An Evolutionary and Automated Virtual Team Making Approach for Crowdsourcing Platforms

Tao Yue, Shaukat Ali and Shuai Wang

Abstract Crowdsourcing has demonstrated its capability of supporting various software development activities including development and testing as it can be seen by several successful crowdsourcing platforms such as TopCoder and uTest. However, to crowd source large-scale and complex software development and testing tasks, there are several optimization challenges to be addressed such as division of tasks, searching a set of registrants, and assignment of tasks to registrants.Since in crowdsourcing a task can be assigned to registrants geographically distributed with various backgrounds, the quality of final task deliverables is a key issue. As the first step to improve the quality, we propose a systematic and automated approach to optimize the assignment of registrants in a crowdsourcing platform to a crowdsourcing task. The objective is to find the best fit of a group of registrants to the defined task. A few examples of factors forming the optimization problem include budget defined by the task submitter and pay expectation from a registrant, skills required by a task, skills of a registrant, task delivering deadline, and availability of a registrant. We first collected a set of commonly seen factors that have impact on the perfect matching between tasks submitted and a virtual team that consists of a selected set of registrants. We then formulated the optimization objective as a fitness function‖the heuristics used by search algorithms (e.g., Genetic Algorithms) to find an optimal solution. We empirically evaluated a set of well-known search algorithms in software engineering, along with the proposed fitness function, to identify the best solution for our optimization problem. Results of our experiments are very positive in terms of solving optimization problems in a crowdsourcing context.

T. Yue (✉) · S. Ali · S. Wang
Certus Software V&V Center, Simula Research Laboratory, Oslo, Norway
e-mail: tao@simula.no

S. Ali
e-mail: shaukat@simula.no

S. Wang
e-mail: shuai@simula.no

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_7

1 Introduction

Crowdsourcing software engineering is gaining more and more attention these days as increasing number of companies start looking for an innovative way to develop software and conducting other software engineering activities such as testing. The main reason is that the cost can be significantly reduced. Moreover some crowdsourcing platforms such as Topcoder[1] and UTest[2] have shown their success and a large number of registrants of these platforms form a large virtual pool for performing tasks virtually. However, to compare with traditional software development practices, crowdsourcing software engineering is still at its early stage, which leaves a lot of space for research. Especially large-scale software engineering on crowdsourcing platforms are facing a lot of challenges, one of which is how to decompose, schedule and integrate tasks such that the overall quality and productivity can be maintained as they are developed in a traditional software development environment.

Towards supporting large-scale, crowdsourcing software engineering, in this chapter, we propose a search-based approach, along with a series of experiments to demonstrate how search-based software engineering can be applied to address optimization problems in crowdsourcing software engineering. In this chapter, we particularly focus on assisting platform managers to form a virtual team on a crowdsourcing platform for a submitted task such that the overall quality and productivity of performing this task can be ensured to a certain extent. This is an optimization problem as there are some constraints to find such a virtual team. For example, the cost to hiring the team members of the virtual team should be within the budget, the task should be completed within certain duration, and the task should match the background of the virtual team members.

The core of Search Based Software Engineering (SBSE) are search algorithms (e.g., Genetic Algorithms mimicking natural selection process) that can efficiently find optimal solutions to the problems that have large complex search spaces. Typical examples of such problems in software engineering include: optimal allocation of requirements, optimal architecture design, test case generation, and test optimization. According to the comprehensive review of Harman et al. [11], SBSE has been extensively investigated to address various software engineering problems spanning from requirements, testing to reengineering of a typical software development lifecycle. Particularly for requirements, SBSE has been applied for various optimization problems such as requirements selection [6], prioritization [7], and assignment [10, 14], with different objectives such as maximizing customers'/stakeholders' satisfaction, maximizing benefits/value and minimizing cost. For testing SBSE has been applied to successfully address test generation and test optimization problems [1, 11]. To the best of our knowledge, SBSE has never been applied to address optimization issues

[1] http://www.topcoder.com/.

[2] http://www.utest.com/.

existing in crowdsourcing. In this paper, we mainly present a pilot study we recently conducted to demonstrate that SBSE can also be applied for addressing optimization issues in crowdsourcing.

The rest of paper is organized as follows. Section 2 provides the overview of our approach. Section 3 presents a conceptual model that formalizes key elements of our approach. In Sect. 4, our search-based crowdsourcing methodology and results of the experiment we conducted to evaluate the fit-ness function, the key element of our search-based crowdsourcing methodology. Section 5 discusses the threats to validity of our experiments and we conclude the paper and discuss the future work in Sect. 6.

2 Overview

In this section, we provide an overview of the approach we propose in this paper and its extensions for future, as shown in Fig. 1. We classify stakeholders that are relevant to a crowdsourcing platform into four groups: Task Submitter, Crowd, Platform Manager and Virtual Team. A task submitter is a person who submits a task through the crowdsourcing platform and looks for a virtual team (from the crowd) to complete the task. A platform manager (employee of the crowdsourcing platform) is assigned to manage the task (including assisting the formation of the virtual team and negotiation between the task submitter and the crowd) on the behalf of the crowdsourcing platform. The virtual team is formed by selecting a group of registrants of the crowdsourcing platform who expressed their willing to complete the task via the crowdsourcing platform.

Our objective is to propose a solution, integrated as part of the services provided by the crowdsourcing platform, to assist the platform manage to form a virtual team according to the requirements from the task submitter (provided as part of the task description) at the same time satisfying the expectation of the virtual team members. In other words, such a solution aims to find a match between the task submitter side and the virtual team members. Doing so, we believe, will indirectly improve the quality and productivity of software development activities via a crowdsourcing platform.

In Fig. 1, we highlight the key features, properties and technologies to apply of our solution. It is important to notice that such a solution is Generic in the sense that it is not specific to a particular type of task that a crowdsourcing platform can provide such as testing. Therefore, the solution can be widely applied in any crowdsourcing platform, as far as we can see.

The first key component of our solution is to provide a set of specification methodologies for task submitters to specify tasks (including budget, duration, task content, requirements) and for an individual to define her/his profile (e.g., experience, skills). Such specification methodologies ideally should be easy to use for end users (in our context, task submitters, crowd and platform manager). Submitted tasks and crowd profiles specified using these methodologies could then be automatically collected,

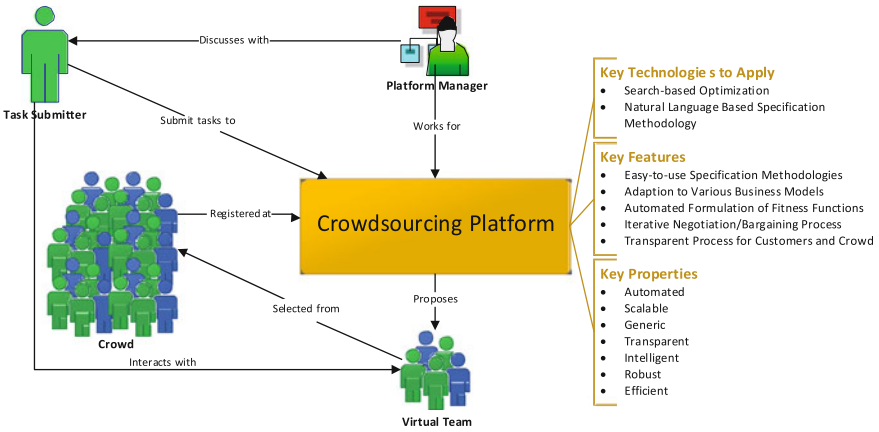

Fig. 1 Overview

analyzed to serve other components of the platform such as the automated formation of fitness functions.

After all these information are collected by the crowdsourcing platform, our solution is then ready to take a task submitted by a task submitter and automatically propose a solution, which is a virtual team selected from the crowd. The platform manager can then coordinate the virtual team and the task submitter to complete the task. We rely on search-based optimization methodologies to automatically identify a virtual team that is optimal in the sense that in the scope of the crowd, the profiles of the team members of the virtual team fit the requirements of the task best. It is worth noting that search algorithms work together with carefully designed fitness functions, which are used to guide search algorithms towards the direction of finding an optimal solution. Such methodologies are *Automated*, *Scalable* and *Efficient*. Details of the search-based methodologies will be provided in Sect. 4.1. In case that all the required information to derive a fitness function for search algorithm can be extracted automatically from the task specifications and crowd profile specifications, the derivation of a fitness function can be *Automated*.

It is important to notice that we are not aiming to replace platform managers. Instead, we aim to assist platform managers to better conduct their jobs. For example, the process of identifying a virtual team from the crowd to perform a task, if it is required to be manually done by a platform manager, she/he has to go through the registrants' profiles, their bids and try to, mostly based on their experience, to form a virtual team that can complete the task submitted and satisfies the constraints such as budget, time schedule and required expertise. One can instantly understand that if the task is complex enough to require a virtual team with more than 10 members, different expertise, and tight schedule, which is often the case for supporting large-scale crowdsourcing software development, manually forming such a virtual team satisfying all these constraints is simply unmanageable. Therefore, this inspires us to pro-pose an automated, scalable, intelligent, robust and efficient solution to assist

platform managers. In addition, our solution can be easily customized for catering needs of various crowdsourcing platforms executing different business models.

Ideally a crowdsourcing platform should be able to provide an effective mechanism to support the negotiation between a task submitter and the crowd in terms of price, schedule, etc. The common practice is that a platform manager plays the role to coordinate the negotiation or bargaining process without any intelligent support from the platform, which might lead to low productivity and therefore less customer satisfaction. We however propose a simulation-based, intelligent negotiation/bargaining process. Based on our search-based methodologies, we can instantly find a solution that satisfies a set of constraints that are defined based on the results of a round of negotiation. A new negotiation implies updating this set of constraints and therefore triggers the execution of our search-based optimization methodologies to find another solution satisfying the updated set of constraints.

Depending on the business model adopted by a crowdsourcing platform, it might be useful to make the formation of virtual teams, the negotiation process transparent to task submitters and the crowd as well. Doing so might somehow lead to a healthier (virtual) working environment on the crowdsourcing platform and therefore indirectly contributing to a higher quality and productivity of the development process via the crowdsourcing platform.

3 Conceptual Model

In this section, we formally specify concepts that are related to the method-ologies we propose as a conceptual model in UML class diagram (Fig. 2). Each concept is presented as a class and the relationships among concepts are captured as UML associations or generalizations. From the figure, one can notice that we can classify stakeholders into three types: `Registrant`, `Submitter`, and `Platform-ProjectManager`.

`Registrant` registered her/himself on the crowdsourcing platform and specified her/his profile accordingly. Such a `Profile` should contain a list of information that is required to evaluate an individual based on his/her programming language skills, natural language skills and rank at the platform, experience, etc. It is important to notice that in our conceptual model, we capture the classifications of `expertise`, `programmingLanguageSkill` and `naturalLanguage-Skill` of a registrant's profile as two enumerations: `ExpertiseType`, `ProgrammingLanguageType` and `NaturalLanguageType`. Such enumerations/classifications can be easily extended for different purposes. `rankAtPlatform` is an attribute of `Profile` that represents a rank of a registrant maintained by the platform. A platform provides a mechanism to rank a registrant according to her/his performance, which is usually evaluated by task submitters, and other registrants who worked with this particular registrant. `Experience` of a registrant is calculated based on the types of tasks that the registrant has completed via the platform. `averagePaymentPer-Task` is derived from the information

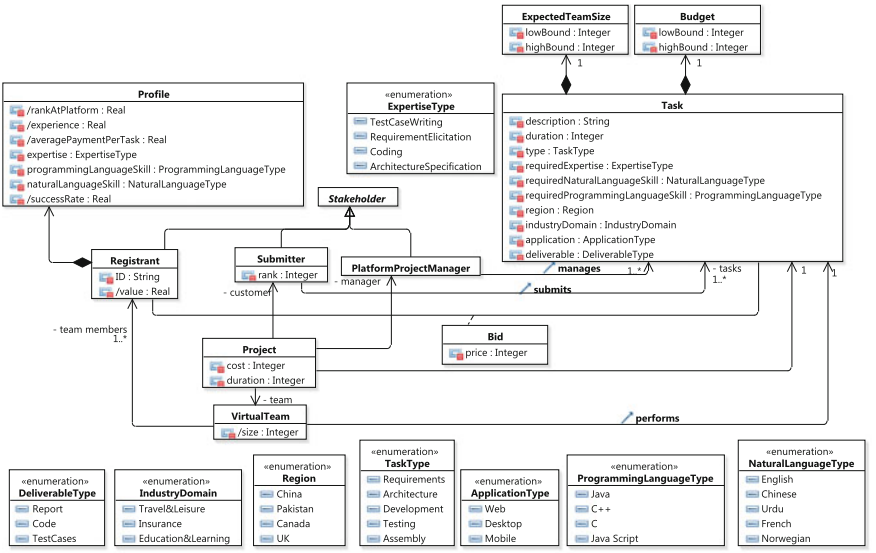

Fig. 2 General conceptual model

maintained by the platform as it has records all the payments that have been done for the registrant. `successRate` is another factor that should be accounted for when evaluating a registrant. It represents the rate of the successful task delivered via the platform.

Another important concept is `Task`, which captures the task `Description`, `Duration`, etc. We defined an enumeration `TaskType` to classify different types of tasks including `Requirements`, `Architecture`, `Development`, `Testing` and `Assembly`, which can be easily extended when needed. When a task submitter submits a task, as part of the specification of the task, she/he has to also define `requiredExpertise`, `requiredNaturalLanguageSkill`, `requiredProgrammingLanguageSkill`, which are classified using the enumerations (`ExpertiseType`, `ProgrammingLanguageType` and `NaturalLanguageType`) also referenced by three attributes of a registrant's profile. Therefore, it is easy to understand that ideally an optimal solution should match required expertise and skills of a task and profiles of the virtual team members. Besides these six attributes of class Task, we also capture `Region`, `IndustryDomain`, `ApplicationType` and `DeliverableType`, which are useful information needed to form a virtual team to finish a task. Two other important pieces of information that are associated to a task are `ExpectedTeamSize` and `Budget`, which are provided by a task submitter while submitting a task, which are constraints in terms of forming a virtual team. The size of a formed virtual team should be within the range of the expected team size of a submitted task and the budget should be sufficient to pay the virtual team members.

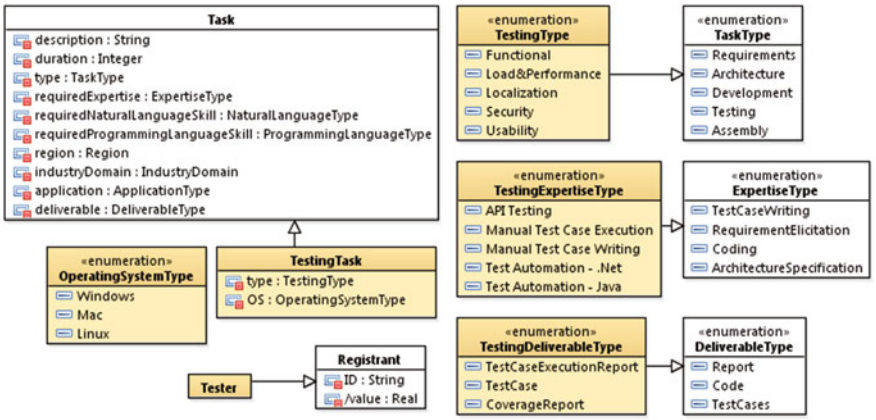

Fig. 3 Conceptual model for testing

After a virtual team is formed, a project is then created in the platform. Such a project should define the real cost to complete a task and time required to finish the task. Such information is used by a fitness function used by search-based algorithms to find an optimal solution. `Bid` plays an important role in most of existing crowdsourcing platforms such as TopCoder by linking a registrant to a specific task. Bids submitted by registrants for a task are used to check whether the cost to form a virtual team (i.e., the sum of the bids submitted by the virtual team members) is within the budget range of the task submitter.

Note that the conceptual model presented in Fig. 2 is generic and therefore not targeting to any specific type of tasks. However, since it is generic, we show an example (Fig. 2) how it can be extended for conducting a specific task. In the context of Fig. 2, our general conceptual model is extended for capturing concepts for conducting testing tasks. For example, crowdsourcing platform UTest is a platform focusing on testing tasks only. From Fig. 3, one can notice that for testing, four new enumerations are defined to extend four enumerations defined in Fig. 2. By doing so, we can extend the generic platform into a specialized one by introducing more specialized information such as `TestingType`. Besides introducing additional enumerations, we also define two next concepts (i.e., `TestingTask` and `Tester`) to extend generic concepts Task and Registrant.

4 Search-Based Crowdsourcing Methodologies

According to the comprehensive review of Harman et al. [11], Search Based Software Engineering (SBSE) has been extensively investigated to address various software engineering problems spanning from requirements, testing to reengineering of a typical software development lifecycle. Particularly for requirements, SBSE has been applied for various optimization problems such as requirements selection

[6], prioritization [7], and assignment [10, 14], with different objectives such as maximizing customers'/stakeholders' satisfaction, maximizing benefits/value and minimizing cost. For testing SBSE has been applied to successfully address test generation and test optimization problems [1, 11]. To the best of our knowledge, SBSE has never been applied to address optimization issues existing in crowdsourcing. In this section, we mainly present a pilot study we recently conducted to demonstrate that SBSE can also be applied for addressing optimization issues in crowdsourcing and it is promising in a future to integrate such as an optimization methodology as part of crowdsourcing platforms.

The main challenge in proposing a SBSE solution is to propose and assess a fitness function for the intended optimization problem. In the rest of the section, we propose the fitness function for our crowdsourcing problem and evaluate the fitness function in conjunction with the following search algorithms, i.e., Genetic Algorithms (GAs), (1+1) Evolutionary Algorithm (EA), Alternating Variable Method (AVM). Random Search (RS) was used as the baseline to evaluate the performance of these algorithms.

4.1 Description on Selected Search Algorithms

The most common search algorithms that have been employed for SBSE are evolutionary algorithms, simulated annealing, hill climbing (HC), ant colony optimization, and particle swarm optimization [8]. Among these algorithms, HC is a simpler, local search algorithm. The SBSE techniques using more complex, global search algorithms are often compared with the techniques based on HC and random search to determine whether their complexity is warranted to address a specific SBSE problem. The use of the more complex search algorithm may only be justified if it performs significantly better than, for instance, random search.

To use a search algorithm, a fitness function needs to be defined. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. The fitness function will be used to guide the search toward fitter solutions.

Below, we provide a brief description of the search algorithms that we used in the pilot study and we will investigate more algorithms in the future. AVM was selected as a representative of local search algorithms. GA was selected since it is the most commonly used global search algorithm in SBSE [1]. (1+1) EA is simpler than GAs, but in previous software testing work we found that it can be more effective in some cases (e.g., see [3]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [1].

4.1.1 Genetic Algorithms

Genetic Algorithms (GAs) are the most well-known [1] and are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through crossover and mutation operators. GAs are the most commonly used algorithms and hence we do not provide further details; however an interested reader may consult the following reference for more details [13].

4.1.2 (1+1) Evolutionary Algorithm

(1+1) Evolutionary Algorithm (EA) [9] is simpler than GAs. In (1+1) EA, popu-lation size is one, i.e., we have only one individual in the population and the individual is represented as a bit string. As opposed to GAs, we do not use the crossover operator but only rely on a bitwise mutation operator for exploring the search space. To produce an offspring, this operator independently flips each bit in the bit string with a probability (p) based on the length of the string. If the fitness of the child is better than that of the parent (bit string of the child before mutation), the child is retained for the next generation.

4.1.3 Alternating Variable Method

Alternating Variable Method (AVM) is a local search algorithm first introduced by Korel [12]. The algorithm works in the following way: Suppose we have a set of variables $\{v_1, v_2, v_n\}$, we then try to maximize fitness of v_1, while keeping the rest of the variables constant, which are generated randomly. The search is stopped if a solution is found; otherwise, if the solution is not found, but we found a minimum with respect to v_1, we switch to the second variable. Now, we fix v_1 at the found minimum value and try to minimize v_2, while keeping the rest of the variables constant. The search continues in this way, until we find a solution or we have explored all the variables.

4.2 Problem Representation and Fitness Function

Our goal is to form a virtual team to complete a task within the budget specified for a task by a submitter by accounting for an optimal matching between required expertise, skills, and other relevant information with the team members' experience, expertise, skills and bids. The ultimate objective is to form a virtual team that should work like in a real world in the sense that the real values of the virtual team should be appreciated (reflected as payment received for the task) and deliver high quality deliverables in a productive way. We believe, by doing so, the overall quality and

productivity of software development via crowdsourcing platforms would be improved. Moreover, we also expect this philosophy of forming a virtual team would be very useful especially in the context of practicing large-scale software development via crowdsourcing, as conducting large-scale software development tasks is not anymore one person task. Teamwork in crowd should be taken into account for managing and conducting this kind of tasks. The problem is more complicated if we account for scheduling and dependencies among sub-tasks that are required to be completed by more than one persons. To this end, a scalable, systematic task scheduling, virtual team formation is a very important issue to tackle. In this paper, we make a first step towards this direction.

Suppose we have a Crowd C with a set of m registrants $C = \{r_1, r_2, \ldots, r_m\}$. A task submitter submits a task by defining the budget range: $Budget_{min}$ and $Budget_{max}$ and team size range n: $TeamSize_{min}$ and $TeamSize_{max}$.

$$TeamSize_{min} \leq n \leq TeamSize_{max} \tag{1}$$

A solution would be a virtual team of registrants who bided for the task: $V = \{r_1, r_2, r_n\}$, where $n \leq m$. Registrant i has a property defining his/her value $Value_i$, which is calculated based on the four factors: $SuccessfulRating(0-1)$, $CustomerRating(0-1)$, $Experience(0-1)$, and $PaymentHistory$ (average payment/task in the past).

$$Value_i = \frac{SuccessfulRating + CustomerRating + Experience + PaymentHistory}{4} \tag{2}$$

Notice that all these four values are normalized between 0 and 1. In the above formula, we take average of all these four values and the resultant $value_i$ will be again between 0 and 1.

Each registrant i provides $RBid_i$ to complete the task. To form an optimal virtual team, the solution must satisfy the following requirements: (1) Budget and team size requirements; (2) A solution must provide a bid values for all registrants as much as closer to their requested bids ($f_{bidGap}(n)$); (3) Each registrant in a virtual team must be assigned a bid which is fair according to his/her experience, ratings, and payment history ($f_{similarity}(n)$).

For optimization problem, our optimization parameter is Bid_i corresponding to registrant i. For the first requirement, for each registrant in a virtual team, a search algorithm finds a bid value for each registrant ($RBid_i \leq Bid_i \leq RBid_i$) such that:

$$Budget_{min} \leq \Sigma_{i=1}^{n} Bid_i \leq Budget_{max} \tag{3}$$

For the second requirement, we calculate $f_{bidGap}(n)$, whose formula is shown below, where we try to make the Bid_i as close as possible to $RBid_i$ for each registrant.

$$f_{bidGap}(n) = \frac{\Sigma_{i=1}^{n} nor(|RBid_i - Bid_i|)}{n} \tag{4}$$

In the formula below, $nor()$ is a normalization function, which is calculated as $nor(x) = x/(x+1)$. We adopted this normalization function from the literature and has proven to be more robust than other normalization functions in the context of search-based software engineering [2, 3].

For the third requirement, we calculate $f_{similarity}(n)$, which is calculated by the following formula:

$$f_{similarity}(n) = \frac{\Sigma_{i=1}^{n} nor(|Value_i * Budget_{max} - Bid_i|)}{n} \tag{5}$$

Based on the above requirements, our fitness function can be formulated as below:

$$f_{Fitness}(n) = \frac{(f_{similarity}(n) + xor(f_{bmax}(n), f_{bmin}(n)) + f_{bidGap}(n))}{3} \tag{6}$$

where f_{bmax} and f_{bmin} are defined as follows:

$$f_{bmax}(n) = \begin{cases} 0, & \Sigma_{i=1}^{n} Bid_i - Budget_{max} \leq 0 \\ nor(\Sigma_{i=1}^{n} Bid_i - Budget_{max}), & \Sigma_{i=1}^{n} Bid_i - Budget_{max} > 0 \end{cases} \tag{7}$$

$$f_{bmin}(n) = \begin{cases} 0, & Budget_{max} - \Sigma_{i=1}^{n} Bid_i \leq 0 \\ nor(Budget_{max} - \Sigma_{i=1}^{n} Bid_i), & Budget_{max} - \Sigma_{i=1}^{n} Bid_i > 0 \end{cases} \tag{8}$$

4.3 Empirical Evaluation

This section discusses the experiment design, execution, and analysis of the evaluation of the fitness function with the four search algorithms for addressing our optimization problem.

4.3.1 Experiment Design

The objective of our experiments is to evaluate proposed fitness function in conjunction with the selected search algorithms in terms of solving our optimization problem: Find a virtual team (v) of size n from crowd C of size m registrants, such that v meets all budget and team size requirements of a project, each registrant must obtain a bid that is closer to what was requested, and each registrant must obtain a bid value that matches his/her ratings, experience, and payment history.

4.3.2 Research Questions

In these experiments, we address the following research question:

RQ1: Are the search algorithms effective to solve our optimization problem, to compare with RS?

RQ2: Among AVM, (1+1) EA and GA, which one fares best in solving our optimization problem?

4.3.3 Selection Criteria of Search Algorithms and Parameter Settings

In our experiments, we compared four search algorithms: AVM, GA, (1+1) EA, and RS (Sect. 4.1). AVM was selected as a representative of local search algorithms. GA was selected since it is the most commonly used global search algorithm in search-based software engineering [1]. We selected steady state GA with a population size of 100 and a crossover rate of 0.75, with a 1.5 bias for rank selection. We used a standard one-point crossover, and mutation of a variable is done with the standard probability 1/n, where n is the number of variables. Different settings would lead to different performance of a search algorithm, but standard settings usually perform well [5]. (1+1) EA is simpler than GAs, but in previous software testing work we found that it can be more effective in some cases (e.g., [3]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [1].

4.3.4 Artificial Problems Design

In addition, to empirically evaluate whether the fitness function defined in Sect. 4.2 really address our optimization problem, we created artificial problems inspired from famous crowdsourcing platforms such as TopCode and uTest. Topcoder has 480,000 software developers, algorithmists, and digital designers, whereas Utest has 60,000 testers. Keeping this information, we created a crowd C of size 60,000 for our pilot study. For each bidder in the crowd, we assigned random values for the four parameters: $SuccessfulRating$, $CustomerRating$, $Experience$, and $PaymentHistory$. Each value ranges from 0 to 1.

After populating the crowd, we created projects with various characteristics. In total, we created 6000 projects. The budget for each project ranged from 100USD C 10000USD with the increment of 100USD. Since each project can have a minimum and maximum budget value, the minimum value was set to 10 % less of the given project budget and the maximum value was set to 10 % more of the given project budget. For example, if the given project is 100USD, then the minimum budget would be 90 and maximum budget would be 110. For each project, we set a series of number of team sizes, which are as: 2–4, 5–7, 8–10, 11–13, 14–16.

For each project, we set the number of bidders into the following three classes:

- Low (20, 50, 80, 100)
- Medium (200, 300, 400, 500)
- High (1000, 2000, 3000, 4000)

For each bidder, we randomly generate a value for $RBid$ from 0 to $\frac{Budget_{max}}{teamSize_{max}}$ of a project and to make $RBid$ fair based on the four parameters, i.e., $SuccessfulRating$, $CustomerRating$, $Experience$, and $PaymentHistory$, we modified the generated $RBid$ as follows:

$$\frac{SuccessfulRating + CustomerRating + Experience + PaymentHistory}{4} * \frac{Budget_{max}}{teamSize_{max}} \tag{9}$$

Moreover, we restricted search algorithms to generate a bid value ranging from $0.5RBid$–$1.5RBid$. The purpose was to avoid generating unrealistic bids values.

4.3.5 Statistical Tests

To compare the obtained results of the four search algorithms, the Kruskal-Wallis test, the Wilcoxon signed-rank test and the Vargha and Delaney statistics are used, based on the guidelines for reporting statistical tests for randomized algorithms presented in [3, 8].

To check if there are significant differences across the four algorithms, we first performed the KruskalCWallis test. Obtained p-value indicates whether there is significant difference among the four algorithms. However this test does not tell us which algorithm is significantly different with which algorithm. Therefore, we further performed the Wilcoxon signed-rank test to calculate a p-value for deciding whether there is a significant difference between a pair of search algorithms. We chose the significance level of 0.05, which means there is a significant difference if a p-value is less than 0.05.

As investigated in [3], it is not sufficient to interpret results only using p-values. To better interpret the results, the statistical test results must be interpreted in conjunction with an effect size measure, which helps determining practical significance of the results. We used the Vargha and Delaney statistics ($\widehat{A}_{12}$) to calculate the effect size measure, which is selected based on the guidelines proposed in [3]. In our context, given the fitness function FS ($f_{Fitness}(n)$), $\widehat{A}_{12}$ is used to compare the probability of yielding highest fitness value (low FS value) for two algorithms A and B. If $\widehat{A}_{12}$ is equal to 0.5, the two algorithms are equivalent. If $\widehat{A}_{12}$ is more than 0.5, it means the first algorithm A has higher chances of obtaining higher fitness value than B.

4.3.6 Experiment Execution

For each of the 100 artificial problems, we ran experiments 100 times for each of the four search algorithms for each problem. We let all the four algorithms run up to 2000 generations for each problem and collected final fitness value calculated in the 2000th generation. We used a PC with Intel Core Duo CPU 2.20 GHz with 4 GB of RAM, running Linux Ubuntu operating system for the execution of experiment.

4.3.7 Results and Analysis

To answer our research questions, we compared the three search algorithms with RS based on mean fitness values achieved after 2000 generations for each algorithm and each of the 100 problems. Recall that each problem was repeated for 100 times to account for random variation.

Table 1 provides the Vargha and Delaney statistics. The column $A > B$ means the number of problems out of 100 for which an algorithm A has higher chances of obtaining higher fitness value than B, $A < B$ means vice versa, and $A = B$ means the number of problems for which there were no differences between A and B as $\widehat{A}_{12}$ equals to 0.5.

Table 2 summarizes results of the Wilcoxon signed-rank test for RQ1 and RQ2. The column $A > B$ means the number of problems out of 100 for which an algorithm A was significantly better than B, $A < B$ means vice versa, and $A = B$ means the number of problems for which there were no significant differences between A and B based on p-values calculated by the Wilcoxon test.

Results for RQ1

To answer RQ1, we compared each search algorithm with RS, based on the mean fitness values of 100 runs obtained for each problem. Results for RQ1 are shown in the first three rows of Tables 1 and 2.

AVM versus RS: As we can see in Table 1, AVM performed better than RS for 4045 problems but for 1889 problems the results were statistically significant (Table 2).

Table 1 Results for the Vargha and Delaney $\widehat{A}_{12}$ statistics

	Pair of Algorithms (A vs. B)	$A > B$	$A < B$	$A = B$
RQ1	AVM versus RS	4045	1953	2
	(1+1)EA versus RS	2847	3149	4
	GA versus RS	2883	3115	2
RQ2	AVM vs (1+1)EA	4104	1892	4
	AVM versus GA	4115	1880	5
	(1+1)EA versus GA	2873	3127	0

Table 2 Results for the Wilcoxon signed-rank test at significance level of 0.05-artificial problems

	Pair of algorithms (A vs. B)	$A > B$	$A < B$	$A = B$
RQ1	AVM versus RS	1889	28	4083
	(1+1)EA versus RS	143	186	5671
	GA versus RS	151	150	5699
RQ2	AVM vs (1+1)EA	1931	14	4055
	AVM versus GA	1881	19	4100
	(1+1)EA versus GA	156	158	5686

RS performed better for 1953 problems as shown in the first row of Table 1, and there were no significant differences for 4083 problems.

(1+1) EA versus RS: (1+1) EA performed better than RS for 2847 problems (Table 1), 143 problems out of which were significantly better than RS (Table 2). There were no significant differences for 5671 problems (Table 2).

GA versus RS: In case of GA, it performed better than RS for 2883 problems (Table 1). Out of 2883, for 151 problems GA was significantly better than RS (Table 2). For 5699 problems there were no significant differences (Table 2).

Concluding Remarks: Based on the above results, we can answer RQ1 as follows: AVM is significantly better than RS for finding an optimal solution for our problem. For other two algorithms (GA and (1+1) EA), we didn't observe significant differences than RS.

Results for RQ2

The results to answer RQ2 are presented in the last three rows of Tables 1 and 2. These results are also based on the mean fitness values obtained for each problem for each algorithm after running the problem 100 times.

AVM versus (1+1) EA: AVM performed better than (1+1) EA for 4104 problems (Table 1), but for 1931 problems it was significantly better than (1+1) EA (Table 2). AVM performed worse than (1+1) EA for 1892 problems (Table 2) and in 14 out of these 1892 problems (1+1) EA was significantly better than AVM (Table 2). There were no significant differences between the algorithms for 4055 problems as shown in Table 2.

AVM versus GA: AVM performed better than GA for 4115 problems (Table 1) and out of these 4115 problems AVM performed significantly better than GA for 1881 problems (Table 2). AVM performed significantly worse than AVM for 19 problems (Table 2).

(1+1) EA versus GA: Regarding the (1+1) EA versus GA pair, as we can see from Table 2 that (1+1) EA was significantly better than GA for 156 problems, whereas GA was significantly better than (1+1) EA for 158 problems. For the rest of the problems, there were no significant differences.

Concluding Remarks: Based on the above results, we can answer RQ2 as follows: AVM is the best algorithm in terms of finding an optimal solution in our context and

the rest of the algorithms performed worse than AVM and there were no significant difference between the performance of the three algorithms.

Discussion
In this section, we provide an overall discussion based on the results of the experiments.

We observed from the results that AVM is significantly better than RS in finding an optimal solution (RQ1) and for the rest of the algorithms there are no significant differences than RS. Among all the studied algorithms, AVM is significantly better than the rest of the algorithms (RQ2).

The performance of algorithms can be argued based on their working. AVM works is a local search algorithm. If the fitness function provides a clear gradient towards the global optima, then AVM will quickly converge to one of them, which might be the case for our current context. On the other hand, (1+1) EA puts more focus on the exploration of the search landscape. When there is a clear gradient toward global optima, (1+1) EA is still able to reach those optima in reasonable time, but will spend some time in exploring other areas of the search space. This latter property becomes essential in difficult landscapes where there are many local optima. In these cases, AVM gets stuck and has to restart from other points in the search landscape. On the other hand, (1+1) EA, thanks to its mutation operator, has always a non-zero probability of escaping from local optima. Similar is the case for GA, which tries to explore (mutation operator) and exploit (crossover) the search space and hence may require more generations. By increasing the number of generations, we expect that the performance of GA and (1+1) EA can be improved.

Based on the above results, we can conclude that in our current context AVM has the ability to solve a wide range of problems. However, more experiments are needed in the future to thoroughly evaluate our fitness function with the real data from crowdsourcing platforms.

5 Threats to Validity

To reduce construct validity threats, we chose an effectiveness measure called fitness value, which is comparable across all four search algorithms (AVM, (1+1) EA, GA and RS. Furthermore, we used the same stopping criterion for all algorithms, i.e., number of generations. This criterion is a comparable measure of efficiency across all the algorithms.

The most probable conclusion validity threat in experiments involving randomized algorithms is due to random variations. To address it, we repeated experiments 100 times to reduce the possibility that the results were obtained by chance. Furthermore, we performed the Wilcoxon test to compare the algorithms mean fitness values of 100 runs and determine the statistical significance of the results. We chose this test since it is appropriate for the continuous data [4], thus matching our situation. To determine the practical significance of the results obtained, we measured the effect

size using the $\widehat{A}_{12}$ values, which is recommended to be used in conjunction with the Wilcoxon test to better interpret the results [3].

A possible threat to internal validity is that we have experimented with only one configuration setting for the GA parameters. However, these settings are in accordance with the common guidelines in the literature and our previous experience on testing problems. Parameter tuning can improve the performance of GAs, although default parameters often provide reasonable results [5].

One common external validity threat in the software engineering experiments is about generalization of results. To deal with this, we conducted an empirical evaluation of our proposed fitness function using 6000 artificial problems of varying complexity.

6 Conclusion and Future Work

To compare with traditional software engineering development, crowdsourcing software engineering, especially for developing large-scale software, is still far away from being mature. In this paper, we propose a search-based approach to make a very first step toward this direction by providing an automated, scalable and intelligent solution to assist platform managers to find an optimal solution when forming a virtual team for a submitted task via a crowdsourcing platform. We conducted a pilot study and results show that AVM is a promising search algorithm, together with the defined fitness function, can efficiently find an optimal solution for our problems. In the future, we plan to conduct more experiments based on real data that can be collected from existing crowdsourcing platforms such as Topcoder and UTest. We also plan to provide an integrated solution starting from specifying tasks and profiles of registrants, automatically collecting data for search, until providing feedback to end users such as registrants, platform managers and submitters in a transparent manner. By doing so, we hope, in certain extent, we can improve the quality and productivityranking, repuation and reward system of the current practice of performing soft-ware engineering tasks via crowdsourcing platforms.

References

1. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. **36**(6), 742–762 (2010)
2. Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: 2010 Third International Conference on Software Testing, Verification and Validation (ICST), pp. 205–214. IEEE (2010)
3. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Softw. Test. Verif. and Reliab. **23**(2), 119–147 (2013)

4. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 1–10. IEEE (2011)
5. Arcuri, A., Fraser, G.: Search based software engineering. On Parameter Tuning in Search Based Software Engineering, pp. 33–47. Springer, New York (2011)
6. Bagnall, A.J., Rayward-Smith, V.J., Whittley, I.M.: The next release problem. Info. Softw. Technol. **43**(14), 883–890 (2001)
7. Baker, P., Harman, M., Steinhofel, K., Skaliotis, A.: Search based approaches to component selection and prioritization for the next release problem. In: 22nd IEEE International Conference on Software Maintenance. ICSM'06. pp. 176–185. IEEE (2006)
8. Burke, E.K., Kendall, G.: Search Methodologies. Springer, New York (2005)
9. Droste, S., Jansen, T., Wegener, I.: On the analysis of the (1+ 1) evolutionary algorithm. Theor. Comput. Sci. **276**(1), 51–81 (2002)
10. Finkelstein, A., Harman, M., Mansouri, S.A., Ren, J., Zhang, Y.: A search based approach to fairness analysis in requirement assignments to aid negotiation, mediation and decision making. Requir. Eng. **14**(4), 231–245 (2009)
11. Harman, M., Mansouri, S.A., Zhang, Y.: Search based software engineering: a comprehensive analysis and review of trends techniques and applications. Technical Report Department of Computer Science, Kings College London, TR-09-03 (2009)
12. Korel, B.: Automated software test data generation. IEEE Trans. Softw. Eng. **16**(8), 870–879 (1990)
13. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004)
14. Yue, T., Ali, S.: Applying search algorithms for optimizing stakeholders familiarity and balancing workload in requirements assignment. In: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, pp. 1295–1302. ACM, New York (2014)