Automated Software Engineering (2022) 29:45
https://doi.org/10.1007/s10515-022-00343-z

MERIT: improving neural program synthesis by merging collective intelligence

Yating Zhang[1] · **Daiyan Wang**[1] · **Wei Dong**[1]

Received: 29 December 2020 / Accepted: 25 May 2022 / Published online: 22 June 2022

Abstract

Program synthesis is the task of automatically generating programs from user intent, which is one of the central problems in automated software engineering. Recently many researchers use a neural network to learn the distribution over programs based on user intent (such as API and type name), known as neural program synthesis (NPS). The generated programs of NPS are highly dependent on user intent. However, it is difficult for users to provide an accurate and complete intent for the NPS model, which decreases the synthesis accuracy of NPS. Collective Intelligence (CI) is an emerging trend, which illustrates that collective wisdom surpasses individual wisdom. Inspired by CI techniques, we propose an automatic task-specific user intent merging framework for NPS named MERIT (Merge User Intent of Program Synthesis). The key point of our framework is that we propose an improved Unsupervised Ant Colony Optimization (UACO) algorithm to selectively merge effective intent from multiple developers, and design three selection strategies to guide the merge process. The experiments show that our approach is able to provide more adequate and efficient input for NPS and improve the synthesis accuracy. Besides, our evaluation shows that selectively merging knowledge from multiple developers could be a significant way of promoting automated software engineering.

Keywords Neural program synthesis · Swarm intelligence · Collective intelligence · Pattern mining · User intent

✉ Yating Zhang
zhangyating18@nudt.edu.cn

Daiyan Wang
wangdaiyan@nudt.edu.cn

Wei Dong
wdong@nudt.edu.cn

[1] College of Computer, National University of Defense Technology, Changsha, China

1 Introduction

Program synthesis, the task of automatically generating programs from user intent (Gulwani et al. 2017), is considered as one of the central problems in automated software engineering. Recently, many studies solved the program synthesis problem on top of neural network architecture, known as Neural Program Synthesis (NPS). NPS tends to learn a conditional distribution that satisfies user intent over the high-level abstraction of programs. One challenge in NPS is the diversity of user intent since the efficiency and correctness of NPS are highly dependent on the accurate description of user intent.

User intent can be expressed in different forms, significant progress has been made in solving programming tasks from natural language descriptions, input–output examples, and small information desired by the target program. Programming by descriptions (Lin et al. 2018; Xu et al. 2017) is faced with a problem that is the accuracy of synthesis is limited by the ambiguity and hugeness of natural language. Programming by input–output examples (Balog et al. 2017; Gulwani et al. 2012; Devlin et al. 2017; Zohar and Wolf 2018) have to avoid the false positive answers by requiring elaborate effort to provide multiple corner cases of input–output pairs. Programming from small information (Feng et al. 2017; Shi et al. 2019; Murali et al. 2018) generates programs from the Application Programming Interfaces (APIs) or types, which also faces the limitation of incompleteness and inaccuracy of the intent. These factors make it clear that the expression of user intent has a close influence on the effectiveness of the synthesis models and systems while providing precise intent is not an easy task for some users who need to use synthesis tools to aid development.

To improve the automation and intelligence of software development, combining Collective Intelligence (CI) with software development is an emerging trend. CI is inspired by Swarm Intelligence (SI), which typically refers to the emergence of intelligent global behavior by interactions between individual agents in biological systems (Yudong et al. 2014). Studies of SI focus on biological systems, while CI concentrates mainly on humans, but they both demonstrate that collective wisdom surpasses individual wisdom. Recently, many advanced techniques based on CI have been proposed, including SwarmDebugging (Petrillo et al. 2019), PyReco (D'Souza et al. 2016), TopCoder (Lakhani et al. 2010), GitMerge,[1] and IntelliMerge (Shen et al. 2019). These techniques could significantly improve the efficiency of software development when the wisdom of multiple developers is collected and merged. Taking a cue from the traditional software development process of gathering and collating requirements before proceeding with the actual development, we exploit the advantages of CI by merging intentions from multiple developers to build a more adequate and accurate NPS input, thereby improving the accuracy of the NPS and enabling automated software code generation.

[1] https://git-scm.com/docs/git-merge.

Nevertheless, simply merging multiple knowledge cannot bring the best efficiency. We explore strategies for selectively merging user intent by applying the bio-inspired optimization algorithms in SI to NPS. The bio-inspired algorithms are designed from the studies on the collective behaviors in SI system, including Particle Swarm Optimization (PSO) (Kennedy and Eberhart 1995), Ant Colony Optimization (ACO) (Bonabeau et al. 1999), Bat Algorithm (BA) (Yang 2011) and many others (Yang et al. 2014; Fong et al. 2015). They have been proved to be efficient in feature selection in some areas of Artificial Intelligence, such as text clustering (Abualigah and Khader 2017), text classification (Kyaw and Limsiroratana 2019; Peng et al. 2018; Moslehi and Haeri 2020) and recommendation systems (Jain and Dixit 2019; Peška et al. 2019). However, these studies tend to find a fixed feature subset for the entire dataset, which is not suitable for the case where a specific feature subset needs to be selected for each task. Program synthesis certainly belongs to the latter problem. Therefore, to extract a specific set of features for each programming task, we also try to improve the bio-inspired algorithm to be more suitable for NPS in this paper.

Extracting more accurate labels for each task to represent the intent of the user is meaningful and also challenging. Meanwhile, with the unprecedented growth of online platforms and open source code, we also expect to combine the general knowledge on these bulky code resources to facilitate the development of program synthesis. We notice that some API methods are often called together in some programs, namely, the usage of API follows certain patterns. For example, when writing a file, we usually need to handle a file by opening, writing, and finally closing. Big data is usually the cornerstone of the probabilistic model and machine learning and machine learning technology develops rapidly consequently, many advanced techniques have emerged in API usage pattern mining, such as MAPO (Zhong et al. 2009), UPMiner (Wang et al. 2013) and PAM (Fowkes and Sutton 2016). Hence we integrate the frequency patterns information of APIs into the process of label extraction to express the intent more precisely.

Selectively merging information from CI produced by multiple developers is a significant way to construct adequate and accurate user intent for NPS. In this paper, we propose an automatic task-specific user intent merging method named MERIT combining both SI and CI to underpin NPS. It contains two meanings: one is to merge user intent to facilitate program synthesis, and the other is to merge the collective intelligence of developers from big code. We choose a bayesian encoder–decoder synthesis model BAYOU (Murali et al. 2018) as the synthesizer, which takes some labels (APIs and types possibly used in the target program) as input. Considering the ability of bio-inspired algorithms in SI to solve the optimization problem, we take the ACO algorithm to perform label selection. We divide our method into two stages: preparation for the label selection strategies and program synthesis based on Unsupervised ACO. In the preparation stage, we propose three label selection strategies. The ultimate goal of the three strategies is to combine big code knowledge and CI of multiple developers to improve the effectiveness of label selection in the ACO algorithm. The first strategy is the construction of a Label Movement Efficiency Matrix (LMEM), which deposits efficient movements constructed by the ants in more similar labels. We obtain this matrix by executing

an improved Supervised ACO (SACO) algorithm. To promote our second strategy, we mine API frequent usage patterns from the Internet code and build them into a Pattern Conditional Probability Matrix (PCPM) of labels to store general knowledge between the usage of the label. The last strategy is the merging of CI. Given the natural language description of a specific programming task, we merge intelligence or knowledge from multiple developers and construct the initial weighted label set by heuristically sorting these labels. After the preparation of three label selection strategies, our second state is the application of our novel Unsupervised ACO algorithm to program synthesis, where "Unsupervised" means that our algorithm does not depend on any specific NPS model. All three strategies are integrated into our UACO algorithm and we also propose a novel *n random proportional rule* to further improve our UACO algorithm. Our UACO algorithm is utilized to perform label selection for the NPS model, then programs are generated based on the selected label set.

We evaluated the performance of our improved ACO algorithms on 10,000 test data. Compared with program synthesis over no label selection, our algorithms increased the synthesis accuracy by 27%. In addition, our evaluation of three label selection strategies shows that either way of the above strategies can increase the efficiency of label selection. Moreover, we evaluated the influence of CI on NPS based on 25 real programming tasks. We simulated the scene of synthesizing programs on an isolated software development environment, and on a collaborative software development environment using our proposed method, the latter significantly increased the accuracy by 38% compared with the former. This evaluation of CI indicates that the cooperation and knowledge merging among isolated software developers is essential for program synthesis.

In summary, this paper makes the following contributions:

- We propose an automatic task-specific user intent merging method named MERIT for NPS. Our method can selectively merge user intent from the wisdom of multiple developers, which provides NPS models with more accurate and complete input.
- We propose an improved Unsupervised ACO algorithm, and design three label selection strategies to optimize the process of feature selection in NPS. Our novel algorithm could be a new way of feature engineering in the field of natural language processing.
- Inspired by the n-gram language model, we innovatively propose an *n random proportional rule* for our UACO algorithm.
- Our experiments on CI shows that code knowledge merging from multiple developers could be a significant way of promoting automated software development at a macro level.

The rest of this paper is organized as follows: we first discuss the related work in Sect. 2, then introduce the motivating example and fundamental knowledge in Sect. 3. Section 4 firstly presents the framework of our heuristic strategies and algorithmic frameworks, and then the detailed description is given in Sects. 5 and 6, respectively. The experimental results for each part are given in Sect. 7, and Sect. 8

discusses the limitations and threats of MeritIt. Finally, Sect. 9 gives our conclusions and discusses our future research.

2 Related work

2.1 Neural program synthesis

Neural Program Synthesis (NPS) typically applied the standard encoder–decoder framework to automatically generate code from a specific input (Ling et al. 2016), in particular, Seq2Seq architecture (Sutskever et al. 2014) is mostly the elemental model. One challenge the baseline model applied to NPS is that it had not considered the underlying syntax of the target programming language. To generate well-formed code, recently some researches predicted the grammar rule of the abstract syntax tree (AST) (Sun et al. 2019; Yin and Neubig 2017) or modeled the ASTs with a modular decoder in a top-down manner (Rabinovich et al. 2017). Another challenge is that these seq2seq models penalized many semantically correct programs satisfying the given input because they maximized the likelihood of only a single reference program. To address this limitation, *Bunel* (Bunel et al. 2018) and *Daniel* (Abolafia et al. 2018) performed reinforcement learning on top of a supervised model with the log-likelihood objective that maximizes the top-k semantically correct programs. Furthermore, considering the fact that RNN cannot capture a long sequence in some cases, *Sun* (Sun et al. 2019) proposed the grammar-based structural convolutional neural network for code generation, *Allamanis* represented source code with graphs (Allamanis et al. 2018) and applied gated graph neural network to model the source code graph (Brockschmidt et al. 2019).

Besides the researches on the improvement of the model architecture mentioned above, significant progress has been made in program synthesis from different kinds of specific inputs. Programming by example such as FlashFill (Gulwani et al. 2012), RobustFill (Devlin et al. 2017), and deepCoder (Balog et al. 2017), PCCCoder (Zohar and Wolf 2018), mainly generates code from given sample input–output pairs, these methods struggle with the expansion of the types of programs for they rely on a limited domain-specific language(DSL), and need a large engineering effort to provide multiple examples. Programming by description, such as NL2Bash (Lin et al. 2018), SQLNet (Xu et al. 2017), exists the problem in achieving high accuracy of synthesis programs caused by the ambiguity and hugeness of natural language. Component-based synthesis generates a program from a library of components, each component is a domain-specific function that could be used in the target program, existing work of this type contain SYPET (Feng et al. 2017) and FrAngel (Shi et al. 2019).

As mentioned earlier, the degree of accuracy with which users express their intents has a significant impact on the performance of the NPS model. The main scenario of our concern is those where it is difficult for the average or non-professional user to provide an absolutely complete and accurate user intent for the NPS, and this is where our work differs from the above. We design a user-friendly, automatically merged NPS extension framework that treats small information representing

user intent as features and uses the feature selection process to improve the accuracy of intent representation, thereby improving the performance of program synthesis.

2.2 Swarm intelligence

We focus on the application of swarm intelligence in artificial intelligence, especially the successful application of bio-inspired algorithms in feature selection (Brezočnik et al. 2018). *Laith* proposed a hybrid of PSO algorithm with genetic operators for the feature selection, their experiments show that the proposed algorithm encourages the k-means clustering algorithm to obtain more accurate clusters in text clustering (Abualigah and Khader 2017). In addition, great progress has been made in enabling bio-inspired algorithms to extract more informative and efficient features for text classification problems. Most researches obtained more accurate classification results by improving the algorithms such as PSO (Bai et al. 2018; Jain et al. 2019) and ACO (Kyaw and Limsiroratana 2019; Peng et al. 2018), or by proposing a hybrid of multiple algorithms such as a binary hybrid Grey Wolf Optimation and PSO algorithm (BGWOPSO) (Al-Tashi et al. 2019) and a hybrid Genetic Algorithm and PSO algorithm (GAPSO) (Moslehi and Haeri 2020). Moreover, bio-inspired algorithms have been widely used in the recommendation system (RS) (Jain and Dixit 2019), according to the comprehensive review of 77 research publications applying SI in RS (Peška et al. 2019).

Working on applying the bio-inspired algorithm to program synthesis differ from the above works in that, these studies tend to find a fixed optimal subset of features for the entire dataset, whereas the goal of program synthesis is to generate the corresponding code that satisfied the specific user intent. The native ACO algorithm is not fit for situations where a specific subset of features needs to be selected for each task, so we have modified and extended the ACO algorithm in the process of application with this in mind.

2.3 API usage pattern mining

The emergence and growth of big code resources on the Internet brings hope and inspiration to API usage pattern mining. MAPO (Zhong et al. 2009) is the first algorithm for mining API usage patterns, which applies a clustering on the extracted API sequence to mine API usage patterns. After that, an extension of MAPO is proposed named UP-Miner (Wang et al. 2013), in an attempt to effectively reduce redundancy and improve the succinctness of the mined API usage patterns. Others frequent pattern mining are the work by Acharya et al. (2007) and Buse and Weimer (2012). A major challenge in the above frequent pattern mining is the sheer size of its mining results (Han et al. 2007). In many cases, an enormous number of output patterns severely limit the usage of a frequent pattern miner. Researchers have proposed various techniques to reduce a large number of frequent patterns while maintaining the quality of identified patterns. For example, pCluster (Wang et al. 2002) is an algorithm to detect clusters of patterns. PAM (Fowkes and Sutton 2016) is a near

```
1  public String appendAllElemTostring(List<String>
       list) {{
2          ///call:forEach,append type:List,
               StringBuilder
3  }}
```

Fig. 1 A draft provided by 5th user in Table 1

parameter-free probabilistic API mining method, which largely avoids returning redundant and spurious sequences.

Our focus is on using pattern mining techniques to extract relationships between API sequence patterns, so we extracted the API sequences from the AST and mined them using PAM, and then learned the API usage pattern letters.

3 Motivation and preliminaries

In this section, we firstly illustrate our motivation through a programming task and then introduce the fundamental knowledge of the ACO algorithm on feature selection.

3.1 Motivating example

In this paper, we focus on generating programs conditioned on some labels that may contain small information such as API calls and types desired by the target program. Bayou (Murali et al. 2018) is a typical example of this case. The input of Bayou is a code draft such as Fig. 1, which represents the user intent by labels X (that consists of API calls and types) and a function signature, then generates a high-level abstraction of the program named *sketches* (that only contains APIs sequences and control structure), the *sketches* are further concretized into type-safe programs by adding variable usage and verifying whether the syntax is correct. Bayou utilizes a Bayesian encoder–decoder (BED) model to learn the distribution over *sketches* based on X.

The *sketches* synthesized by Bayou is strongly dependent on the labels provided in the draft. As an example, we have a programming task for *List* manipulation in *Java*, we require to append all elements in the *List* to a *String* variable. We asked five users to utilize Bayou to automatically synthesize the program of this task, the labels they provided are shown in Table 1 (User range from 1 to 5). However, no generated programs of all the five users satisfied our demand (we use 0 and 1 to denote whether the top-10 generated programs of Bayou satisfy our demand or not in Table 1. Figure 2 is the program synthesized based on the labels of user 1, which is the best result generated from five inputs. The code in Fig. 2 is finished with iterating over a *List* but failed to return a *String* containing all elements of the *List*. This example shows that people have trouble providing absolutely complete and accurate labels for a specific task, especially for those non-professionals.

Table 1 Labels provided by five users and corresponding synthesis results

User	API calls	Types	Result
1	*iterator, hasNext, next, join*	*ArrayList, Iterator, String*	0
2	*size, get, append, toString*	*ArrayList, StringBuilder*	0
3	*asList, next*	*Arrays, Iterator*	0
4	*hasNext, next, add, join*	*Iterator, ArrayList, String*	0
5	*forEach, append*	*List, StringBuilder*	0
6	API calls and types that simply merged from five users		0

```
1  public String appendAllElemTostring(List<String>
       list) {
2          boolean b1;
3          String s1;
4          ArrayList<String> al1;
5          boolean b2;
6          Iterator<String> i1;
7          i1 = (al1 = new ArrayList(list)).iterator();
8          while ((b1 = i1.hasNext())) {
9                  s1 = i1.next();
10                 b2 = s1.equals(list);
11         }
12         return s1;
13 }
```

Fig. 2 The best generated program of five users in Table 1

Considering that CI usually surpasses individual intelligence, we could obtain a more efficient label set satisfying our demand by merging all the incomplete knowledge from individuals. However, the result shows that simply merging all the labels (denoted by user 6 in Table 1) cannot synthesize the correct program, either. To address this problem, we propose a user intents merging method named MERIT, which can selectively merge efficient information from CI of multiple developers. Besides that, MERIT combines big code knowledge with SI algorithm, which integrates the frequent patterns of API usage mined from code knowledge into our UACO algorithm to improve the efficiency of ant movement. The details of MERIT will be shown in Sect. 4.

3.2 Ant colony optimization

ACO is inspired by the foraging behavior in ant colony systems on how to find the shortest path between their nest and food. Each ant explores a path individually and discards a chemical substance called pheromone during their search path, this allows the swarms to move to the optimal path via following the reinforced pheromone trail. ACO could be applied to find the best feature subset by

constructing a completed directed graph through n features (Peng et al. 2018). At each iteration of ACO, all ants individually construct a search path. Before the first iteration ($t = 1$), all the pheromone concentration $\tau_j(t)$ is initialized with the same value, where $\tau_j(t)$ is the pheromone concentration value on the feature node j at t-iteration. The detailed flow of each iteration is as follows:

Initialize population At the initial moment, the pheromone concentration $\tau(1)$ of each feature node is initialized to a fixed value and m ants are placed on randomly selected feature nodes.

Move ant Each ant applies a probabilistic action choice rule, called *random proportional rule*, to decide which node to visit next. In particular, the probability that ant k, currently at feature node i, chooses to go to feature node j is:

$$P_{ij}^{k}(t) = \frac{\tau_j(t)^{\alpha}\eta_{ij}(t)^{\beta}}{\sum_{s \notin N_k}\tau_s(t)^{\alpha}\eta_{is}(t)^{\beta}}, \ if\ j \notin N_k \tag{1}$$

where N_k is the set of feature nodes that ant k has not visited yet. α and β are the pheromone factor and heuristic factor, which are used to determine the relative influence of pheromone concentration and heuristic information, respectively. The heuristic item η_{ij} is generally set to the value $\frac{1}{d_{ij}}$, where d_{ij} is the Euclidean distance between feature nodes i and j.

Update phromone concentration After all the ants have constructed their search path at each iteration, the pheromone concentration on node j is updated:

$$\tau_j = (1-\rho)\tau_j + \sum_{k=1}^{m}\Delta\tau_j^k \tag{2}$$

where ρ is the evaporation rate to avoid the unlimited accumulation of the pheromone concentration, $\Delta\tau_j^k$ is the adding pheromone of node j on the path L that the ant k have crossed in their tours. The adding pheromone value $\Delta\tau_j^k$ is generally followed:

$$\Delta\tau_j^k = \begin{cases} \frac{Q}{L_k}, & if\ j \in L \\ 0, & else \end{cases} \tag{3}$$

where Q is a constant that is used to control the rate at which the pheromone concentration is increased, and L_k is the search path length of ant k.

Update global best solution The global best solution, the best feature subset, in this case, is updated at each iteration based on the *fitness function*, which is used to measure the quality of the selected subset and is usually defined as the accuracy of the target model. The algorithm finishes and returns the global best solution when t reaches the maximum number of iterations.

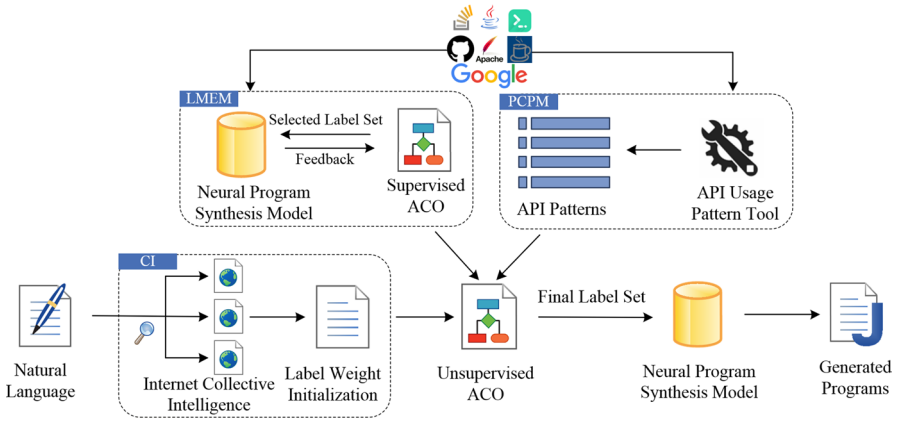


Fig. 3 The framework of MERIT, which merge the user intent of NPS by three strategies

4 Framework

The framework of MERIT proposed in this paper as shown in Fig. 3, basically includes two stages: (1) preparation for the label selection strategies; (2) program synthesis based on Unsupervised ACO.

Preparation stage We firstly propose three label selection strategies, as shown in the dotted box of Fig. 3. The ultimate goal of our three strategies is to combine big code knowledge and CI of multiple developers to improve the effectiveness of label selection in the ACO algorithm.

The first strategy constructs a label movement efficiency matrix (LMEM) based on an improved supervised ACO algorithm (SACO, Supervised ACO) in our previous work (Wang et al. 2020). In the primary ACO, the pheromone concentration between features is added dynamically and is not stored after the training, which is a difference from our work. We treat all labels in the code corpus as features and store the efficient moves by ants to mine for more similar labels in the big code. We perform label selection of a large *Java* corpus by using an improved Supervised ACO algorithm (SACO) with real-time feedback from the BED model of BAYOU. Ultimately, the two labels with bigger values of effective movement in LMEM imply a higher task similarity in the big data code.

The second strategy is to build a pattern conditional probability matrix (PCPM), in which each element represents the conditional probability of two labels in the API usage patterns. We have considered mining the patterns of API usage to shorten the synthesis path of reachable graph-based programs and accelerate synthesis speed (Liu et al. 2019). Far apart from before, we utilize conditional probabilities to represent relationships between the patterns of API usage, thus prompting the model to capture some API usage inertia and store it in the PCPM, which then allows the NPS decoder to consider the probabilities of API usage patterns during the inference, hence prompting more efficient synthesis. Specifically, we use a tool of API usage pattern mining to mine the patterns in the

code snippet of the same corpus and then calculate the probability of labels from the mined API usage patterns.

While the first two strategies effectively guide the label selection process by tapping into the knowledge of large code sources, the third strategy optimizes the label selection process by combining the intelligence of multiple developers from the CI perspective. Given only the natural language description of a programming task, we firstly merge labels crawled from web pages involving the CI of multiple developers. And then by heuristically weighting and sorting the mined labels, we construct the initial weighted label set. The third strategy provides the basis for establishing a user-friendly framework that automatically merges user intent based on natural language descriptions.

Application unsupervised ACO Our second stage is the application of our novel Unsupervised ACO (UACO) algorithm to program synthesis. "Unsupervised" in UACO means that our algorithm is independent of the NPS model, that is, it does not depend on the loss function, objective function, or other feedback from the NPS. Here we emphasize the difference and connection between UACO and SACO. SACO only passes the weight matrix of the labels to the NPS model, while the UACO model makes inferences together with NPS to generate the code, and the learning parameters of the two models are not shared.

The above three label selection strategies of the first stage eventually become the three heuristic information in the *random proportional rule* of our novel UACO algorithm, and the weighted label set constructed by the CI Strategy is the range of labels we can select from. Meanwhile, we propose a novel *n random proportional rule* to further improve the accuracy of our UACO algorithm. By combing three strategies and the *n random proportional rule*, our novel UACO algorithm is utilized to perform label selection. Finally, programs are generated based on the final label set.

5 Label selection strategies

In this section, we will illustrate our three label selection strategies in detail. Section 5.1 introduces the SACO algorithm and the construction of the LMEM. Section 5.2 introduces the process of API usage pattern mining and how to construct PCPM. Section 5.3 introduces how to efficiently organize labels from the Internet from the perspective of CI, and how to build our initial weighted label set.

5.1 Label moving efficiency matrix

We perform label selection on a large Java corpus based on our improved SACO algorithm and utilize feedback from the NPS model to gradually optimize the selected label subset. Furthermore, with the aim of taking advantage of the effective movements made by all the ants in the procedure of label selection, we construct an LMEM.

5.1.1 Supervised ACO algorithm

To be better suitable for the NPS model, label selection in the ACO algorithm takes the feedback from the NPS model into consideration. The quality of the label subset selected by the ACO is evaluated by the generated loss of the NPS model based on these selected labels. By gradually obtaining feedback from the NPS model, the ACO algorithm updates the pheromone concentration of label nodes and optimizes the selected label subset. We call our improved algorithm as SACO, the details of optimization are as follows:

Firstly, we modify the strategy of *Initialize Population*. Similar to the Round Robin algorithm, each ant is put on the different label node in turn at the initial time of each iteration rather than put randomly, to avoid bias towards some labels occasionally.

Secondly, to overcome the inadaptability of the Euclidean distance in the original *random proportional rule* to labels of NPS, we consider the cosine similarity of the labels in the entire corpus. the cosine similarity between label i and label j is calculated as follows:

$$Sim_{ij} = \frac{DocsVec_i DocsVec_j}{\left|DocsVec_i\right|\left|DocsVec_j\right|} \tag{4}$$

where $DocsVec$ is an N-dimensional vector, N is the data size of the corpus and each value in the vector represents the label frequency in the corresponding programming task. $|DocsVec|$ is the norm of vector. We take cosine similarity between two labels as the heuristic information, thus the *random proportional rule* is modified to:

$$P_{ij}^k(t) = \frac{\tau_j(t)^\alpha {Sim_{ij}}^\beta}{\sum_{s \notin N_k} \tau_s(t)^\alpha {Sim_{is}}^\beta}, \quad if\ j \notin N_k \tag{5}$$

Thirdly, we consider the feedback on the ACO algorithm from the BED model, the generated loss of model is used to measure the quality of the selected label set. Specifically, $\Delta\tau_j^k$ and *fitness function* are both defined as:

$$\Delta\tau_j^k = \frac{Q}{loss(J^k(t))}, \quad if\ j \in J^k(t) \tag{6}$$

$$fitness\ function = \frac{Q}{loss(J^k(t))} \tag{7}$$

where $J^k(t)$ is the label subset selected by ant k at iteration t. The reason of defines the Eq. 6 in this way, is that when the search path constructed by an ant makes the NPS model bring less loss, the label node j contained in this path will be updated with more pheromone concentration values. At the same time, the label subset that makes the model bring lower loss value has a higher *fitness function* value.

Algorithm 1 Improved Supervised Ant Colony Algorithm

Input: data set D, number of ants m, number of iterations t
Output: Label Move Efficiency Matrix (LMEM)

```
initialize LMEM.
for all (X, Y) ∈ D do
    initialize pheromone concentration.
    for i = 1 to t do
        initialize population
        for j = 1 to m do
            move ant j based on Equation 5.
        end for
        calculate fitness function of the label subset made by each ants based on
        Equation 7.
        sort and record top-d+1 search paths with the highest fitness function.
        update LMEM based on Equation 8.
        update pheromone concentration based on Equation 6 and 2.
    end for
end for
return LMEM
```

5.1.2 Label move efficiency matrix

By deeply and carefully researching our SACO algorithm, we analyzed the questions of (1) how to define the effective search paths and (2) how to take advantage of these efficient movements constructed by the ants. To answer the question (1), similar to the research of (Peng et al. 2018), we define both the optimal search path and top-d adjacent search path of the optimal path as the effective search paths. Namely, the top-$d+1$ label subset with the lowest generated loss in each iteration is finally considered. To answer question (2), a Label Moving Efficiency Matrix (LMEM) is created with the dimension $S \times S$, where S is the vocabulary size of APIs and types. With the aim to deposit effective movements between labels, and further find higher frequency movements from one label to another label, the LMEM can finally find two labels with higher similarity or frequently used together in the same code snippet. Each element in the LMEM is initialized with 0 and the moving efficiency from label i to label j, at iteration t, are updated as follows:

$$LMEM_{ij}(t) = LMEM_{ij}(t-1) + \Delta LMEM_{ij}^{bs}(t) + \sum_{near=1}^{d} \Delta LMEM_{ij}^{near}(t) \tag{8}$$

where bs, $near$ is the ant that made the best solution and the ants made the *top-d* best adjacent paths, respectively. $\Delta LMEM_{ij}^{bs}(t)$ and $\Delta LMEM_{ij}^{near}(t)$ are defined as follows:

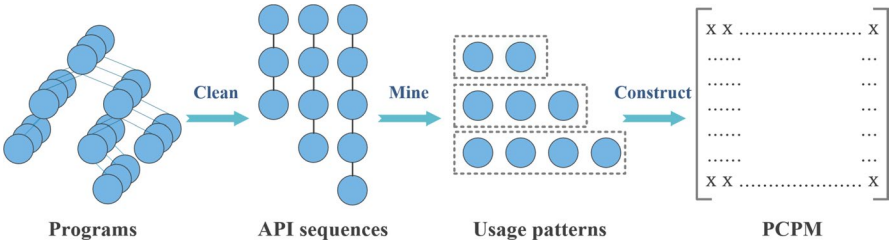


Fig. 4 The workflow of PCPM construction

$$\Delta LMEM_{ij}^{bs}(t) = \begin{cases} \frac{\mu}{loss(J^{bs}(t))}, & if\ i\ \in J^{bs}(t)\ and\ j \in J^{bs}(t) \\ 0, & else \end{cases} \tag{9}$$

$$\Delta LMEM_{ij}^{near}(t) = \begin{cases} \frac{1-\mu}{loss(J^{near}(t))}, & if\ i\ \in J^{near}(t)\ and\ j \in J^{near}(t) \\ 0, & else \end{cases} \tag{10}$$

where μ is the balance coefficient of nearest and optimal paths, generally set to 0.7. As shown in Eq. 8, the smaller the *loss* value, the larger the adding value to *LMEM*, and the more effective the movement from label i to label j. Actually, the above equations show that, in each iteration, when the optimal search path and the top-d adjacent search path are obtained, if the label i and j appear in the above-mentioned top-$d+1$ effective search paths at the same time, we hold the opinion that an effective movement is formed between these two labels. And then, we accumulate the moving efficiency $LMEM_{ij}(t)$ of the two labels according to the generated loss of the corresponding search path.

In summary, the workflow of our SACO algorithm is shown in Algorithm 1, which takes the data set D, the number of ants m, and the total number of iterations t as input. We first initialize the value of each element in LMEM to zero, and then perform label selection for each (X, Y) pair in the corpus. In each iteration, each ant constructs its search path according to Eq. 5. After all the ants have constructed their respective search paths, we use the fitness function of Eq. 7 to evaluate the label subset constructed by each ant, and record top-$d+1$ search paths with the highest *fitness function*. And then, we update the LMEM according to Eq. 8. Meanwhile, we update the pheromone concentration according to Eqs. 6 and 2. After all the data is traversed, the algorithm returns the heuristic LMEM.

5.2 Pattern conditional probability matrix

We collect the Java projects from Github and then use the API usage pattern mining tool to mine the patterns in the code snippets. After that, we construct a Pattern Conditional Probability Matrix (PCPM) for labels that extracts information and calculates probability from the mined API usage patterns. The workflow of constructing the PCPM is shown in Fig. 4. The first thing to build the PCPM is to collect a large number of code snippets. Afterward, the API sequences should be cleaned and

```
String match(String expr, String str, int gr) {
        String s1 = null;
        Pattern p1 = Pattern.compile(expr);
        Matcher m1 = p1.matcher(str);
        if(m1.find()){
                s1 = m1.group(gr);
        }
        return s1;
}
```

Fig. 5 A code snippet for API pattern mining

```
"java.util.regex.Pattern.compile",
"java.util.regex.Pattern.matcher",
"java.util.regex.Matcher.find",
"java.util.regex.Matcher.group"
```

Fig. 6 The extracted API sequence corresponding in Fig. 5

sorted out, then the patterns of API usage are mined by mining tools. Finally, the PCPM of labels is extracted and constructed according to patterns.

5.2.1 API usage pattern mining

In order to collect a huge amount of high-quality code snippets, we rank the Java projects on Github by stars, obtain projects on it, clean and collect the appropriate snippets. Considering the aim is that to synthesis a more generic application, we are no longer mining like previous classification pattern mining methods [such as mining by projects (Fowkes and Sutton 2016) and mining by application domain (Liu et al. 2019)], but rather mining all of the common and possibly cross-domain API usage patterns at once.

When got a large number of method-level code, we extract a sequence of API calls for each method, where we only consider method calls and object initialization. The complete API information is extracted based on the AST, which contains the fully qualified names from the top to the lowest layer packages, an example is shown in Fig. 6. When the sequences of API calls for each method are extracted, we mine the usage patterns of the API from those sequences. In this paper, we use an advanced API usage mining tool named PAM (Fowkes and Sutton 2016) to mine patterns. PAM is a near parameter-free probabilistic algorithm for mining the most interesting API patterns from a list of API call sequences. PAM largely avoids returning redundant and spurious sequences, unlike API mining approaches based on frequent pattern mining. Figure 5 is a method-level code, the API sequence corresponding in it is extracted as shown in Fig. 6.

5.2.2 Pattern conditional probability matrix construction

According to the excavated API usage patterns, we construct the PCPM of labels, which contains API usage information at the label level. The first step is to extract

the label from the API patterns. Here we qualify the label as the short name of the API method, and the type of the calling object. Besides, we remove the duplicate label. We then calculate the PCPM, which is a conditional probability matrix that describes the frequency relationship between every two labels. The dimension of PCPM is $K \times K$, where K represents the label number in all the API usage patterns. For the two labels i and j, we define the matrix element $PCPM_{ij}$ as follows:

$$PCPM_{ij} = P(j|i) = \frac{\sum_t Pattern_t(i) \cap Pattern_t(j)}{\sum_t Pattern_t(i)}, \quad t \in [1,\ T] \tag{11}$$

where T represents the number of patterns we mined, $Pattern_t(i)$ represents whether i appears in pattern $Pattern_t$. This formula represents the conditional probability relationship between two labels, that is, the probability that label j occurs when label i appears.

5.3 Collective intelligence

In order to address the problem of artificially providing absolutely complete and accurate labels for those NPS tools, we take CI of the extraordinary knowledge from the Internet into additional consideration. The main idea is that given only a natural language description of one programming task, we can search and merge labels from web pages involving CI of multiple developers and then obtain an initial weighted label set.

We used our API recommendation approach via searching on general search engines (Google Search) proposed in our previous study (Liu et al. 2020), called Args, to mine APIs and types from web pages. Args takes the natural language description of the programming task as input and returns a list of APIs. The web pages that Args retrieves include StackOverFlow, JavaDocs,[2] ProgramCreek[3] and Codota,[4] as they contain important programming information of Java. In this paper, we record the searching result containing both APIs and types from top-10 returned URLs. From the perspective of CI, we regarded each URL or web page as an individual agent of CI, under the hypothesis that each web page independently explores the solution to a given problem. After that, we further filter out irrelevant APIs and types depending on the Java Package name that each programming task belongs to. After that, all the labels that appeared in the 10 individual agents are weighted heuristically:

$$W_j = \frac{N_j}{M} \tag{12}$$

where N_j is the frequency of occurrence of label j in the 10 individual agents and M is the total number of agents (In case there are less than 10 web pages returned by

[2] https://docs.oracle.com/javase/8/docs/api/.
[3] https://www.programcreek.com.
[4] https://www.codota.com.

ARGS). Note that if one label appears more than one time on a web page, it will only be counted once on this web page. A higher weight value of a label indicates that most of the individual agents of CI tend to provide this label. At the end of this step, the top-15 labels with the highest weight constitute the initial label set.

6 Program synthesis based on unsupervised ACO

In this section, we will illustrate how to use the three label selection strategies mentioned in Sect. 5 (LMEM, PCPM, CI) to guide the label selection in the prediction phase of the NPS model. We introduce our Unsupervised ACO (UACO) algorithm on the prediction phase, which takes the weighted label set constructed by the CI strategy mentioned in Sect. 5.3 as input. Then, we optimize the *random proportional rule* of UACO by integrating three label selection strategies into it. Additionally, inspired by the *n*-gram language model, we proposed a novel *n random proportional rule* to further increase the accuracy of our UACO algorithm.

6.1 Improved unsupervised ACO algorithm

In the prediction phase of the NPS model, it should be emphasized that no feedback from the NPS model could be used to gradually guide the process of label selection in ACO, namely, the target *sketch Y* is invisible and unknown for the given label set *X*. To address this problem, we propose an improved Unsupervised ACO (UACO) algorithm, which takes the label selection strategies mentioned in Sect. 5 as three heuristic information in the *random proportional rule*: (1) LMEM that deposit effective movement between labels; (2) PCPM that mines frequent API usage patterns among big code data; (3) CI that merges intelligence and code knowledge of multiple developers. More specifically, “Unsupervised" in UACO have two meanings: (1) it makes inferences together with NPS to generate the code, the parameters between SACO and UACO are not shared, and SACO only pass the LMEM to the decoder of NPS; (2) it does not depend on the loss function, objective function, or other feedback from the NPS, that is, no use of the *fitness function* in Eq. 7. The details of the improvements are as follows:

(1) The initial pheromone concentration of each label j is modified to the weight of the label W_j obtained in Sect. 5.3:

$$\tau_j = W_j \tag{13}$$

(2) In Sect. 5.1, the heuristic information term in the *random proportional rule* is modified to Cosine Similarity. But in our UACO algorithm, we take the LMEM, the PCPM, and the weight of the labels into consideration. A higher value of $LMEM_{ij}$ means that a lot of effective movements are deposited between label i and label j. A higher value of $PCPM_{ij}$ means label i and label j are commonly used together in some frequency API usage patterns, and label j with a higher weight value W_j represents that most of the developers tend to provide the label

j when given a specific programming task. In summary, combing both the concentration information and three heuristic information, the *random proportional rule* is defined as:

$$P_{ij}^{k}(t) = \frac{\tau_j(t)^{\alpha} LMEM_{ij}{}^{\beta} PCPM_{ij}{}^{\omega} CI_j^{\gamma}}{\sum_{s \notin N_k} \tau_s(t)^{\alpha} LMEM_{is}{}^{\beta} PCPM_{is}{}^{\omega} CI_s^{\gamma}}, \quad if\ j \notin N_k \tag{14}$$

where CI_j is equal to W_j, β, ω, γ is the heuristic factor of LMEM, PCPM and CI respectively, their magnitudes modulate the influence level of each heuristic rule on label selection.

(3) As for the update rule of pheromone concentration τ_j at each iteration, from an unsupervised perspective, we define it as:

$$\tau_j = (1-\rho)\tau_j + \frac{A_j(t)}{B(t)} \tag{15}$$

where $A_j(t)$ is the total times of label j is selected by all ants at iteration t, and $B(t)$ is the total number of movements of all ants. The higher the value $\frac{A_j(t)}{B(t)}$, the larger the total number of times label j was selected by all ants, and the higher the adding pheromone concentration to label j.

Algorithm 2 Improved Unsupervised Ant Colony Algorithm

Input: initial weighted label set X, number of ants m, number of iterations t, $LMEM$, $PCPM$,numbers of label to be selected k

Output: selected label set X_s

```
initialize pheromone concentration based on Equation 13
for i = 1 to t do
    initialize population
    for j = 1 to m do
        move ant j based on Equation 14
    end for
    update pheromone concentration based on Equation 15
end for
sort X by the accumulative pheromone of each label.
return top-k label set X_s.
```

After the end of all iterations, we sort labels by the total accumulative pheromone concentration of each label and finally return top-k labels with the highest pheromone concentration, where k is the number of labels to be selected for a specific programming task. The workflow of UACO is shown in Algorithm 2. It takes the initial weighted label set X, number of ants m, number of iterations t, LMEM, PCPM, number of labels to be selected k as input, and then returns the selected label set.

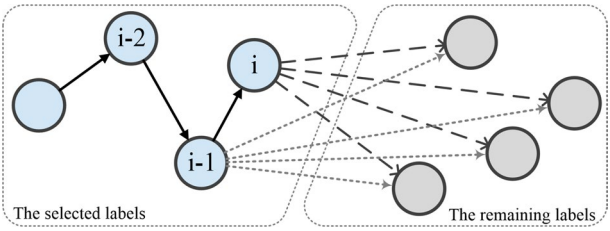


Fig. 7 The workflow of *n random proportional rule* when *n* is 2

6.2 N random proportional rule

In the field of natural language processing, the *n*-gram language model assumes that the appearance of the *i*th word is only related to the previous *n*-1 words. This idea can be applied to the label selection of the NPS model. In the current *random proportional rule*, the probability that label *j* is selected is only dependent on the last selected label *i*. However, when writing a line of code, we should consider both the declared variables and the API call in the previous lines. This means that the current *random proportional rule* does not consider the complete code context. In order to solve this problem, we propose a *n random proportional rule* based on multiple visited labels. This rule divides the total labels into two parts: a set of selected labels (as shown on the left in Fig. 7) and a set of labels to be selected (as shown on the right in Fig. 7). In our new rule, the label *j* to be selected is depend on the previous *n* visited labels(label *i*-*n*+1 to label *i*). The probability that label *j* is to be selected by ant *k* at iteration *t* is modified as follows:

$$P_t^k(j|i, i-1, \ldots, i-n+1) = \prod_{y=i}^{i-n+1} P_{yj}^k(t) \tag{16}$$

where $P_{yj}^k(t)$ is the probability with which ant *k*, currently at feature node *y*, chooses to go to feature node *j*, this is equal to Eq. 14. Figure 7 is the calculation diagram of *n random proportional rule* when *n* is 2. For each unvisited label on the right, their probability of being selected is determined by label *i*-1 and label *i* label in the selected set of labels.

7 Evaluation

In this section, we provide a comprehensive evaluation of our approach to addressing the following research questions:

RQ1: How does our approach perform compared to the baseline model? How much improvement does each of the two strategies from the general knowledge (LMEM and PCPM) of big data bring to the original method? Can two strategies capture useful label information and knowledge internally?

RQ2: How effective is the *n random proportional rule* in unsupervised ACO in boosting the model? How does the value of *n* affect performance, and what is the best value?

RQ3: How many contributions do the merging of task-specific knowledge derived from the developer (CI) to program synthesis in real programming tasks? How many differences between independent development, simple merging development, and selective merging for program synthesis?

7.1 Experiment setup

Data collection We constructed a new dataset from approximately 3000 high-star Java projects crawled from Github. By analyzing and cleaning the data, 466,129 method-level programs use *java.io*, *java.lang*, *java.util* libraries were extracted, which are three of the most common Java standard packages in Bayou model. After that, we pre-processed all method-level programs by parsing the code from Java to *sketch Y*, where each method-level code is considered as a programming task. To construct the initial label set *X* for each programming task, we first extracted labels from *sketches*, including API calls and types, and then added extra noise items randomly selected from all the labels appearing in the *sketches* yet. The noise items were used to better train our algorithms and to better evaluate our proposed strategies. The reason for this treatment is, on the one hand, to make the method generic and to build on top of the model training for program synthesis, so that only noise is added to the input. On the other hand, to verify the effectiveness of our UACO framework to select the right labels for program synthesis in the presence of interference information. Finally, the ratio of the correct items to noise items is 7:3 in each programming task. As a result, over 88,000 programs randomly selected from the top-1000 Java projects and 10,000 programs randomly selected from the top-1000 to top-3000 Java projects, constructed the training dataset and test dataset, respectively. The statistics in the test set are given in Fig. 8a, each column from left to right represents the information of the three labels in *X*, the last column indicates the sequence size of sketch, where there is no statistic of the control structure, but the sequence can also reflect the size of each task. As for the data preparation for API usage pattern mining, we selected the training dataset to mine API usage patterns.

Preparation for PCPM and LMEM In order to obtain the PCPM mentioned in Sect. 5.2, We extracted API sequences from the training set for API pattern mining and used the pattern mining tool PAM to mine for 1000 iterations.In order to obtain the PCPM mentioned in Sect. 5.2, we used the pattern mining tool PAM to mining for 1000 iterations. Secondly, to obtain the LMEM mentioned in Sect. 5.1, we used our SACO algorithm (Sect. 5.1) to extract 70% labels from the initial label set for each programming task in the training dataset. In this process, the feedback from BED model gradually guides the optimization of the selected label set. As for the hyper-parameter of SACO, the value of the initial pheromone concentration $\tau(0)$, the pheromone evaporation rate ρ, the pheromone factor α, the heuristic factor β, the number of ants m, the times of iterations t, the ratio of increased pheromone concentration Q, the number of adjacent search paths d and the ratio of nearest and optimal

	X_{call}	X_{type}	X_{key}	Y_{seq}
Min	0	2	3	1
Max	15	19	41	10
Median	1	5	10	2
Mean	2.09	5.10	11.22	1.99

(a) The extracted set of 10,000 tests from the top-1,000 to top-3,000 Java projects.

	X_{call}	X_{type}	X_{key}	Y_{seq}
Min	0	2	4	2
Max	15	19	41	10
Median	3	6	13	2
Mean	3.29	6.26	14.29	2.98

(b) The extracted set of 5049 tests from Fig.8(a), where the minimum length is greater than 1.

	X_{call}	X_{type}	X_{key}	Y_{seq}
Min	0	3	6	3
Max	15	19	41	10
Median	4	7	17	4
Mean	4.56	7.65	17.68	4.19

(c) [The extracted set of 2197 tests from Fig.8(a), where the minimum length is greater than 2.

	X_{call}	X_{type}	X_{key}	Y_{seq}
Min	0	2	3	1
Max	15	19	41	10
Median	3	5	12	2
Mean	2.54	5.53	12.40	2.36

(d) The extracted set of 3000 tests, where the half is randomly selected from Fig. 8(b) and another half is randomly selected from the remainder.

Fig. 8 Statistics on test sets which collected from the big code

paths μ were set to 0.2, 0.2, 1, 1, 30, 20, 0.1, 6, 0.7, respectively. These hyper-parameters were set partly by referring to others of the related collect intelligence research and partly selected by our experience.

Training preparation Note that the ultimate goal of label selection on the above training dataset with noise items is to deposit the effective movement of ants between similar labels, so as to construct our LMEM. Nevertheless, in the prediction phase of BED model, our UACO algorithm is independent of the model, that is, it does not rely on any feedback from the model. Therefore, toward providing a better model for the UACO in the prediction phase, we train the BED model on the above training dataset without noise items. The absence of noise under the training is to simulate the training situation of the original model and to ensure the independence of our approach. Besides, we have added attention mechanism to the origin BED models to improve the synthesis accuracy in our previous work (Zhang et al. 2020). The setting of other hyper-parameters in the BED models is the same as BAYOU.

Metrics To measure the program equivalence between the predicted results and the expected program, we use the same metrics proposed in BAYOU. They define the following metrics on the top-10 programs predicted by the model: (1) M1, a binary metric, measures whether the expected program appeared in a syntactically equivalent form in the results; (2) M2 and M3, measure the minimum *Jaccard distance* between the sequences of API calls and the sets of API calls in the expected and predicted programs, respectively; (3) M4 and M5, measure the minimum absolute difference between the number of statements and the number of control structures in the expected and sampled programs, respectively. To summarize, the higher the value of M1, and the lower the values of M2 to M5, the better the generated programs.

Table 2 Synthesis results of different models in 10,000 test data, to evaluate the effectiveness of LMEM and PCPM

Extraction method	M1	M2	M3	M4	M5	β	ω
Random	0.19	0.81	0.72	0.19	0.06	–	–
All	0.29	0.70	0.59	0.14	0.05	–	–
Uaco-Zero	0.21	0.79	0.70	0.19	0.06	0	0
Uaco-PCPM	0.40	0.59	0.46	0.13	0.05	0.25	0
Uaco-LMEM	0.55	**0.43**	0.29	0.12	**0.04**	0	1
Uaco-Mix	**0.56**	**0.43**	**0.28**	**0.11**	**0.04**	0.25	1

Bold values indicate the optimal values in the same column

7.2 Evaluation of label selection strategies of PCPM and LMEM (RQ1)

External analysis Our first experiment focuses more on the label selection strategies of both LMEM and PCPM in Sect. 5, because they are heuristic strategies prompted by large code data. It is conducted on 10,000 test data mentioned in Sect. 7.1, with noise labels in each programming task. Our UACO algorithm is used to extract 70% of labels from the initial label set of every task. In this experiment, we have not considered the CI strategy from the Internet, in that it needs a natural language description of each task to promote this crawler process, but these descriptions are missing in this 10,000 test data. Nevertheless, this allows us to better evaluate our LMEM and PCPM individually. We restrain the availability of LMEM and PCPM by controlling β and ω values in Eq. 14 (*random proportional rule*). Besides that, the initial pheromone concentration $\tau(0)$ of each label was set to a constant 0.2, the heuristic factor γ was set to 0, and other parameter settings were consistent with SACO (α, ρ, m, t).

We select three baselines and three comparison models to evaluate our strategies. Three baseline models are as follows: (1) randomly selecting the same amount of labels from the initial label set (denoted by Random); (2) selecting all the labels (denoted by All); (3) selecting the labels using our framework but without LMEM and PCPM (denoted by Uaco-Zero). The Uaco-Zero model shows the base result in our framework, the Random model simulates the result of random selection, and the All model represents the case that generating programs use not label extraction method. Three comparison models are as follows: (1) Uaco-PCPM, which only contains PCPM in the framework; (2) Uaco-LMEM, which only contains LMEM in the framework; (3) Uaco-Mix, which contains both PCPM and LMEM in the framework.

Table 2 shows the synthesis accuracy of different models, we bolded the best results of each metric. The comparison between Random and All in the baseline models reflects the conclusion in Bayou, that is, the performance improves with the increase of the number of labels, and the comparison between Uaco-Zero and Random proves that our swarm intelligence framework without the assistance of any label selection strategies is also better than random selection. The last four lines of models exhibit a similar trend, our Uaco-PCPM and Uaco-LMEM both improved over Random, which proved that PCPM and LMEM strategies are able to generate

better programs and extract more correct labels from the label set with noise items, respectively. Furthermore, our mixed model UACO-MIX improves by 37% in accuracy compared to the RANDOM model. It can be seen from the differentiation that UACO-MIX is slightly superior to UACO-LMEM. The result of our analysis is that LMEM vocabulary is much larger than PCPM, so the similarity of those label pairs that do not exist in PCPM is amplified by LMEM. Besides that, not every program will utilize API usage patterns. The two reason leads to a slight improvement in the performance of the mixing effect in the big data test.

Internal analysis For the PCPM, we mined API usage patterns on more than 88,000 programs with 1000 iterations. A total of 5639 patterns were mined. Table 3 shows some of the mined API patterns, and we present these API patterns by five domains. According to the mined API patterns, we constructed the PCPM. The total number of labels in the matrix is 1321, where labels consist of API calls and types. The matrix has 6857 non-empty terms. Figure 9a shows the top-10 labels that are most similar to the type *Iterator* in PCPM and the corresponding conditional probabilities. On the other hand, we obtained the colony moving LMEM after the combined training of BED with SACO described in Sect. 7.1. The number of labels contained in LMEM is 1422, which has 454,386 non-zero items. Figure 9b shows the top-20 labels that are most similar to the type *Iterator* in LMEM. The value on Fig. 9b represents the accumulative moving effective times that the ant moves from type *Iterator* to the next label, which reflects the effective frequency between two labels. We normalize it and then put it into the *n random proportional rule* in Eq. 14. As shown in Fig. 9, the LMEM contains some labels that are not in the API patterns, but the forefront labels in the PCPM keep the same trend in the LMEM.

We found that the range in LMEM is from 1e−9 to 1 and the range in PCPM is from 1e−3 to 1, therefore, we set β to 0.25 and ω to 1 is that we want to control two strategies with the matching influence, that is, the maximum and minimum range to be similar. In addition, the statistics show that the number of non-zero elements in LMEM is about 66 times higher than in PCPM. Because not every program uses the API usage patterns, which explains why PCPM has only a slight boost effect in 10,000 test data. In Sect. 7.4, some of our 25 real programming tasks contain API patterns, which experiments demonstrate the effectiveness of PCPM for label selection.

Answer to RQ1 The results show that our model improves the accuracy by 27% compared to the baseline model that provides all label information. The PCPM and LMEM improve by 11% and 26%, respectively, and both matrixes hold some label association knowledge from the visualization of the example.

7.3 Evaluation of *N random proportional rule* (RQ2)

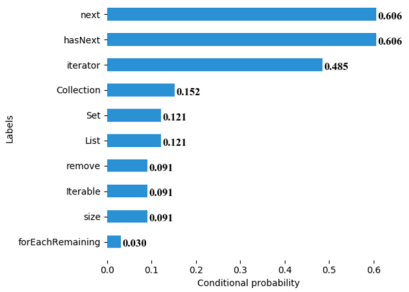
In order to evaluate the influence of different values of n in *n random proportional rule* mentioned in Sect. 6.2 on label selection, we extracted 3000 data from the 10,000 test dataset for experiments. Among the 3000 test data, 1500 test data have a sequence length greater than 2, in that a larger set of labels can better evaluate the influence of n. All the parameter of UACO is the same as model UACO-MIX in

Table 3 Some of the API usage patterns are mined by PAM

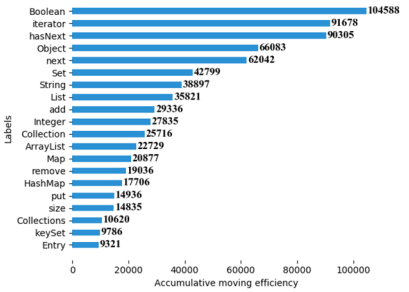
Domian	API usage pattern	ID
I/O	java.util.Scanner.Scanner	1
	java.util.Scanner.nextInt	
	java.io.PrintStream.println	
	java.io.FileInputStream.FileInputStream	2
	java.io.InputStreamReader.InputStreamReader	
	java.io.BufferedReader.BufferedReader	
	java.io.BufferedReader.readLine	
	java.io.BufferedReader.close	
	java.io.FileOutputStream.FileOutputStream	3
	java.io.FileOutputStream.write	
	java.io.FileOutputStream.close	
	java.io.File.exists	4
	java.io.File.mkdirs	
Regex	java.util.regex.Pattern.compile	5
	java.util.regex.Pattern.matcher	
	java.util.regex.Matcher.find	
	java.util.regex.Matcher.group	
	java.util.regex.Pattern.matcher	6
	java.util.regex.Matcher.replaceAll	
Collection	java.util.Map.containsKey	7
	java.util.Map.get	
	java.util.Map.put	
	java.lang.Iterable.iterator	8
	java.util.Iterator.hasNext	
	java.util.Iterator.next	
	java.util.Collection.stream	9
	java.util.stream.Stream.map	
	java.util.stream.Collectors.toList	
	java.util.stream.Stream.collect	
String	java.lang.StringBuilder.StringBuilder	10
	java.lang.StringBuilder.append	
	java.lang.StringBuilder.toString	
	java.util.StringTokenizer.StringTokenizer	11
	java.util.StringTokenizer.hasMoreTokens	
	java.util.StringTokenizer.nextToken	
	java.lang.String.lastIndexOf	12
	java.lang.String.length	
	java.lang.String.substring	
Date	java.util.Date.Date	13
	java.util.Date.getTime	
	java.util.Calendar.getInstance	14
	java.util.Calendar.setTime	
	java.util.Calendar.get	

Table 3 (continued)

Domian	API usage pattern	ID
Math	java.lang.Double.isNaN	15
	java.lang.Double.isInfinite	
	java.lang.Math.max	16
	java.util.Arrays.copyOf	

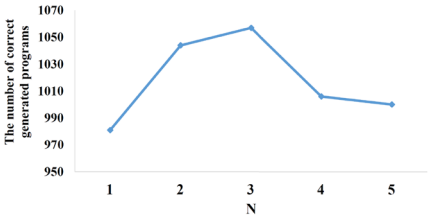


(a) The conditional probability of top-10 labels corresponding to *Iterator*.

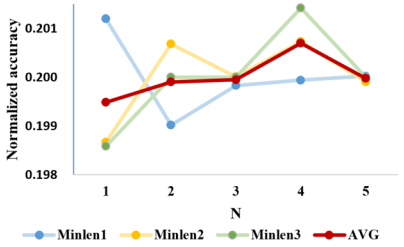


(b) Top-20 most similar labels of type *Iterator* according to the LMEM.

Fig. 9 A comparison example of type *Iterator* in PCPM and LMEM



(a) Fluctuating accuracy on different n values with the test set 3k(Fig. 8(d)).



(b) Fluctuating normalized accuracy on different n values with the sequence length L of test set.

Fig. 10 Influence of different n values on model accuracy

Sect. 7.2. In this experiment, we completed the label selection process by setting n to 1 to 5. When n ranges from 1 to 5, the number of correct generated programs are shown in Fig. 10a. The reason why we used the number of correct generated programs instead of the metrics mentioned above to evaluate the influence of n on the synthesis result is that the number of correct generated programs can more clearly reflect the influence of n on the synthesis result. It can be seen from Fig. 10a that when n ranges from 1 to 5, the number of correct generated programs firstly increases and then decreases. When n takes 3, the number of correct generated

programs reaches the highest value. This experiment shows that when the value of n in the *n random proportional rule* is 3, it will be most beneficial to our UACO algorithm to perform label selection.

The above experiments also imply some relevance between the value of n and the length of the sequence in the sketch. So as to study their specific relationship, we analyze the relationship between n and API sequence of length L, the experimental results are shown in Fig. 10b. We use the 10,000 test data collected in the Sect. 7.1 as the test set with the shortest length L of 1, and then filter the test sets with the shortest lengths of 2 and 3 respectively on this basis, and the statistics of these three data is shown in Fig 8. We perform three experiments on each data set separately and calculate the accuracy means. Because of the incoherence size on each test set, for comparison, we also normalize the accuracy at different models to facilitate the observation of the relationship between n and L. Finally, the average fluctuations over the three test sets were calculated as shown by the red line in Fig. 10b. The results appear that as L increases from 1 to 3, the optimal value of n rises from 1 to 4, indicating that the relationship between n and L is indeed a positive growth. Furthermore, The choice of n from 2 to 5 is better than 1 (without *n random proportional rule*), nevertheless, we finally choose n as 3 to control the complexity of computation and over-length dependence.

Answer to RQ2 Compared to the baseline, the *n random proportional rule* has some boosting effect, and n choices 2 to 5 all improve the accuracy, while there needs to be a balance between the boosting effect of n and the complexity of computation, and the result is 3.

7.4 Overall evaluation of UACO framework (RQ3)

In the above-mentioned experiments on big test data in Sect. 7.2 and 7.3, we lacked the evaluation of CI strategy mentioned in Sect. 5.3. Thus, in this experiment, we will first evaluate the efficiency of CI on 25 real programming tasks. And then, we performed an overall evaluation of our UACO framework mentioned in Sect. 6 by combing LMEM, PCPM, and CI strategies.

Evaluation of collective intelligence In this experiment, we simulated the scene of synthesizing programs on an isolated software development environment and on a collaborative software development environment, to evaluate how much support CI can provide for program synthesis. To promote this evaluation, we collected 25 real programming tasks from open-source repositories such as StackOverflow and ProgramCreek, involving six categories (manipulation of I/O, Math, Regex, Collection, and Date). Each task uses *java.io*, *java.util*, and *java.lang*, and has a corresponding natural language description and the number of labels to be selected. The detailed natural language description of each task is shown in Table 4. The statistics for the 25 tasks are shown in Fig. 11, which contains the number of types and API contained in each task, the sequence length, the selected value k of labels. The number of labels to be selected is pre-specified by professionals through the difficulty of the task, which is measured from the length of the task solution, the complexity of the

Table 4 Summary of the 25 real programming tasks

ID	Method name	Natural language description
1	insertChar	Insert the character to the specified position in the string
2	isBeginWith	Determines whether the string ends with a character and returns a substring of the string
3	readFile	Read a string from the specified file
4	listFiles	Determine if the input is a folder and return all the subfiles and subfolders of the folder
5	getSuffixName	Get the suffix name of the file string
6	judgeDir	Determine if a directory exists, or create a new directory if it doesn't
7	judgeFileExists	If the file does not exist, create a new file
8	getTimeOfTimeZone	Get the current time of a given time zone
9	addTime	Get the time after 7 days
10	appendAllELemTostr	Add all the string from the List collection to the a String variable
11	addNewStr	Add a new String to a List set
12	removeElem	Delete an element if it exists in the collection
13	visitAllElem	Iterate over all the elements in list and print it
14	visitAllElemMap	Get all the keys and values in map
15	breakingString	How to break a string into tokens
16	getNextint	How to get the next integer from the scanner
17	toBinaryString	Read an integer from the console and convert it to binary
18	getNextRandomInt	Use a random number seed to generate the next random number
19	getCurrentTime	How to get the current time
20	replaceStringWithRegex	Replace the string with regex expression
21	getABSValue	Generate a random number and return its absolute value
22	convertHexString	Convert hex string to float in Java
23	matchString	Compile the expression and find the matched string
24	reverseString	Reverse a string and append the result to it
25	readFile2Str	Return the content of the file

task across domains, and so on. The average number of labels to be selected for all tasks is 4.36, which accounts for about 77% of labels in each gold sketch and about 30% of all user-supplied labels.

In order to simulate these two environments, we defined each web page that we crawled labels from as an individual agent from the perspective of CI, under the hypothesis that each web page independently explores the solution to a given problem. We simulated the two environments of isolation and collaboration by controlling the number of individual agents of CI. In a collaborative environment, we considered 10 individual agents for each task at the same time and had knowledge fusion among them. While in an isolated environment, we only considered one individual agent at a time for each task and had no knowledge fusion. More specifically, to simulate the collaborative environment, we first constructed the initial weighted label set by merging the cooperative knowledge of 10 web pages

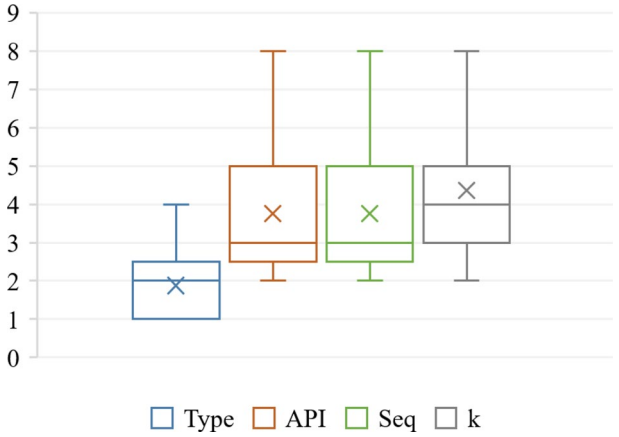


Fig. 11 Statistics on 25 programming tasks

Table 5 Synthesis accuracy of different models in 25 programming tasks, to evaluate the effectiveness of CI and heuristic information

Extraction Method	M1	M2	M3	M4	M5	β	ω	γ
CollectUaco	**0.56**	**0.44**	**0.36**	**0**	0.02	0.25	1	1
Random	0.03	0.96	0.72	0.05	0.11	–	–	–
All	0.04	0.96	0.65	0.21	0.09	–	–	–
IsolatedUaco	0.18	0.81	0.64	0.2	0.16	0.25	1	1
UACO-RemCI	0.44	0.56	0.46	**0**	0.02	0.25	1	0
UACO-RemLMEM	0.40	0.6	0.45	**0**	**0**	0	1	1
UACO-RemPCPM	0.44	0.56	0.43	0.01	0.03	0.25	0	1

Bold values indicate the optimal values in the same column

as mentioned in Sect. 5.3, and then UACO was applied to obtain the final label set, this method is denoted by CollectUaco. To simulate the isolated environment, for each task, we enabled each web page (individual agent of CI) to construct the initial label set individually based on its own knowledge, rather than merged labels from 10 web pages, afterwards, each individual agent performed UACO separately. Note that the weight value W of each label in UACO, in this case, represented the word frequency on a single web page. Finally, the metrics of each task is the average value of the 10 synthesis results based on these 10 individual agents. This method is denoted by IsolatedUaco.

The same parameter list of UACO is used in the above two experiments, and the total number of ants and iterations were both set to 10, ρ, α, β, γ, ω, were set to 0.2, 1, 0.25, 1, 1, respectively. The synthesis results are shown in Table 5. Compared with the IsolatedUaco, the CollectUaco significantly improved the accuracy of generating programs (M1) by 38%, which indicates that cooperation and knowledge fusion among isolated developers is essential for program synthesis.

```
1  public String appendAllElemTostring(List<String> list) {
2          boolean b1;
3          String s1;
4          StringBuilder sb1;
5          String s2;
6          Iterator<String> i1;
7          sb1 = new StringBuilder();
8          i1 = list.iterator();
9          while ((b1 = i1.hasNext())) {
10                 s1 = i1.next();
11                 sb1.append(s1);
12         }
13         s2 = sb1.toString();
14         return s2;
15 }
```

Fig. 12 Generated program from UACO-REMPCPM model

Overall evaluation of our UACO framework In fact, the experimental result of COLLECTUACO is the overall evaluation of our entire UACO framework. However, in this section, we will pay more attention to the impact of the lack of the three heuristic information(as mentioned in the *random proportional rule* in Eq. 14) on label selection and program synthesis. Specially, we separately evaluated the influence of the three heuristic information. All other details of UACO are consistent with COLLECTUACO, except for the heuristic information to be considered in the *random proportional rule* (Eq. 14). We evaluated the influence of CI (denoted by UACO-REMCI), LMEM (denoted by UACO-REMLMEM), and PCPM (denoted by UACO-REMPCPM) by setting the heuristic factor γ, β, ω to 0, respectively. The synthesis result is shown in the Table 5.

As can be seen from Table 5, the inexistence of LMEM (UACO-REMLMEM) and CI (UACO-REMCI) both reduce the accuracy of the synthesis results (M1) by 16% and 12% comparing with COLLECTUACO, respectively. To be more specific of our UACO algorithm and the influence of the LMEM and CI, we show the labels merging process of the motivation example based on strategy UACO-REMPCPM (which has a combined effect of both CI and LMEM). We use the 10th programming task (ID = 10) to illustrate the influence of LMEM on label selection, where the task can not be solved when the model is without LMEM. The initail weighted label set merged from 5 users in Sect. 3.1 are: *ArrayList* (0.6), *Iterator* (0.6), *next* (0.6), *append* (0.4), *join* (0.4), *hasNext* (0.4), *StringBuilder* (0.4), *String* (0.4), *size* (0.2), *get* (0.2), *toString* (0.2), *iterator* (0.2), *add* (0.2), *asList* (0.2), *forEach* (0.2). After performing UACO-REMPCPM, we obtained the top-7 labels: *Iterator*, *String*, *next*, *ArrayList*, *hasNext*, *iterator*, *StringBuilder*, which could synthesize the program(shown in Fig. 12) satifying our demand. This can be explained by Fig. 9b, which shows the top-20 most similar labels of type *Iterator* according to the accumulative moving efficiency in LMEM. Although the API call *iterator* has the lowest weight value (0.2) at the perspective of CI, it was finally selected by UACO-REMPCPM because of its high similarity with type *Iterator* (which has

Table 6 A comparison example between the CollectUaco and UACO-RemPCPM model, which reflect the final accumulative pheromone concentration of labels about the task 6 in Table 4

UACO-RemPCPM		CollectUaco	
Label name	Concentration	Label name	Concentration
File	2.134	*File*	1.999
exists	2.051	*exists*	1.684
get	0.245	*mkdirs*	**0.377**
mkdirs	**0.173**	*close*	0.367
mkdir	0.021	*write*	0.164
getAbsoluteFile	0.011	*get*	0.032
write	0.011	*mkdir*	0.021
close	0.011	*getAbsoluteFile*	0.011
getAbsolutePath	0.011	*getAbsolutePath*	0.011
isFile	0.011	*isFile*	0.011
isDirectory	0.011	*isDirectory*	0.011
FileWriter	0.011	*FileWriter*	0.011
BufferedWriter	0.011	*BufferedWriter*	0.011

Bold values indicate the concentration of mkdir in two models

the highest weight value (0.6)). This further reflects the balance of two heuristic information (LMEM and CI) in Eq. 14.

Meanwhile, compared with CollectUaco, the lack of PCPM (UACO-RemPCPM) also reduced the accuracy of synthesis results by 12%. We use the 6th programming task (ID = 6) to illustrate the influence of PCPM on label selection, where the model can solve this task when it contains PCPM. The purpose of the sixth programming task is to determine whether a directory exists, and create a new directory if it does not exist. Through label mining from web pages, the initial weighted label set we obtained are: *File* (0.5), *get* (0.3), *exists* (0.3), *mkdir* (0.2), *mkdirs* (0.2), *getAbsoluteFile* (0.1), *write* (0.1), *close* (0.1), *getAbsolutePath* (0.1), *isFile* (0.1), *isDirectory* (0.1), *FileWriter* (0.1), *BufferedWriter* (0.1), the weight value is marked in parentheses. By executing UACO-RemPCPM, the accumulative pheromone concentration of each label is shown in the left half of Table 6, where the labels in the table are arranged in descending order of the accumulative pheromone concentration. And we got the top-3 labels with the highest accumulative pheromone concentration: *File*, *exists*, *get*. The best-generated program based on these 3 labels is as shown in Fig. 13 on the left, which cannot satisfy our demand to create a new directory. But when we use CollectUaco to perform label selection, the top-3 labels we obtained are: *File*, *exists*, *mkdirs*, as shown in the right half of Table 6, which can generate the correct program shown in Fig. 13 on the right. Compared with UACO-RemPCPM, the concentration of *get* in the CollectUaco method dropped from 0.245 to 0.032, and then the concentration of *mkdirs* rose from 0.173 to 0.377, and the concentration of *mkdirs* rose to third place. The reason for this phenomenon is that our PCPM has mined frequent pattern between the API *exists* and API *mkdirs*, which can be seen in Table 3 at Pattern 4. And this pattern we mined increases the probability

```
public void judgeDir(String path) {
   boolean b1;
   File f1;
   String s1;
   f1 = new File(path, path);
   if ((b1 = f1.exists())) {
      s1 = f1.getAbsolutePath();
   }
   else {
   }
   return;
}
```

(1) UACO-REMPCPM

```
public void judgeDir(String path) {
   boolean b1;
   File f1;
   boolean b2;
   f1 = new File(path);
   if ((b1 = !f1.exists())) {
      b2 = f1.mkdirs();
   }
   else {
   }
   return;
}
```

(2) COLLECTUACO

Fig. 13 The best synthesis results between the COLLECTUACO and UACO-REMPCPM model about task 6

of *mkdirs* being selected when *exists* is selected by ants. In addition, by using the strategy COLLECTUACO instead of UACO-REMPCPM, API usage patterns 2, 12, and 11 we mined from big data, as shown in Table 3, played similar roles in the programming task 3, 5, and 15, respectively. Namely, our API usage pattern mining and PCPM help increase the correct rate of label selection of those programming synthesis tasks involving API usage patterns.

In summary, the results of this experiment show that the three heuristic information terms we proposed in this paper are both essential for the application of the ACO algorithm to program synthesis.

Answer to RQ3: The experimental results show that the task-specific knowledge from developers (CI) improves accuracy by 12%, while it also illustrates that selective merging of developer knowledge has a significant improvement over independent development and simple merging development.

8 Threats to validity

Generality of approach Due to the current state of research, we can only perform the evaluation of MERIT on the Bayou. However, this does not affect the generality of our approach, because the ACO strategy is to select the optimal subset of a feature set. Meanwhile, in the field of neural program synthesis, when the intent of the user is similar to token-level or vocab-level representation, each token is discrete, it is possible to extend and apply the model on the framework of our proposed method.

Degree of coverage A threat to the external validity of the results is the big data we collected from GitHub. We selected the high-star projects and filtered them by strategies, which did not cover all labels eventually. However, the current NPS approach cannot solve the problem outside of the vocabulary, even the large

pre-trained models are shielded from this problem by organizing a huge training set. In addition, the 25 programming tasks we chose were collected from multiple sources and manually filtered, which may introduce some subjectivity.

9 Conclusion

In this paper, we innovatively applied both the CI merged from multiple developers and a bio-inspired algorithm of SI to NPS, to address the problem of incompleteness and inaccurate user intents in NPS. As a result, an automatic task-specific user intent merging technique based on natural language description for NPS was proposed. We implement our approach on Bayou and propose an improved UACO algorithm to improve the accuracy of program synthesis from small information. Meanwhile, we propose three label selection strategies to optimize the process of label selection of our UACO algorithm. Our experiments show that our method is able to provide more adequate and efficient input for NPS and meanwhile, either way of these three label selection strategies can increase the synthesis accuracy. Besides, the extra evaluation of CI, which was conducted on 25 real programming tasks, shows that selectively knowledge merging among multiple developers in our approach could be a significant way of promoting automated software engineering at a macro level. More generally, our proposed user intent merging framework provides non-professionals a more convenient way to use these NPS methods by just taking natural language as input. Lastly, We also hold the opinion that our selective knowledge merging framework based on CI and SI could be a new way of feature engineering in the field of natural language processing.

We have demonstrated that our framework can help to improve the accuracy of label selection for program synthesis from small information. However, there are still more potentialities that can be dug. Firstly, our UACO algorithm returns a sorted set based on the pheromone concentrations of labels and is required to manually provide the number of labels to be selected (denoted by k) to determine the final subset. Nevertheless, this difficulty can be achieved by introducing additional test cases for the programming task, specifically, synthesis iteratively by increasing the value k until all the test cases are passed, this can be addressed in a manner similar to the forward search algorithm in feature selection. Secondly, our current considered labels only include API calls and data types. In fact, there are many other ways of expressing the user intent of a specific programming task, such as input–output examples, keywords in natural language descriptions, and so on. Thus, further work can be done to improve our algorithm to adapt to program synthesis from different kinds of user intents. Moreover, the API usage pattern mining tools we currently used are aimed at API sequences, and information of the input parameters types and return types that correspond to the related API is removed from the API sequences. However, this useful information could also be utilized to support pattern mining that contains both API calls and types, which can better assist our algorithm in label selection. All these aspects will be considered in our future work.

Acknowledgements This work was supported by National Natural Science Foundation of China (Nos. 61690203, 62032019).

References

Abolafia, D.A., Norouzi, M., Le, Q.V.: Neural program synthesis with priority queue training. CoRR abs arXiv:1801.03526 (2018)

Abualigah, L.M., Khader, A.T.: Unsupervised text feature selection technique based on hybrid particle swarm optimization algorithm with genetic operators for the text clustering. J. Supercomput. **73**(11), 4773–4795 (2017)

Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, 3–7 Sept 2007, pp. 25–34 (2007). https://doi.org/10.1145/1287624.1287630

Al-Tashi, Q., Kadir, S.J.A., Rais, H.M., Mirjalili, S., Alhussian, H.: Binary optimization using hybrid grey wolf optimization for feature selection. IEEE Access **7**, 39496–39508 (2019)

Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, 30 April–3 May 2018, Conference Track Proceedings, OpenReview.net (2018)

Bai, X., Gao, X., Xue, B.: Particle swarm optimization based two-stage feature selection in text mining. In: 2018 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. IEEE (2018)

Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: Deepcoder: Learning to write programs. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, 24–26 April 2017, Conference Track Proceedings, OpenReview.net (2017)

Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence—From Natural to Artificial Systems. Studies in the Sciences of Complexity, Oxford University Press, Oxford (1999)

Brezočnik, L., Fister, I., Podgorelec, V.: Swarm intelligence algorithms for feature selection: a review. Appl. Sci. **8**(9), 1521 (2018)

Brockschmidt, M., Allamanis, M., Gaunt, A.L., Polozov, O.: Generative code modeling with graphs. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, 6–9 May 2019, OpenReview.net (2019)

Bunel, R., Hausknecht, M.J., Devlin, J., Singh, R., Kohli, P.: Leveraging grammar and reinforcement learning for neural program synthesis. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, 30 April–3 May 2018, Conference Track Proceedings, OpenReview.net (2018)

Buse, R.P.L., Weimer, W.: Synthesizing API usage examples. In: 34th International Conference on Software Engineering, ICSE 2012, 2–9 June 2012, Zurich, Switzerland, pp. 782–792 (2012). https://doi.org/10.1109/ICSE.2012.6227140

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., Kohli, P.: Robustfill: neural program learning under noisy I/O. In: Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6–11 Aug 2017, PMLR, Proceedings of Machine Learning Research, vol. 70, pp. 990–998 (2017)

D'Souza, A.R., Yang, D., Lopes, C.V.: Collective intelligence for smarter API recommendations in python. In: 16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2016, Raleigh, NC, USA, 2–3 Oct 2016, pp. 51–60. IEEE Computer Society (2016). https://doi.org/10.1109/SCAM.2016.22

Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex apis. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 Jan 2017, pp. 599–612. ACM (2017)

Fong, S., Deb, S., Yang, X.S.: A heuristic optimization method inspired by wolf preying behavior. Neural Comput. Appl. **26**(7), 1725–1738 (2015)

Fowkes, J.M., Sutton, C.: Parameter-free probabilistic API mining across github. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium

on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, 13–18 Nov 2016, pp. 254–265. ACM (2016). https://doi.org/10.1145/2950290.2950319

Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Commun. ACM **55**(8), 97–105 (2012). https://doi.org/10.1145/2240236.2240260

Gulwani, S., Polozov, O., Singh, R., et al.: Program synthesis. Found. Trends® Program. Lang. **4**(1–2), 1–119 (2017)

Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. Data Min. Knowl Discov. **15**(1), 55–86 (2007). https://doi.org/10.1007/s10618-006-0059-1

Jain, D.K., Kumar, A., Sangwan, S.R., Nguyen, G.N., Tiwari, P.: A particle swarm optimized learning model of fault classification in web-apps. IEEE Access **7**, 18480–18489 (2019)

Jain, P., Dixit, V.S.: Recommendations with context aware framework using particle swarm optimization and unsupervised learning. J. Intell. Fuzzy Syst. **36**(5), 4479–4490 (2019)

Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of ICNN'95-International Conference on Neural Networks, vol. 4, pp. 1942–1948. IEEE (1995)

Kyaw, K.S., Limsiroratana, S.: Traditional and swarm intelligent based text feature selection for document classification. In: 2019 19th International Symposium on Communications and Information Technologies (ISCIT), pp. 226–231. IEEE (2019)

Lakhani, K.R., Garvin, D.A., Lonstein, E.: Topcoder (a): Developing software through crowdsourcing. Harvard Business School General Management Unit Case (610-032) (2010)

Lin, X.V., Wang, C., Zettlemoyer, L., Ernst, M.D.: Nl2bash: a corpus and semantic parser for natural language interface to the linux operating system. In: Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, 7–12 May 2018. European Language Resources Association (ELRA) (2018)

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K.M., Kociský, T., Wang, F., Senior, A.W.: Latent predictor networks for code generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, 7–12 Aug 2016, Berlin, Germany, vol. 1: Long Papers. The Association for Computer Linguistics (2016) https://doi.org/10.18653/v1/p16-1057

Liu, B., Dong, W., Zhang, Y.: Accelerating API-based program synthesis via API usage pattern mining. IEEE Access **7**, 159162–159176 (2019). https://doi.org/10.1109/ACCESS.2019.2950232

Liu, J., Liu, B., Dong, W., Zhang, Y., Wang, D.: How much support can API recommendation methods provide for component-based synthesis? In: 44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, 13–17 July 2020, pp. 872–881 (2020). https://doi.org/10.1109/COMPSAC48688.2020.0-155

Moslehi, F., Haeri, A.: A novel hybrid wrapper-filter approach based on genetic algorithm, particle swarm optimization for feature subset selection. J. Ambient. Intell. Humaniz. Comput. **11**(3), 1105–1127 (2020)

Murali, V., Qi, L., Chaudhuri, S., Jermaine, C.: Neural sketch learning for conditional program generation. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, 30 April–3 May 2018, Conference Track Proceedings, OpenReview.net (2018)

Peng, H., Ying, C., Tan, S., Hu, B., Sun, Z.: An improved feature selection algorithm based on ant colony optimization. IEEE Access **6**, 69203–69209 (2018)

Peška, L., Tashu, T.M., Horváth, T.: Swarm intelligence techniques in recommender systems—a review of recent research. Swarm Evol. Comput. **48**, 201–219 (2019)

Petrillo, F., Guéhéneuc, Y., Pimenta, M., Freitas, C.M.D.S., Khomh, F.: Swarm debugging: the collective intelligence on interactive debugging. J. Syst. Softw. **153**, 152–174 (2019). https://doi.org/10.1016/j.jss.2019.04.028

Rabinovich, M., Stern, M., Klein, D.: Abstract syntax networks for code generation and semantic parsing. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, 30 July–4 Aug, vol. 1: Long Papers, pp. 1139–1149. Association for Computational Linguistics (2017). https://doi.org/10.18653/v1/P17-1105

Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., Wang, Q.: Intellimerge: a refactoring-aware software merging technique. Proc. ACM Program. Lang. **3**(OOPSLA), 170:1-170:28 (2019). https://doi.org/10.1145/3360596

Shi, K., Steinhardt, J., Liang, P.: Frangel: component-based synthesis with control structures. Proc. ACM Program. Lang. **3**(POPL), 73:1-73:29 (2019). https://doi.org/10.1145/3290386

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., Zhang, L.: A grammar-based structural CNN decoder for code generation. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth

AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, 27 Jan–1 Feb 1 2019, pp. 7055–7062. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33017055

Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, 8–13 Dec 2014, Montreal, QC, Canada, pp. 3104–3112 (2014)

Wang, D., Dong, W., Zhang, Y.: Collective Intelligence for Smarter Neural Program Synthesis, pp. 98–104. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3417113.3423371

Wang, H., Wang, W., Yang, J., Yu, P.S.: (2002) Clustering by pattern similarity in large data sets. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, 3–6 June 2002, pp. 394–405. https://doi.org/10.1145/564691.564737

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D.: Mining succinct and high-coverage API usage patterns from source code. In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, 18–19 May 2013, pp. 319–328 (2013). https://doi.org/10.1109/MSR.2013.6624045

Xu, X., Liu, C., Song, D.: Sqlnet: generating structured queries from natural language without reinforcement learning. CoRR abs arXiv:1711.04436 (2017)

Yang, X.: Bat algorithm for multi-objective optimisation. Int. J. Bio Inspired Comput. **3**(5), 267–274 (2011). https://doi.org/10.1504/IJBIC.2011.042259

Yang, X.S., Karamanoglu, M., He, X.: Flower pollination algorithm: a novel approach for multiobjective optimization. Eng. Optim. **46**(9), 1222–1237 (2014)

Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, 30 July–4 Aug, vol. 1: Long Papers, pp. 440–450. Association for Computational Linguistics (2017). https://doi.org/10.18653/v1/P17-1041

Yudong, Z., Praveen, A., Vishal, B., Saeed, B., Xuewu, Z.: Swarm intelligence and its applications (2014). https://doi.org/10.1155/2014/204294

Zhang, Y., Dong, W., Wang, D., Liu, B., Liu, J.: Accuracy improvement for neural program synthesis via attention mechanism and program slicing. In: 44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, 13–17 July 2020, pp. 963–972. IEEE (2020). https://doi.org/10.1109/COMPSAC48688.2020.0-146

Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: mining and recommending API usage patterns. In: ECOOP 2009—Object-Oriented Programming, 23rd European Conference, Genoa, Italy, 6–10 July 2009. Proceedings, pp. 318–343 (2009). https://doi.org/10.1007/978-3-642-03013-0_15

Zohar, A., Wolf, L.: Automatic program synthesis of long programs with a learned garbage collector. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018. NeurIPS 2018, 3–8 Dec 2018, pp. 2098–2107. Montréal, Canada (2018)