CodeRAnts: A recommendation method based on collaborative searching and ant colonies, applied to reusing of open source code

Coderants: método de recomendación basado en la búsqueda colaborativa y las colonias de hormigas. Aplicado a la reutilización del código fuente abierto

I. Caicedo-Castro[1] and H. Duarte-Amaya[2]

ABSTRACT

This paper presents CodeRAnts, a new recommendation method based on a collaborative searching technique and inspired on the ant colony metaphor. This method aims to fill the gap in the current state of the matter regarding recommender systems for software reuse, for which prior works present two problems. The first is that, recommender systems based on these works cannot learn from the collaboration of programmers and second, outcomes of assessments carried out on these systems present low precision measures and recall and in some of these systems, these metrics have not been evaluated. The work presented in this paper contributes a recommendation method, which solves these problems.

Keywords: Recommender Systems on Software Engineering, recommendation method based on collaborative searching, software reuse, open source software, ant colony.

RESUMEN

Este artículo presenta CodeRAnts: un nuevo método de recomendación basado en la técnica de búsqueda colaborativa e inspirada en la metáfora de la colonia de hormigas. Este método es propuesto con el objetivo de llenar el vacío en el estado del arte en cuanto a los sistemas de recomendación diseñados para reutilizar software, cuyos trabajos previos presentan dos problemas. El primero, es que los sistemas de recomendación basados en esos trabajos no pueden aprender de la colaboración de los programadores, y segundo, que los resultados de las pruebas realizados sobre estos sistemas presentan medidas bajas de precisión y remembranza, incluso, en algunos de estos sistemas no se hizo una evaluación de estas métricas. La contribución de este trabajo es un método de recomendación que resuelva dichos problemas.

Palabras clave: Sistemas de recomendación para ingeniería de software, método de recomendación basado en la búsqueda colaborativa, reutilización de software, software de fuente abierta y colonia de hormigas.

Received: April 28th 2013
Accepted: December 4th 2013

Introduction

This paper presents the concepts and design taken into account in CodeRAnts, which is a new recommendation method proposed to assist software engineers and computer programmers in the reuse of source code by allowing them to retrieve useful snippets of code (potentially written in any programming language) for the implementation of new software products. CodeRAnts is based on two approaches. The first is collaborative searching, which takes advantage of the similarity and repetition of queries that have been used by programmers, who are the stakeholders in the search for snippets of code. The second is the ant colony metaphor. We consider this approach to tackle two issues. First, it pretends to solve the cold start problem; for example, a system that implements CodeRAnts can suggest snippets of code, even as it receives new queries. Secondly, it initiates the use of system of recommendations, the structure used to save the queries will have little information; therefore, it is necessary to solve the problem related with the data sparsity of the query-ranking matrix, which is used in collaborative searching.

The preliminary evaluation carried out in this work shows better values for the metrics of precision and recall than those achieved in the state of the art. These metrics are the most commonly used to evaluate recommender systems (Basu et al., 1998; Billsus and

[1] Isaac Caicedo. BSc in Computer Science Engineering, Pontificia Universidad Bolivariana, Colombia. MSc in Engineering, University of Los Andes, Colombia. Dr. Eng. (c), Universidad National de Colombia, Colombia. Affiliation: Full time professor, University of Córdoba, Colombia. E-mail: ibcaicedoc@unal.edu.co
[2] Helga Duarte. BSc in Systems and Computing Engineering, Universidad National de Colombia, Colombia. MSc in Engineering, Universidad National de Colombia, Colombia. PhD in Computer Sciences, Joseph Fourier University, Grenoble I, France. Affiliation: Full time professor, Universidad National de Colombia, Colombia.
E-mail: hduarte@unal.edu.co

How to cite: Caicedo, I., Duarte, H., CodeRAnts: A recommendation method based on collaborative searching and ant colonies, applied to reusing of open source code., Ingeniería e Investigación, Vol. 34, No. 1, April, 2014, pp. 72 – 78.

Pazzani, 1998; Sarwar et al., 2000a,b; Picault et al., 2010; Bedi and Sharma, 2012). In mathematical terms, *precision* (see expression 1) is the number of retrieved and relevant items, divided by the total number of retrieved items. On the other hand, *recall* (see expression 2) is the number of retrieved relevant items, divided by the total number of relevant items.

$$precision = \frac{relevantItems \cap retrievedItems}{retrievedItems} \quad (1)$$

$$recall = \frac{retrievedItems \cap relevantItems}{relevantItems} \quad (2)$$

Motivation

According to Ricci et al. (2010), a *recommender system* is a set of software tools and techniques which provide suggestions of worthy items for users. These suggestions are related to several decision-making processes that are difficult when users have a large amount of optional items to choose from. In the e-commerce context, these processes are related to the buying of items such as books. Amazon's recommender system, for example, assists its users in finding books that meet their needs.

The open source software engineering context is similar to that used by e-commerce. Today, there is a substantial amount of open source code available on the World Wide Web, which is stored inside repositories and available through search engines (e.g., Koders, Krugle Sourceforge, Google code, GitHub, and CodePlex). This source code belongs to world class software products (e.g., Linux operating system kernel, JBoss application server, GNU Emacs, etc). This plethora of source code is available to be reused. In fact, software reuse is acknowledged as an important activity because it allows programmers to use preexisting core assets or artifacts rather than creating them from scratch. Indeed, Raymond (1999, p. 4) highlights the importance of software reuse: "Good programmers know what to write. Great ones know what to rewrite (and reuse)".

Moreover, computer scientists such as Mcilroy (1968), Standish (1984), Brooks (1987), Poulin et al. (1993), Boehm (1999), and Pohl and Böckle (2005), have highlighted the following advantages of reusing software: i) reducing time and costs, ii) improving the quality of software, iii) reducing amount of defects and iv) by reusing code there is a higher chance of detecting failures and fixing them.

Search engines allow programmers to find useful source code; however, some problems still remain: i) the probability that two people choose the same word to describe a concept is less than 20% (Furnas et al., 1987; Harman, 1995), ii) users who consider search engines useful for finding code are those who know how to employ the search (Bajracharya and Lopes, 2010), iii) in a study by Coyle and Smyth (2007) more than 20,000 queries were used: its results showed that, on average, Google delivered at least one useful result only 48% of the time, iv) in the domain of collaborative-based recommender systems, research indicates that the design problems of search engines are twofold: solitary nature and one-size-fits-all. (Resnick and Varian, 1997; Balabanovic and Shoham, 1997; Schafer et al., 1999; Jameson and Smyth, 2007; Smyth, 2007; Morris, 2008).

On the one hand, *solitary nature* means that searches take the form of an isolated interaction among the user and the search engine. Due to this drawback, search engines overlook the experience of users, which is useful for offering a more accurate result list in comparison to the others with similar preferences. On the other hand, *one-size-fits-all* means that several users achieve the same result list when they use the same query in spite of having different preferences.

The same researchers highlight the importance of recommender systems technology, in particular, the concepts of collaboration and user preferences, in order to cope with the above mentioned search engine design problems. *Preference* is information about users' needs and the *collaboration* concept refers to preferences supplied by a group (or community) of users. The solution proposed, consists of influencing recommendations with information learned from users' preferences and their collaboration, thereby, suggestions are guided by users' behavior rather than only the items' features.

For instance, if a user performs the following query: "I need something with four legs where I can sit down". The search engine's answer is a result list with items like: horses, tigers, chairs and tables. These objects match the user's query. A search engine replies regardless of the user's preferences and the collaboration of similar users. Even though the user selects the chair, the search engine is not able to learn the preference of the active user, and hence, the engine cannot change the relevance level of the chair for future users with the same preference. Conversely, a recommender system suggests items based on what it has learned from the users' preferences and collaboration. In this case, the suggestion of the recommender system is to use a chair. This illustrative example depicts the advantages of recommendation techniques based on collaboration, which motivated the design of CodeRAnts.

Related Work

In the context of software engineering, Robillard et al. (2010) define recommender systems as a set of software applications that provide information items, which are considered valuable to perform software engineering tasks, e.g., reusing artifacts, maintenance of software products and the identification of defects and bugs.

Various recommender systems have been created to assist software tasks, e.g., eRose (Zimmermann et al., 2005), Suade (Robillard et al. 2008), Dhruv (Ankolekar et al., 2006), and Expertise Browser (Mockus and Herbsleb, 2002). However, in the particular context of software reuse, several recommender systems have been made: CodeBroker (Ye and Fischer, 2005), Hipikat (Cubranic et al., 2005), Strathcona (Holmes et al., 2006), ParseWeb (Thummalapenta and Xie, 2007), and ORIPC (Outil de Recommandation et Instanciation des Patrons de Conception) (Bouassida et al., 2011).

All prior recommender systems for software reuse have presented two important drawbacks, which represent the gap in the state of the art: i) These recommender systems are not able to learn from users' preferences and their collaboration. Therefore, these systems cannot learn to identify items which were considered useful by users in past, hence, these systems will not suggest them in the future. Consequently, these recommender systems have the same problems as search engines that are mentioned above, i.e., solitary nature and one-size-fits-all. ii) Only in Hipikat and CodeBroker were precision and recall measures were evaluated and these indicators are still far from being satisfactory. It is possible that in the other systems, these metrics were not assessed because a dataset with information about programmers retrieving source code did not exist, as in the case of other kinds of systems based on collaborative filtering, which are assessed with

classical datasets such as Jester (http://www.ieor.berkely.edu/~goldberg/jester-data) and MovieLens (http://www.grouplens.org/node/73).

Table 1 presents a summary of the literature review on recommender systems for software reuse. In this table, the third requirement (*Rq3*) is missing because these systems lack of the second one (*Rq2*), due to a recommender system, which cannot learn from preferences and collaboration of its users, shares the design problems of search engines (one-size-fits-all and solitary nature). The contribution of this work is a recommendation technique designed to fulfill all requirements described in this table.

Table 1. Comparison of recommender systems for software reuse according to the requirements described at the bottom of this table. In this table, X means a fulfilled requirement, and NA means a requirement that has not been assessed.

Recommender system	Rq1	Rq2	Rq3	Rq4
CodeBroker (Ye and Fischer, 2005)	X			X
Hipikat (Cubranic et al., 2005)	X			X
Strathcona (Holmes et al., 2006)	X		NA	X
ParseWeb (Thummalapenta and Xie, 2007)	X		NA	X
ORIPC (Bouassida et al., 2011)			NA	X

Rq1: The recommender system can reuse code stored inside open source repositories. *Rq2*: The recommender system can learn from preferences and collaboration of its users. *Rq3*: Results of assessment of the recommender system with respect to the values of precision and recall are acceptable. *Rq4*: The recommender system lacks a cold start problem.

Design of CodeRAnts

CodeRAnts is designed to be implemented in a recommender system with a proxy architecture, where this system is the proxy of a code search engine (e.g., Koders). Fig. 1 depicts a recommender system that implements the CodeRAnts method and this system may be plugged to a search engine (or other recommendation system like Strathcona). The explanation of its operation is as follows: 1) The proxy agent receives the query, q, which comes from user agent, 2) the proxy agent redirects q to code search engine, 3) the code search engine retrieves a result list, a, with links to code that could be useful for the user, 4) the proxy agent sends forward a to recommender systems 5) recommender systems compute a new result list, R, in accordance with the recommendation technique described below (CodeRAnts), and sends R to the user agent.

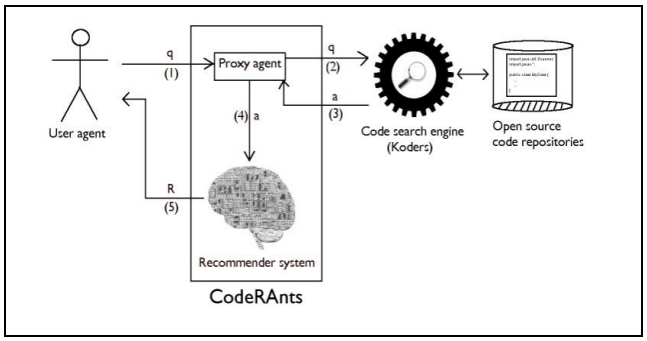

Figure 1. CodeRAnts method implemented in a recommender system

CodeRAnts method

Similar to Collaborative Searching (Smyth et al., 2010), the design goal of CodeRAnts is to take advantage of similarity and repetition of queries performed by programmer communities as a source of recommendations.

Nevertheless, the collaborative searching approach has similar research challenges to that of collaborative filtering, i.e., data sparsity of the input-ranking matrix and cold start problem for queries recently used. Therefore, in order to address these drawbacks, we have taken into account the ant colony metaphor, which was successfully used by Bedi and Sharma (2010) to overcome these problems in the context of collaborative filtering, achieving good values of precision and recall through off-line assessment.

Algorithms based on this metaphor are those that reproduce the behavior of real ants in order to build better solutions, by using artificial pheromones as a means of communication among ants, which tend to lay pheromone trails while walking from their nests to the food source and vice versa.

Ants do not communicate directly with each other. These insects are guided by pheromone smell and hence, ants choose paths marked by the highest concentration of pheromones. The indirect communication among ants through pheromone trails enables them to find a shorter path between their nest and food sources.

The CodeRAnts method consists of creating a directed graph, whose vertexes represent queries performed in the past and the weight edge is based on textual similarity, the correlation, and the confidence between vertexes.

Let $Q = \{q_1,\ q_2, \dots, q_m\}$ be a set of queries performed by programmers in the past and $C = \{c_1,\ c_2, \dots, c_n\}$, a set of snippets of code. In the same way that the collaborative searching method proposed by Smyth et al. (2010), computes the matrix, the CodeRAnts method also computes the matrix $QC_{m\,x\,n}$, such that $QC_{i,j}$ corresponds to the amount of times that the snippet of code c_j was retrieved when the query q_i was used by a programmer in the past.

Similar to the technique proposed by of Bedi and Sharma (2012), the CodeRAnts method is structured in two processes. The first is performed off-line; it consists of creating the directed graph, which shall be used as a set of paths with pheromone trails for ants. The second is on-line and it is designed to generate recommendations through ant movement in order to find the goal, namely, to collect a ranking for each snippet of source code.

The off-line process is described in the following two steps: i) the matrix of rankings $R^{QC}_{m\,x\,n}$ is initialized by normalizing the $QC_{m\,x\,n}$ matrix: $\forall_{i,j} : R^{QC}_{i,j} \leftarrow \frac{QC_{i,j}}{\sum_{k=1}^{m} QC_{i,k}}$, where $i, j, k, m, n, QC_{i,j} \in \mathbb{N}, 1 \leq j \leq m,\ 1 \leq k \leq m, 1 \leq i \leq n, R^{QC}_{i,j} \in \mathbb{R}$, and $0 \leq R^{QC}_{i,j} \leq 1$.

For the purpose of illustrating this step, let us consider the $QC_{m\,x\,n}$ matrix depicted in Table 2, where $q_1 = comprimir$, $q_2 = compact$, $q_3 = compress$, $q_4 = zip$, and $q_3 = conpresser$. Table 3 shows the normalized $QC_{m\,x\,n}$ matrix, namely, $R^{QC}_{m\,x\,n}$.

Table 2. Instance of $QC_{m\,x\,n}$ matrix.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7
q_1	0	0	0	3	0	0	0
q_2	0	2	0	1	0	0	0
q_3	0	0	0	1	2	1	0
q_4	4	0	2	0	2	1	0
q_5	4	0	9	0	1	0	1

ii) the directed graph $G = (V, E)$ is created; let V be a set of vertexes and let E be a set of edges. The vertexes represent queries performed by programmers in the past, hence, $V = Q$. On the other hand, edges are links between queries. Each edge represents

a path where ants have laid pheromone trails at certain time t; therefore, edge weigh is the level of pheromone track at time t, that is denoted by queries $\rho_{q_i,q_j}(t)$, q_i and q_j. $\rho_{q_i,q_j}(t)$ is computed based on similarities and confidence among vertexes. If the level of pheromone among two vertexes is equal to zero, then there is no edge between both vertexes.

Table 3. Instance of the matrix of rankings R^{QC}_{mxn} that is achieved from QC_{mxn} matrix, in Table 2, by normalizing it.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7
q_1	0	0	0	1	0	0	0
q_2	0	0.67	0	0.33	0	0	0
q_3	0	0	0	0.25	0.5	0.25	0
q_4	0.5	0	0.2	0	0.2	0.1	0
q_5	0.26	0	0.6	0	0.07	0	0.07

Before delving into the similarity between queries, it is important to clarify certain issues concerning the language used to form queries. Recall from the previous section that a system, which implements CodeRAnts, has proxy architecture and it is plugged into a search engine, thereby query language is the same as that supported by the engine. For instance, if the recommender system is plugged into Koders, the queries are formed with words from natural language and with the same syntax supported by Koders by using identifiers such as cdef, fdef, mdef, idef, and sdef that refer to names of classes, files, methods, interfaces, and structures, respectively. In this particular case, a query could be: *cdef:util mdef:compress*. With this query, classes whose name contains the word *util* and whose method contains the word *compress are s*earched.

The similarity among queries is defined in Expression 3, where $\alpha, \beta \in \mathbb{R}, \alpha + \beta = 1$. These constants represent weights for balancing two similarity measures. The first, $sim_w(q_i, q_j)$ is the similarity among queries based on the number of edition operations performed (the edit distance proposed by Levenshtein (1966), $edDis(q_i, q_j)$) to transform q_i into q_j (see Expression 4). If $q_i = q_j$, then $sim_w(q_i, q_j) = 1$, this is, $sim_w(q_i, q_j)$ reaches its maximum value due to the edition distance equal to cero, $edDis(q_i, q_j) = 0$. Following with the above mentioned example, an engineer may use the following words for the query: compress, conpresser (in French), or comprimir (in Spanish). If $q_i = conpresser$, $q_j = compress$, and $th = 5$ (threshold equal to five), then $sim_w(q_i, q_j) = 0.4$, because there are three edits to change a query into the other one: 1) con*p*resser → co*m*presser (substitution of the letter *n* for *m*), 2) compresser → compresse (removal of letter *r*) and 3) compresse → compress (removal of letter e). Table 4 presents all computations of edit distance and sim_w between queries from the above mentioned example.

Table 4. Computation of edit distance, sim_w, sim_c, and sim, based on table 3, where $q_1 = comprimir$, $q_2 = compact$, $q_3 = compress$, $q_4 = zip$, $q_3 = conpresser$, and $th = 5$

	Edit distance	sim_w	sim_c	sim
q_1, q_2	5	0	0.2	0.14
q_1, q_3	4	0.2	0.27	0.25
q_1, q_4	8	0	0.32	0.22
q_1, q_5	5	0	0.12	0.09
q_2, q_3	4	0.2	0.39	0.34
q_2, q_4	6	0	0.47	0.33
q_2, q_5	7	0	0.19	0.13
q_3, q_4	7	0	0.62	0.44
q_3, q_5	3	0.4	0.25	0.29
q_4, q_5	9	0	0.29	0.21

The edit-distance-based measure is the extent of the typographical similarity between two queries. This has been considered in this work because sometimes, typographical mistakes are included within the users' query. Nevertheless, this measure does not consider the semantic similarity between two queries, e.g., $q_i = they$, and $q_j = the$. Therefore, $sim(q_i, q_j)$ is also based on the correlation between the rankings associated with both queries.

$$sim(q_i, q_j) = \alpha\, sim_w(q_i, q_j) + \beta\, sim_c(q_i, q_j) \quad (3)$$

$$sim_w(q_i, q_j) = \begin{cases} \dfrac{th - edDis(q_i, q_j)}{th} & if\ th > edDis(q_i, q_j) \\ o & otherwise \end{cases} \quad (4)$$

Let $sim_c(q_i, q_j)$ be the similarity based on correlation coefficient between queries; it is calculated using Expression 5. C_{q_i,q_j} is the correlation coefficient between the row vectors R_i^{QC} and R_j^{QC}, which is defined in Expression 6, where $C_{q_i,q_j} \in \mathbb{R}$, and $-1 \leq C_{q_i,q_j} \leq 1$, $\overline{R_i^{QC}}$ and σ_{q_i} represent the average and the standard deviation of the row vector R_i^{QC}, respectively. If the value of C_{q_i,q_j} trends to one, it means that both queries are correlated, but if the value is close to zero, there is no correlation, otherwise there is an inverse correlation. Table 4 shows all computations of sim_c between queries, taking into account the matrix of rankings R^{QC}_{mxn} depicted in Table 3.

$$sim_c(q_i, q_j) = \begin{cases} C_{q_i,q_j}\ if\ C_{q_i,q_j} > 0 \\ 0 \quad otherwise \end{cases} \quad (5)$$

$$C_{q_i,q_j} = \frac{\sum_{k=1}^{m}(R_{i,k}^{QC} - \overline{R_i^{QC}})(R_{j,k}^{QC} - \overline{R_j^{QC}})}{\sigma_{q_i}\sigma_{q_j}} \quad (6)$$

Confidence between queries is computed using Expression 7. $conf(q_j|q_i)$ is the conditional probability of making the query q_j given the query q_i. Table 5 also shows all computations of confidence between queries in the above mentioned example.

Table 5. Computation of confidence between queries (i.e. $conf(q_j|q_i)$)

	q_1	q_2	q_3	q_4	q_5
q_1	1	0.5	0.33	0	0
q_2	1	1	0.33	0	0
q_3	1	0.5	1	0.5	0.25
q_4	0	9	0.67	1	0.75
q_5	0	0	0.33	0.75	1

Fig. 2 depicts the directed graph created by using Expression 8, with initial pheromone paths, when $t = 0$, namely, $\rho_{q_i,q_j}(0)$, where $k \in \mathbb{R}, and\ k \to 0$ (i.e., k tends to be very small). $\tau(q_i, q_j)$ is a function based on similarity and confidence among queries, it is computed using Expression 9 (adapted from Bedi and Sharman, 2012). Table 6 shows all computations performed to create the directed graph.

$$conf(q_j|q_i) = \frac{retrieved\ code\ when\ both\ queries\ were\ performed}{retrieved\ code\ when\ q_i\ was\ performed} \quad (7)$$

$$\rho_{q_i,q_j}(0) = \begin{cases} \tau(q_i, q_j)\ if\ sim(q_i, q_j) \neq 0\ and\ conf(q_i, q_j) \neq 0 \\ k\ conf(q_i, q_j)\ if\ sim(q_i, q_j) = 0\ and\ conf(q_i, q_j) \neq 0 \\ 0 \quad otherwise \end{cases} \quad (8)$$

$$\tau(q_i, q_j) = \frac{2 \times sim(q_i, q_j) \times conf(q_j|q_i)}{sim(q_i, q_j) + conf(q_j|q_i)} \quad (9)$$

Table 6. Computation of initial level of pheromones in the directed graph in Fig 3

	q_1	q_2	q_3	q_4	q_5
q_1	I	0.22	0.28	0	0
q_2	0.25	I	0.33	0	0
q_3	0.4	0.4	I	0.47	0.27
q_4	0	0	0.53	I	0.32
q_5	0	0	0.31	0.32	I

Once the directed graph $G = (V, E)$ has been created, as in the example depicted in Fig. 2, the on-line process may start. In this process, an active query vertex is selected by searching the vertex most similar to the query, q, sent by a programmer. Given that this algorithm is inspired in the ant colony, for all the generated ants, there is a time to live (TTL) parameter associated with the number of iterations that the ants can explore in the graph G. If the destination vertex is not found within TTL limit, each ant is removed. Due to the fact that the destination vertex is not known, or in the worst case does not exist, it is mandatory to setup a stop point (the TTL parameter) for the on-line process in order to prevent it from running indefinitely.

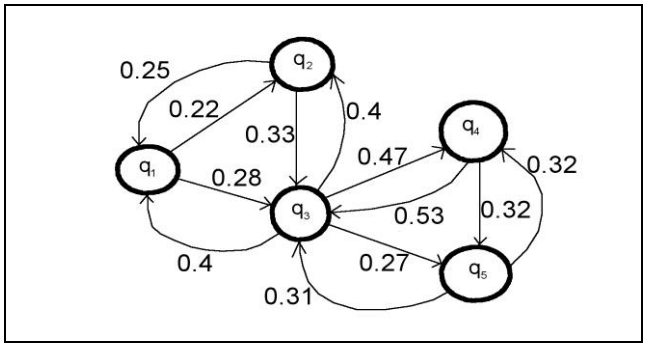

Figure 2. Instance of Graph G with pheromone paths

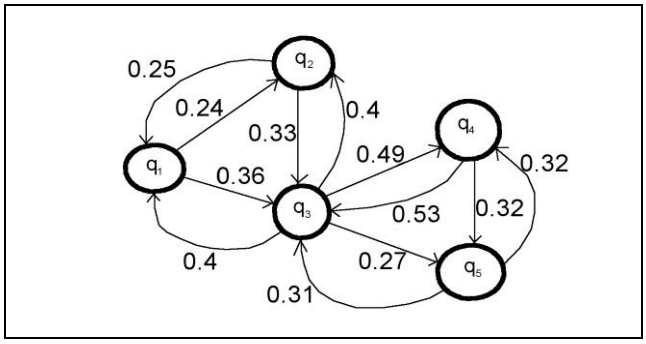

Figure 3. Pheromone graph G updated from the one depicted in Fig. 2.

The virtual ants' source of food consists of ranking most of the items. Hence, ants move through queries, which are similar to or probably related with the active query in order to collect their rank for those snippets of code that are not ranked for the active query. The on-line process of CodeRAnts is described in the following steps:

i) Seek an active query vertex, a, which is selected if it is the most similar to query, q, sent by a programmer. Suppose an engineer makes a query with the word comprimir (i.e., $q = "comprimir"$); by using edition distance as a method for measuring similarity between queries, q_1 is the most similar vertex in the graph depicted in Fig. 3 because there is no edition distance between q and q_1 due to the fact that both are exactly alike. Hereafter, for this example, q_1 is the active query vertex (i.e., $q_1 = a$) in the graph depicted in Fig. 2.

ii) Create x amount of ants, where x is equal to number of outgoing edges from active query vertex, a. In Fig. 2, if the active query vertex is q_1, then two ants are created, because this vertex has two outgoing edges.

iii) Each x-th ant selects the next vertex to be visited with the probability $P^k_{q_i,q_j} = \max(\rho_{q_i,q_j}(t) \times \frac{traced_{q_j}}{unrated_a})$ (adapted from Bedi and Sharman, 2012), where $q_j \notin tabu(x)$, $tabu(x)$ is a set of vertexes which the x-th ant has visited. $traced_{q_j}$ denotes the amount of code snippets traced by the vertex q_j, which have not been ranked by the active vertex a. $unrated_a$ is the total number of code snippets that have not been ranked by the active vertex a. The x-th ant will stop when all its adjacent vertexes are in the set $tabu(x)$. The whole process will finish when all ants may not move anymore or when the ants' TTL reaches its maximum value.

Taking q_1 as the active query vertex, Table 3 shows that snippets of code c_1, c_2, c_3, c_5, c_6 and c_7 do not have a ranking in q_1. Among the neighbors of q_1, q_2 there is a ranking for c_2 and q_3 has a ranking for c_5 and c_6, hence, these rankings are collected. Therefore, c_1, c_3, and c_7 are still without a ranking. Thereafter, ants keep moving, by choosing q_3 as the new destination vertex because the path between q_1 and q_3 has the greatest concentration of pheromones. This step is repeated without passing twice through the same vertex until ants reach their goal (i.e., to rank all snippets of code), or until maximum TTL is reached. It is important to clarify that when a certain snippet of code, c_i, is ranked by at least two neighbor vertexes, the ranking provided by the neighbors is stored, descending sorted, in accordance with trails of pheromone between the active vertex and its neighbors. For instance, when ants are on the vertex q_3, the snippet of code c_1 shall be ranked by its neighbor vertexes q_4 and q_5. In this example, the ranking provided by q_4 is stored before the other one provided by q_5, due to the fact that trails of pheromones between q_3 and q_4 are stronger than the concentration of pheromones among q_3 and q_5.

iv) Generate suggestions through the method proposed by Resnick et al. (1994), with the expression 10, where r_{q_i,c_k} and r_{q_j,c_k} represent the rankings of vertexes q_i and q_j for the snippet of code c_k, respectively. $\bar{r}_{q_i}$ and $\bar{r}_{q_j}$ denote the average rankings of vertexes q_i and q_j, respectively. top_Q is the number of first neighbors of the vertex q_i with the biggest trail of pheromones. For example, if $top_Q = 10$, then $r_{q_1,c_4} = 1 + \frac{1(1-1) + 0.22(0.33-0.5) + 0.28(0.25-0.33)}{1+0.22+0.28} = 0.96$

v) Finally, update $\rho_{q_i,q_j}(t)$ with Expression 11, where ε is the evaporation rate of pheromones and δ is computed with Expression 12, where $\eta_{a,q_k} = \frac{1}{d_{a,q_k}}$, and d_{a,q_k} represents the distance from vertex a to vertex q_k. Table 7 and Fig. 3 depict the update to the pheromone graph, when $\varepsilon = 0.01$.

$$r_{q_i,c_k} = \overline{r_{q_i}} + \frac{\sum_{j=1}^{top\_Q} \rho_{q_i,q_j}(r_{q_j,c_k} - \bar{r}_{q_j})}{\sum_{j=1}^{top\_Q} \rho_{q_i,q_j}} \quad (10)$$

$$\rho_{q_i,q_j}(t) = (1 - \varepsilon)\rho_{q_i,q_j}(t-1) + \delta_{q_i,q_j}(t-1) \quad (11)$$

$$\delta_{q_i,q_j}(t) = \eta_{a,q_k} \prod_{k=1}^{d_{a,q_k}} \rho_{q_i,q_k}(t) \times \frac{traced_{q_j}}{unrated_a} \quad (12)$$

Table 7. Update of pheromone graph in Fig. 6, when $\varepsilon = 0.01$. The meaning of each column is as follows: A $= (1-\varepsilon)\rho_{q_i,q_j}(t-1)$, B $= \eta_{a,q_k}$, C $= \prod_{k=1}^{d_{a,q_k}} \rho_{q_i,q_k}(t-1)$, D $= \frac{traced_{q_j}}{unrated_a}$, and E $= \rho_{q_i,q_j}(t)$.

	A	B	C	D	E
q_1, q_2	0.21	1	0.22	$\frac{1}{6}$	0.24
q_1, q_3	0.27	1	0.28	$\frac{2}{6}$	0.36
q_3, q_4	0.46	$\frac{1}{2}$	0.22 × 0.47 = 0.1	$\frac{4}{6}$	0.49
q_3, q_5	0.26	$\frac{1}{2}$	0.22 × 0.27 = 0.06	$\frac{4}{6}$	0.27

Simulation setting

Shani and Gunawardana (2010) present three methods to evaluate recommender systems, namely, off-line, user studies and on-line. In this work, CodeRAnts was evaluated through the first method with a simulation program written in Java. The program generates a bag-of-terms by assigning random values to the matrix *termsCode,* which holds the frequency of each term in the source code (i.e., if termsCode[3][4] is equal to 15, it means that the term t_3 appears fifteen times in the snippet of code c_4). The dataset is randomly generated due to the fact that there is not a real published dataset of interaction between programmers and search engines, or programmers and recommender systems based on collaborative searching.

A search engine and programmers are simulated in order to train and test the simulated recommender system, which implements the CodeRAnts method. In the training phase, programmers randomly choose certain snippets of code by performing searches through a simulated search engine. Queries are randomly selected from a dataset and these terms appear with high frequency in the snippet of code. In this way, the programmers' knowledge for performing a code search is simulated. The queries are words from natural language, which are used to write source code. This phase aims to fill the $QC_{m\,x\,n}$matrix, by recording which snippets of code are selected by programmers during the searches.

After the training phase, the simulator program performs the off-line phase of the CodeRAnt method. Then, the on-line phase begins the testing phase and other simulated programmers perform source code searches through a simulated recommender system. During the training phase, simulated programmers search at most 99 snippets of code and while testing phase is performed, the other instances seek at most, 25 snippets of code, which are stored in the dataset (i.e., relevant items). In both phases, programmers perform from 5 to 10 queries in order to search a snippet of code. Each query is randomly selected from a bag-of-queries. When a programmer finds a snippet of code, it is counted as a retrieved item in order to compute precision and recall metrics. Three fourths of the set of programmers are used for training and one fourth of this set is used for testing. Assessments were performed with sets of 30, 50, 100, 500, 1000, 5000, and 10000 programmers. Other parameters considered in the assessment are: $\alpha = 0.75$, $\beta = 0.25$, $th = 1$, timeout = 10, $\varepsilon = 10^{-2}$, and $k = 10^{-3}$. These parameters were chosen by running the simulation several times. Thus, we tuned the parameters until the best performance was achieved. The next section shows the results of the assessments with this experimental setting.

Results and discussion

Table 8 shows the results of the simulation. The average of precision and recall values is 0.53 and 0.71 respectively. These values are better than those achieved by Cubranic et al. (2005) with Hipikat, in average, 0.11 and 0.65 for precision and recall, respectively. Furthermore, the outcomes of the simulation performed on CodeRAnts are better than those observed by Ye and Fischer (2005) with CodeBroker, namely, with this system precision is not greater than 0.4, but recall reached 1.

Table 8. Results of assessment performed over CodeRAnts

Number of programmers	Precision	Recall
30	0.57	0.72
50	0.52	0.7
100	0.53	0.71
500	0.51	0.72
1000	0.51	0.7
5000	0.52	0.71
10000	0.52	0.7

Nevertheless, although with the simulations we achieved better values than those obtained by other researchers, this assessment method is not rigorous and its outcomes are slanted, given that the above mentioned systems were evaluated through user-based assessments. Cubranic et al. (2005) evaluated Hipikat with a group of real programmers and the Eclipse source code (version 2.1). In a similar fashion, Ye and Fischer (2005) carried out experiments over CodeBroker with real programmers, but with the Java 1.1.8 core library and the JGL 1.3 library. Thereby, for future studies, CodeRAnts must be assessed with the other systems using the same experimental method and setting. Additionally, CodeRAnts was assessed with a randomly generated dataset; hence, for further work we must collect a real dataset through user-based experiments.

The results of these experiments reveal that edit-distance-based similarity between queries is not useful because the best performance is achieved when the threshold parameter is equal to one. Hence, this is almost the same procedure as checking whether both queries are equal. In the future, we will assess other similarity measures (e.g., cosine distance, Euclidean distance, and etcetera).

Conclusions and directions for further work

The contributions of this study are: i) a recommendation method that can be implemented like a proxy of a code search engine or an above mentioned recommender system (e.g., Strahtcona), in order to allow them to improve their answers and recommendations for programmers; due to this, these systems could learn from users' collaboration through CodeRAnts. ii) a recommendation method which tackles the cold start problem given that a system which implements CodeRAnts can suggest snippets of code, although it receives new queries that do not belong to set Q, by searching other similar in this set. Moreover, through the ant colony technique, a system which implements CodeRAnts can suggest snippets of code, despite the fact that the matrix $R^{QC}_{m\,x\,n}$ does not have a ranking for these snippets given certain queries through the search performed by ants, through possibly related or correlated queries, and collecting ranking for these snippets, iii) a recommendation method, that in a simulated environment, has better preliminary values of precision and recall than prior systems designed for software reuse; however, it is important to highlight that the results of the simulations are not definitive evidence of

the quality of recommendation provided through of CodeRAnts method because in the simulation settings the bag-of-terms and the dataset are randomlygenerate; moreover, CodeRAnts was not evaluated with the same experimental method and setting carried out in the other systems by other researchers.

For future studies the following are proposed: i) collect a real dataset through user-based experiments, ii) carry out the evaluation of CodeRAnts with the other systems with the same experimental method and setting and iii) test other similar measures (e.g., cosine distance, Euclidean distance, etc.).

References

Ankolekar, A., Sycara, K., Herbsleb, J., Kraut, R., Welty, C., Supporting online problem-solving communities with the semantic web., Memoirs from 15th International conference on World Wide Web, ACM Press, 2006, pp. 575–584.

Bajracharya, S. K., Lopes, C. V., Analyzing and mining a code search engine usage log., 2010, Empirical Software Engineering, pp. 1—43.

Balabanovic, M., Shoham, Y., Fab: content-based, collaborative recommendation., Commun, Vol. 40, No. 3, 1997, ACM, pp. 66–72.

Basu, C., Hirsh, H., Cohen, W., Recommendation as classification: Using social and content-based information in recommendation., Proceedings of the 15th National Conference on Artificial Intelligence, 1998, AAAI Press, pp. 714–720.

Bedi, P., Sharma, R., Trust based recommender system using ant colony for trust computation., Expert Systems with Applications, Vol. 39, No. 1, 2012, pp. 1183 – 1190.

Billsus, D., Pazzani, M. J., Learning collaborative information filters., Proceedings of the 15th International Conference on Machine Learning, ICML '98, San Francisco, CA, USA, 1998, Morgan Kaufmann Publishers Inc, pp. 46–54.

Boehm, B. W., Managing software productivity and reuse., IEEE Computer, Vol. 32, No. 9, 1999, pp. 111–113. EPA, Title 40 Subchapter I-Solid waste, 258 criteria for municipal solid waste landfills., Environmental Protection Agency, USA, 2000.

Bouassida, N., Kouas, A., Ben-Abdallah, H. A design pattern recommendation approach., 2nd Interntional Conferences on Software Engineering and Service Science (ICSESS), 2011, IEEE Press.

Brooks, F., No silver bullet essence and accidents of software engineering., Computer, Vol. 20, No. 4, 1987, pp. 10–19.

Coyle, M., Smyth, B., Information recovery and discovery in collaborative web search., Proceedings of the 29th European conference on IR research, ECIR'07, Berlin, Heidelberg, Springer-Verlag, 2007, pp. 356–367.

Cubranic, D., Murphy, G. C., Singer, J., Booth, K. S., Hipikat: Aproject memory for software development., IEEE Trans. Software Eng., Vol. 31, No. 6, 2005, pp. 446–465.

Furnas, G. W., Landauer, T. K., Gomez, L. M., Dumais, S. T., The vocabulary problem in human-system communication. Commun. ACM, Vol. 30, No. 11, 1987, pp. 964–971.

Holmes, R., Walker, R., Murphy, G., Approximate structural context matching: An approach for recommending relevant examples., IEEE Trans. Software Eng., Vol. 32, No. 1, 2006, pp. 952–970.

Jameson, A., Smyth, B., The adaptive web. Chapter Recommendation to groups, Berlin, Heidelberg, Springer-Verlag, 2007, pp. 596–627.

Levenshtein, V. I., Binary codes capable of correcting deletions, insertions and reversals., Soviet Physics Doklady, 1966, pp. 707-710.

Mcilroy, D., Mass–produced software components. Software Engineering, conference by the NATO Science Committee, 1968, pp. 952–970.

Mockus, A., Herbsleb, J., Expertise browser: Aquantitative approach to identifying expertise. 24th International Conference on Software Engineering, New York, United States of America, IEEE CS Press, 2002, pp. 503–512.

Morris, M. R., A survey of collaborative web search practices., Proceedings of the 26th annual SIGCHI conference on Human factors in computing systems, CHI '08, 2008, New York, NY, USA, ACM, pp. 1657–1660.

Picault, J., Ribiere, M., Bonnefoy, D., Mercer, K., Recommeder systems Handbook, chapter 10: How to Get the Recommender Out of the Lab?, 2010, Springer–Verlag, pp. 579–614.

Pohl, K., Böckle, G., Software Product Line Engineering, Foundations, Principles, and Techniques., Springer–Verlag, Germany, 2005.

Poulin, J. S., Caruso, J. M., Hancock, D. R., The business case for software reuse. IBM Syst. J., Vol. 32, No. 4, 1993, pp. 567–594.

Raymond, E. S., The Cathedral and the Bazaar., 1st ed., Sebastopol, CA, USA, O'Reilly & Associates, Inc., 1999.

Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., Riedle, J., Grouplens: An open architecture for collaborative filtering of netnews., Conference on computer supported cooperative work, Chapel Hill, ACM Press, 1994, pp. 175–186.

Resnick, P., Varian, H. R., Recommender systems., Commun. ACM, Vol. 40, No. 3, 1997, pp. 56–58.

Ricci, F., Rokach, L., Shapira, B., Recommeder systems Handbook, chapter 1: Introduction to Recommeder systems Handbook., Springer–Verlag, 2010, pp. 1–38.

Robillard, M., Walker, R., Zimmermann, T., Foreword. Workshop on Recommendation Systems for Software Engineering., ACM Press, 2008.

Robillard, M., Walker, R., Zimmermann, T., Recommmedation systems for software engineering., Software IEEE, Vol. 27, No. 4, 2010, pp. 80–86.

Sarwar, B. M., Karypis, G., Konstan, J. A., Riedl, J. T., Application of dimensionality reduction in recommender system – a case study., ACM WebKDD 2000 Web Mining for ECommerce Workshop, Vol. 1625, No. 1, 2000, pp. 264–268.

Schafer, J. B., Konstan, J., Riedi, J., Recommender systems in ecommerce., Proceedings of the 1st ACM conference on Electronic commerce, EC '99, New York, NY, USA, ACM, 1999, pp. 158–166.

Shani, G. Gunawardana, A., Recommeder systems Handbook, chapter 8: Evaluating Recommendation Systems., Springer–Verlag, 2010, pp. 257–297.

Smyth, B., Case-based recommendation., In: The Adaptive Web, 2007, pp. 342–376.

Smyth, B., Coyle, M., Briggs, P., Recommeder systems Handbook, chapter 18: Communities, Collaboration, and Recommender Systems in Personalized Web Search., Springer–Verlag, 2010, pp. 579–614.

Thummalapenta, S., Xie, T., Parseweb: A programming assistant for reusing open source code on the web. IEEE/ACM International conferences on Automated Software Engineering, ACM Press, 2007, pp. 204–213.

Ye, Y., Fischer, G., Reuse-conducive development environments., Automated Software Eng., Vol. 12, No. 2, 2005, pp. 199–235.

Zimmermann, T., Zeller, A., Weissgerber, P., Diehl, S., Mining versión histories to guide software changes., IEEE Trans. Software Eng., Vol. 31, No. 6, 2005, pp. 429–445.