intel i7-7700HQ and 16 GB RAM. We measured the performance of the generated teams using three non-CTG algorithms: Random (members are randomly selected), Skills Only, Sociometric Only; and two CTG algorithms: CTG-Skills and CTG-Sociometric. We ran all experiments on real-world data and APIs from `StackOverflow` and `programmableWeb`.

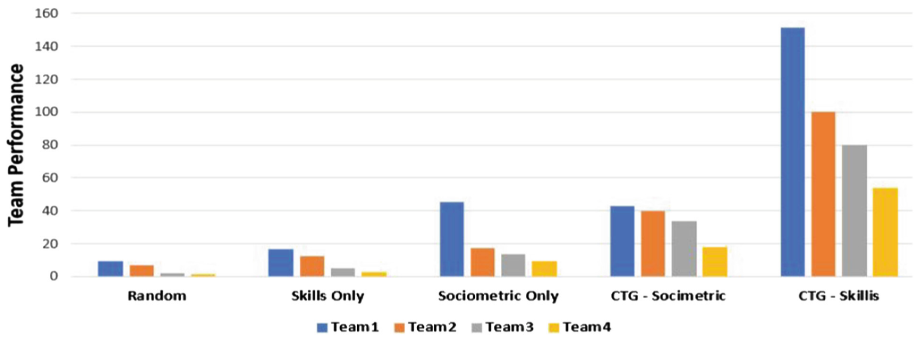


Fig. 5. Single query team performance for non-CTG (random, skills only, sociometric only) and CTG (CTG-skills, CTG-sociometric) Algorithms

Figure 5 compares the five algorithms using the same mashup query to generate four teams with seven members per team. First, we compare CTG vs. non-CTG algorithm in terms of team performance. CTG algorithms perform better than non-CTG algorithms due to combining sociometric and skills. Besides, CTG-Skills generates better teams than CTG-Sociometric. This is because vertices that are outside cliques are unlikely to return high sociometric values. Then, we compare the distribution of the performance of the four teams recommended by each algorithm. Figure 5 shows that team performance decreases steadily from the first to the last team in both CTG algorithms. Hence, CTG shows more balanced teams than non-CTG algorithms. For instance, there is significant difference (more than double) between the performance of the first and second teams in the Sociometric-Only algorithm.

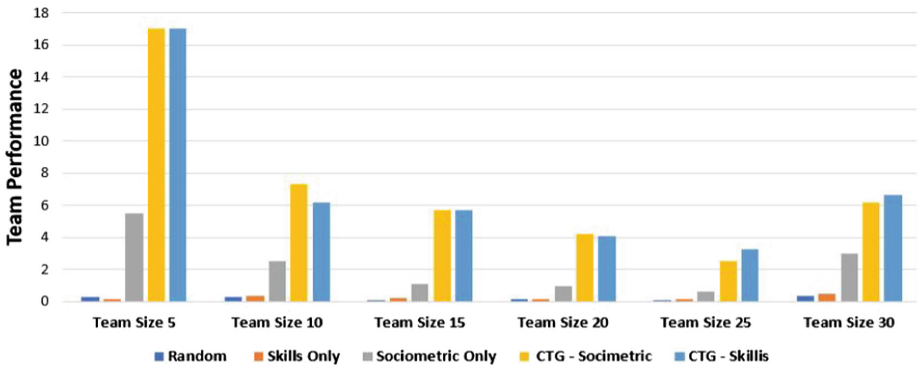


Fig. 6. Multiple queries team performance for different team sizes

We also conduct experiments to explore how forming teams with various sizes is handled by CTG. We randomly generated queries with sizes 5, 10, 15, 20, 25, and 30. We had 5 queries for each time size (for total of 30). As shown in Fig. 6, CTG algorithms always show better team performances than the non-CTG algorithms regardless of the team size. This is because non-CTG algorithms ignore sociometric, skills, or both (in the case of random). Overall, generating teams with bigger sizes (more than 10 members) leads to lower performance, as it is harder to find a large number of developers with the right skills and social relationships. Studies have shown that 3–7 developer teams are key to successful software projects (3–5 person teams would be the best)[10]. Hence, this makes CTG a suitable technique for team recommendation. For large team sizes (e.g., 25), CTG-Skills shows better team performance than CTG-Sociometric as finding cliques or shared cliques with larger sizes becomes challenging. For teams of size 2–5, CTG-Skills and CTG-Sociometric are comparable, and they largely outperform the three other algorithms: Random, Skills Only, and Sociometric Only. In teams of size 6–10, CTG-Sociometric shows better team than CTG-Skills as finding cliques or shared cliques with size 10 is still possible and improves the overall team performance.

4 Related Work

The growth and popularity of crowdsourcing has led to significant research on forming teams to facilitate collaborative software development [7]. Part of this research has focused on team structure, while other contributions focused on the complexity of the algorithm and economic factors for team building. [9] shows that network structure between members has a vital effect on team formation. It uses four different network structures to model team formation and compares the performance of each structure. [6] takes advantage of social network information and uses hierarchical structures (e.g., using "report to") between team members. [16] defines a self-organized team formation technique by allowing members to rate each other and use other information such as demographics (e.g., age, gender). [8] proposes a framework that recommends teams based on the skills and connection among members. It uses co-authorship in DBLP and clustering algorithms to find expert teams (sub-graph). [22] employs a dynamic programming technique in crowdsourcing based on the prior familiarity of members to generate target teams. It considers the availability (response time) of the members to find most familiar alternative members. [26] defines heuristic algorithms based on notions such as weak and strong ties in social networks. It utilizes two metrics to find social connection from an undirected weighted graph.

Several techniques dealt with the issue of improving the efficiency of the team formation process. [28] proposes a genetic algorithm with the goal of finding the best groups that can meet the defined tasks based on members availability, skills, and price. [11] introduces an approach for forming teams with specific skills from a vast professional community using network communication costs to optimize team formation. It calculates communication costs by using minimum spanning tree and the largest shortest path from the graph. [2] describes a greedy approach

[10] http://www.qsm.com/process_improvement_01.html.

for better performance considering team size and workload such as the number of tasks allocated to each member.

[20] and [14] propose a team formation technique based on pricing to find cost effective teams. [12] studies task coordination cost in crowdsourcing teams. It aims to facilitate self-coordination and communication among teams by distributing and synchronizing the project tasks. [10] introduces a technique for forming multiple teams to maximize the global efficiency of the teams considering skills, availability, sociometric (relationship), and allowed time (part-full time) members. [25] proposes a negotiation-based team formation technique where the deal to join the team is used as a formation factor. [15] investigates how personality affects team performance by applying the DISC (dominance, inducement, submission, compliance) personality test. [23] discusses team elasticity in software development such as the skills, experiences, response time and reliability of the workers. [1] proposes a data leak-aware system in crowdsourcing team by applying clustering algorithms that detect social interactions between members to avoid data leakage. [19] conducts a statistical analysis to investigate how to extract influence factors from successful teams.

CrowdMashup differs from existing approaches in multiples ways. First, to the best of our knowledge, this paper is the first to look at team recommendation for mashups. Second, we define a two-level approach to analyze developer communities. At the individual developer's level, we infer developer's interests in APIs through natural language processing and collaborative filtering. At the community level, we consider social relationships among developers as an important factor to recommend team members. We model interactions among developers as a weighted undirected graph and find cliques to identify strongly related developers. Note that our approach is different from the one introduced in [26] where members of the same team are selected from different cliques to ensure the impartiality of the execution result of a task. We use cliques to recommend teams composed of (socially) strongly connected members to improve productivity.

5 Conclusion

We propose the *CrowdMashup* approach to recommend teams for mashup development. The first *CrowdMashup* phase analyzes the `StackOverflow` developer community to infer developers' skills in using APIs. It also models the ability of developers to collaborate with each other via a sociometric graph. The second phase recommends crowdsourcing teams that best satisfy the requirements of a mashup query. We introduce a team recommendation algorithm that combines developers' skills and sociometric. We provide a prototype implementation and conduct experiments on real-world data and APIs from `StackOverflow` and `programmableWeb` to evaluate our approach. Experiments show promising results in generating efficient and balanced teams for mashup development.

References

1. Amor, I.B., Benbernou, S., Ouziri, M., Malik, Z., Medjahed, B.: Discovering best teams for data leak-aware crowdsourcing in social networks. ACM Trans. Web (TWEB) **10**(1), 2 (2016)
2. Anagnostopoulos, A., Becchetti, L., Castillo, C., Gionis, A., Leonardi, S.: Power in unity: forming teams in large-scale community systems. In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, pp. 599–608 (2010)
3. Anagnostopoulos, A., Becchetti, L., Castillo, C., Gionis, A., Leonardi, S.: Online team formation in social networks. In: Proceedings of the 21st International Conference on World Wide Web, pp. 839–848 (2012)
4. Bouguettaya, A., et al.: A service computing manifesto: the next 10 years. Commun. ACM **60**(4), 64–72 (2017)
5. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. Commun. ACM **16**(9), 575–577 (1973)
6. Ding, C., Xia, F., Gopalakrishnan, G., Qian, W., Zhou, A.: Teamgen: an interactive team formation system based on professional social network. In: Proceedings of the 26th International Conference on World Wide Web Companion, pp. 195–199 (2017)
7. Doan, A.H., Ramakrishnan, R., Halevy, A.Y.: Crowdsourcing systems on the world-wide web. Commun. ACM **54**(4), 86–96 (2011)
8. Farhadi, F., Hoseini, E., Hashemi, S., Hamzeh, A.: Teamfinder: a co-clustering based framework for finding an effective team of experts in social networks. In: 12th IEEE International Conference on Data Mining Workshops, ICDM Workshops, Brussels, Belgium, 10 December, pp. 107–114 (2012)
9. Gaston, M., Simmons, J., DesJardins, M.: Adapting network structure for efficient team formation. In: Proceedings of the AAAI 2004 Fall Symposium On Artificial Multi-agent Learning (2004)
10. Gutiérrez, J.H., Astudillo, C.A., Ballesteros-Pérez, P., Mora-Melià, D., Candia-Véjar, A.: The multiple team formation problem using sociometry. Comput. Oper. Res. **75**, 150–162 (2016)
11. Lappas, T., Liu, K., Terzi, E.: Finding a team of experts in social networks. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, 28 June – 1 July, pp. 467–476 (2009)
12. Lee, S.W., Chen, Y., Klugman, N., Gouravajhala, S.R., Chen, A., Lasecki, W.S.: Exploring coordination models for ad hoc programming teams. In: Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems, pp. 2738–2745 (2017)
13. Lemos, A.L., Daniel, F., Benatallah, B.: Web service composition: a survey of techniques and tools. ACM Comput. Surv. **48**(3), 1–41 (2016)
14. Liu, Q., Luo, T., Tang, R., Bressan, S.: An efficient and truthful pricing mechanism for team formation in crowdsourcing markets. In: 2015 IEEE International Conference on Communications (ICC), pp. 567–572 (2015)
15. Lykourentzou, I., Antoniou, A., Naudet, Y., Dow, S.P.: Personality matters: balancing for personality types leads to better outcomes for crowd teams. In: Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work and Social Computing, pp. 260–273 (2016)
16. Lykourentzou, I., Wang, S., Kraut, R.E., Dow, S.P.: Team dating: a self-organized team formation strategy for collaborative crowdsourcing. In: Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems, pp. 1243–1249 (2016)

17. Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D.: The stanford coreNLP natural language processing toolkit. In: Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55–60 (2014)
18. Nasehi, S.M., Sillito, J., Maurer, F., Burns, C.: What makes a good code example? A study of programming q&a in stackoverflow. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 25–34 (2012)
19. Pobiedina, N., Neidhardt, J., Calatrava Moreno, M.D.C., Werthner, H.: Ranking factors of team success. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 1185–1194 (2013)
20. Rokicki, M., Zerr, S., Siersdorfer, S.: Groupsourcing: team competition designs for crowdsourcing. In: Proceedings of the 24th International Conference on World Wide Web, pp. 906–915 (2015)
21. Ryza, S., Laserson, U., Owen, S., Wills, J.: Advanced Analytics with Spark: Patterns for Learning from Data at Scale (2017)
22. Salehi, N., McCabe, A., Valentine, M., Bernstein, M.: Huddler: convening stable and familiar crowd teams despite unpredictable availability. arXiv preprint arXiv:1610.08216 (2016)
23. Saremi, R.L., Yang, Y., Ruhe, G., Messinger, D.: Leveraging crowdsourcing for team elasticity: an empirical evaluation at topcoder. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 103–112 (2017)
24. Stol, K.-J., Fitzgerald, B.: Researching crowdsourcing software development: perspectives and concerns. In: Proceedings of the 1st International Workshop on CrowdSourcing in Software Engineering, pp. 7–10 (2014)
25. Wang, W., Jiang, J., An, B., Jiang, Y., Chen, B.: Toward efficient team formation for crowdsourcing in noncooperative social networks. IEEE Trans. Cybern. **47**(12), 4208–4222 (2017)
26. Yin, X., et al.: Social connection aware team formation for participatory tasks. IEEE Access **6**, 20309–20319 (2018)
27. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding mashup development. IEEE Internet Comput. **12**(5), 44–52 (2008)
28. Yue, T., Ali, S., Wang, S.: An evolutionary and automated virtual team making approach for crowdsourcing platforms. In: Li, W., Huhns, M.N., Tsai, W.-T., Wu, W. (eds.) Crowdsourcing. PI, pp. 113–130. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47011-4_7