Discovering Relationships between Data Structures and Algorithms

Guojin Zhu
Dept. of Computer Science, Donghua University, Shanghai 201620, China
Email: gjzhu.dhu@163.com

Zhiyue Yu and Jiyun Li
Dept. of Computer Science, Donghua University, Shanghai 201620, China
Email: {yzy8701@163.com, jyli@dhu.edu.cn}

***Abstract*—There are numerous of program code resources on the web which are solutions to programming problems on online judges. These program code resources are not organized for students to learn data structures and algorithms although they contain much knowledge of data structures and algorithms. For this reason, we propose an approach to organize the program code resources together with the programming problems systematically in terms of algorithms and data structures. This approach is based on the discovery of associate relationships between data structures and algorithms by applying ontology techniques. 1073 program codes on the web which are solutions to 480 problems distributed on online judges were mined in our experiment to discover the relationships between the data structures and algorithms used in the program codes. With the discovered relationships, the program codes and the corresponding problems were organized into learning materials in terms of algorithms and data structures. We believe that it would be useful for students to learn the programming knowledge.**

***Index Terms*—Data structure, Online Judge, Program code, Programming problem, Ontology**

I. Introduction

Nowadays, there are a lot of solution reports of programming problems on the web in the form of blogs written by programming contestants [1]. Most of these reports contain program source codes that can solve the programming problems on online judge (OJ) systems which are gathered from the ACM International Collegiate Programming Contest (ACM/ICPC) [2]. These program code resources contain lots of programming knowledge including data structures and algorithms. As we know, data structures and algorithms play important roles in programming. When students learn programming, they may not only want to learn the knowledge of data structures and algorithms, but also hope to find out suitable programming materials to learn how to implement data structures and algorithms in solving problems. However, the program code resources on the web are not organized in terms of data structures and algorithms. It is difficult for students to find suitable solution reports of programming problems which are focused on the knowledge points of algorithms and data structures that are just taught in their class.

Some approaches have been proposed to organize the program code resources and programming problems. One method is to discover knowledge units for programming tutoring based on Formal Concept Analysis (FCA) [3]. It organizes programming problems by analyzing the source codes submitted by students into a sequence of knowledge units, each of which consists of problems whose solutions need a common group of programming language points. It just deals with the programming knowledge at the level of a programming language, but does not analyze the knowledge of data structures and algorithms. This method is hardly applied to organizing the programming problems on the web into learning materials. Another method is to use a search engine to obtain solution reports of programming problems, and organize them on a basis of a predefined hierarchical body of programming knowledge [1]. Although this method could connect solution reports on the web together, it does not organize the program code resources in terms of algorithms and data structures systematically.

To address the issue above, we propose an approach to organize the program code resources together with the programming problems systematically in terms of algorithms and data structures. We use ontology techniques to recognize the data structures and algorithms used in program codes, and discover the relationships between them. With the discovered relationships, the program codes on the web and the corresponding problems on online judges are organized into learning materials in terms of algorithms and data structures. Furthermore, the programming problems are sorted from easy to difficult.

The main steps of our approach are shown in Fig. 1. First, we construct ontology individuals for program codes on a basis of a knowledge base. Then, we apply reasoning rules to the constructed ontology individuals for recognition of the data structures and algorithms used in the program codes. After that, we discover the relationships between data structures and algorithms by

Corresponding author: Guojin Zhu; Email: gjzhu.dhu@163.com.

doi:10.4304/jsw.8.7.1726-1735

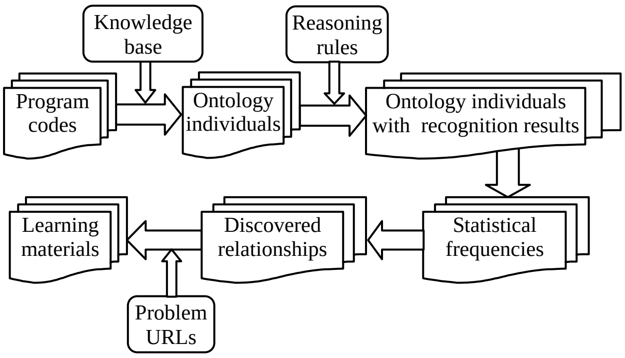

Figure 1. Main steps of our approach.

analyzing the statistical frequencies that each pair of a data structure and an algorithm is recognized appearing in the same program codes. And finally, with hyperlink techniques, we organize the program codes on the web and the corresponding problems on online judges into learning materials for students to learn the programming knowledge in terms of algorithms and data structures.

The rest of this paper is organized as follows. In Section II, we briefly introduce the concept of ontology and construct a knowledge base about data structures and algorithms. In Section III, we recognize the data structures and algorithms used in program codes, and discover the relationships between data structures and algorithms lying in program codes. In Section IV, we introduce the programming problems distributed on online judges and organize them into learning materials in terms of algorithms and data structures. Section V shows the experiment results and the hierarchical structure of the learning materials. Finally, we conclude our paper.

II. KNOWLEDGE BASE

A knowledge base [4] is a special kind of database for knowledge management. Knowledge representation is the core of a knowledge base. Since ontology provides a clear semantic and knowledge description of concepts and interrelation [5], the ontology representation is one of the common methods to represent knowledge.

In order to recognize data structures and algorithms used in program codes, we construct a knowledge base by ontology techniques. This knowledge base mainly contains two types of knowledge. One type is the descriptive knowledge about data structures and algorithms, and the other is the inferential knowledge in the form of reasoning rules which would be used for recognizing the data structures and algorithms contained in program codes.

A. Signals of Data Structures and Algorithms

We give a definition that the signals of data structures and algorithms are strings which can help us to recognize the data structures and algorithms used in program codes. For a data structure, a signal refers to a string that represents its name (or alias) or the name (or alias) of one of its typical operations. For example, the data structure *Stack* may have the following signals: its name or alias such as *mystack* or *mysta*, the name or alias of its pop operation like *pop* or *pop_stack* and the name or alias of its push operation like *push* or *push_stack*. For algorithms, we regard their names or aliases, which are usually used by programmers, as their signals. Take the algorithm *Depth First Search* as an example, its signals may be as following strings: *depth_first_search*, *depthfirstsearch*, *dfs*, *depthsearch*, *depth_search*, *depthfirst* and *depth_first*.

B. Ontology for the Knowledge Base

As a powerful tool, ontology has been widely applied in social science, medicine science and computer science [6]. In the field of computer science, the definition of ontology is that it is a clear formal specification of a shared conceptual model. This definition illustrates four characteristics of ontology [7]: clarification, conceptualization, formalization, and sharing. These features of ontology make it suitable for knowledge representation. The ontology representation [8] is one of commonly used methods to represent knowledge. Ontology can be applied to extract information from texts and documents [9-11]. It can also be used to retrieve information in some other fields, such as e-commerce [12] and crop diseases [13].

We design some classes and properties for the descriptive knowledge of the knowledge base, and design some reasoning rules for the inferential knowledge.

Now, we briefly introduce some ontology classes, some properties, and some individuals about data structures and algorithms. Their names and types are shown in TABLE I. The content of the column named *Name* are names of classes, properties and individuals, and the column named *Type* shows their types. In TABLE I, the names in the rows numbered 2, 4, 6, 8 and 11 are concerned with data structures, and they can be replaced by concrete names about their respective data structures. Similarly, the names in the rows numbered 3, 5, 7, 9 and 12 are concerned with algorithms, and they can be replaced by concrete names about their respective algorithms.

The domain and range of object properties and data properties are displayed in TABLE II. The domains of the object properties are the class *SourceCode*, their ranges are respectively the signal classes (e.g., *DS_Signal_Class*). The functions of these object

TABLE I.
CLASSES, PROPERTIES AND INDIVIDUALS

No.	Name	Type
1	*SourceCode*	Class
2	*DS_Signal_Class*	Class
3	*AL_Signal_Class*	Class
4	*has_ds_signal_object*	Object Property
5	*has_al_signal_object*	Object Property
6	*has_ds_signal_data*	Data Property
7	*has_al_signal_data*	Data Property
8	*has_ds_used_data*	Data Property
9	*has_al_used_data*	Data Property
10	*Code_individual*	Individual
11	*DS_Signal_Class_individual*	Individual
12	*AL_Signal_Class_individual*	Individual

TABLE II.
THE DOMAIN AND RANGE OF PROPERTIES

Property Name	Domain	Range
has_ds_signal_object	*SourceCode*	*DS_Signal_Class*
has_al_signal_object	*SourceCode*	*AL_Signal_Class*
has_ds_signal_data	*DS_Signal_Class*	*String*
has_al_signal_data	*AL_Signal_Class*	*String*
has_ds_used_data	*SourceCode*	*Boolean*
has_al_used_data	*SourceCode*	*Boolean*

properties are to mark whether or not a program code (indicated by an individual, e.g., *Code_individual*, of the class *SourceCode*) has contained some kinds of signals about the data structure *DS* and the algorithm *AL* (indicated by respective signal classes, e.g., *DS_Signal_Class*). The domains of the properties *has_ds_signal_data* and *has_al_signal_data* are their respective signal classes (e.g., *DS_Signal_Class*), and their range types are *String*. These data properties can store the specific signal strings which may appear in the program codes. The domains of the data properties *has_ds_used_data* and *has_al_used_data* are the class *SourceCode*, and their ranges are *Boolean*. The function of the data property *has_ds_used_data* is to indicate whether or not a program code uses the data structure *ds*. The function of the data property *has_al_used_data* is to indicate whether or not a program code uses the algorithm *al*.

C. Descriptive Knowledge about Algorithms

Here, we define some variables to denote the classes, properties and individuals about algorithms in the knowledge base as follows.

- A is the set of the algorithms in the knowledge base;
- A_i is an algorithm i in the set A, e.g., $A_i = DFS$.
- $Sa(i, j)$ is one of ontology classes indicated by *AL_Signal_Class* in TABLE I, which represents a kind j of signals of the algorithm A_i;
- $IAset(i, j, k)$ is one of ontology individuals indicated by *AL_Signal_Class_individual* in TABLE I, which is a member of the class $Sa(i, j)$, representing a signal k belonging to a kind j of signals of the algorithm A_i;
- $DPSa(i, j)$ is one of data properties indicated by *has_al_signal_data* in TABLE II, whose domain is the class $Sa(i, j)$ and whose range is a set of strings representing a kind j of signals of the algorithm A_i.

Take the algorithm *DFS* (*Depth First Search*) in Fig.2 for example, there is only one class *DepthFirstSearch* for its signals, i.e., $Sa(i, 1)$ = *DepthFirstSearch*. It has a data property *has_depth_first_search_signal*, i.e., $DPSa(i, 1)$ = *has_depth_first_search_signal*. Furthermore, there are seven individuals in the class $Sa(i, 1)$. The first individual is $IAset(i, 1, 1)$ = *DepthFirstSearch_individual_1*, whose property value is “dfs”. The last individual is $IAset(i, 1, 7)$ = *DepthFirstSearch_individual_7* , whose property value is “depth_first_search”.

```
<owlx:Class owlx:name="#SourceCode"/>
<owlx:Class owlx:name="#DepthFirstSearch"/>
<owlx:DatatypeProperty
        owlx:name="#has_depth_first_search_signal">
   <owlx:domain owlx:class="#DepthFirstSearch"/>
</owlx:DatatypeProperty>
<owlx:Individual owlx:name="#DepthFirstSearch_individual_1">
     <owlx:type owlx:name="#DepthFirstSearch"/>
     <owlx:DataPropertyValue
             owlx:property="#has_depth_first_search_signal">
        <owlx:DataValue
       owlx:datatype="&xsd;string">dfs</owlx:DataValue>
   </owlx:DataPropertyValue>
</owlx:Individual>
…
<owlx:Individual owlx:name="#DepthFirstSearch_individual_7">
     <owlx:type owlx:name="#DepthFirstSearch"/>
     <owlx:DataPropertyValue
             owlx:property="#has_depth_first_search_signal">
        <owlx:DataValue  owlx:datatype="&xsd;string">
       depth_first_search </owlx:DataValue>
   </owlx:DataPropertyValue>
</owlx:Individual>
```

Figure 2. The descriptive knowledge about the algorithm *DFS*.

D. Descriptive Knowledge about Data Structues

The variables needed to denote the classes, properties and individuals about data structures in the knowledge base are as follows.

- D is the set of the data structures in the knowledge base;
- D_i is a data structure i in the set D, e.g., $D_i = Stack$;
- $Sd(i, j)$ is one of ontology classes indicated by *DS_Signal_Class* in TABLE I, which represents a kind j of signals of the data structure D_i;
- $IDset(i, j, k)$ is one of ontology individuals indicated by *DS_Signal_Class_individual* in TABLE I, which is a member of the class $Sd(i, j)$, representing a signal k belonging to a kind j of signals of the data structure D_i;
- $DPSd(i, j)$ is one of data properties indicated by *has_ds_signal_data* in TABLE II, whose domain is the class $Sd(i, j)$ and whose range is a set of strings representing the kind j of signals of the data structure D_i.

Take the data structure *Stack* in Fig.3 for example, there are three classes *StackVariable*, *StackPopOperation* and *StackPushOperation* for its three kinds of signals, i.e., $Sd(i, 1)$ = *StackVariable*, $Sd(i, 2)$ = *StackPopOperation*, and $Sd(i, 3)$ = *StackPushOperation*. The class $Sd(i, 1)$ has a data property *has_stack_variable_signal*, i.e., $DPSd(i, 1)$ = *has_stack_variable_signal*. There are two individuals in the class $Sd(i, 1)$. The first individual is $IDset(i, 1, 1)$ = *StackVariable_individual_1*, whose property value is “stack”. The second individual is $IDset(i, 1, 2)$ = *StackVariable_individual_2*, whose property value is “mysta”. The class $Sd(i, 2)$ has a data property *has_stack_pop_operation_signal*, i.e., $DPSd(i, 2)$ = *has_stack_pop_operation_signal*. There are two individuals in the class $Sd(i, 2)$. The first individual is $IDset(i, 2, 1)$ = *StackPopOperation_individual_1*, whose property value is “pop”. The second individual is $IDset(i, 2, 2)$ = *StackPopOperation_individual_2*, whose property value is “pop_stack”. The class $Sd(i, 3)$ has a data property *has_stack_push_operation_signal*, i.e., $DPSd(i,$

3) = *has_stack_push_operation_signal*. There are two individuals in the class *Sd*(*i*, 3). One is *IDset*(*i*, 3, 1) = *StackPushOperation_individual_1*, whose property value is “push”; whereas the other is *IDset*(*i*, 3, 2) = *StackPushOperation_individual_2*, whose property value is “push_stack”.

```
<owlx:Class owlx:name="#SourceCode"/>
<owlx:Class owlx:name="#StackVariable"/>
<owlx:Class owlx:name="#StackPopOperation"/>
<owlx:Class owlx:name="# StackPushOperation ">
<owlx:DatatypeProperty owlx:name="#has_stack_variable_signal">
   <owlx:domain owlx:class="#StackVariable"/>
</owlx:DatatypeProperty>
<owlx:Individual owlx:name="#StackVariable_individual_1">
      <owlx:type owlx:name="#StackVariable"/>
      <owlx:DataPropertyValue
              owlx:property="#has_stack_variable_signal">
          <owlx:DataValue
         owlx:datatype="&xsd;string">stack</owlx:DataValue>
    </owlx:DataPropertyValue>
</owlx:Individual>
<owlx:Individual owlx:name="#StackVariable_individual_2">
      <owlx:type owlx:name="#StackVariable"/>
      <owlx:DataPropertyValue
              owlx:property="#has_stack_variable_signal">
      <owlx:DataValue
              owlx:datatype="&xsd;string">mysta</owlx:DataValue>
    </owlx:DataPropertyValue>
</owlx:Individual>
<owlx:DatatypeProperty
         owlx:name="#has_stack_pop_operation_signal">
   <owlx:domain owlx:class="#StackPopOperation"/>
</owlx:DatatypeProperty>
<owlx:Individual owlx:name="#StackPopOperation_individual_1">
  <owlx:type owlx:name="#StackPopOperation"/>
    <owlx:DataPropertyValue
  owlx:property="#has_stack_pop_operation_signal">
          <owlx:DataValue
         owlx:datatype="&xsd;string">pop</owlx:DataValue>
    </owlx:DataPropertyValue>
</owlx:Individual>
<owlx:Individual owlx:name="#StackPopOperation_individual_2">
   <owlx:type owlx:name="#StackPopOperation"/>
    <owlx:DataPropertyValue
  owlx:property="#has_stack_pop_operation_signal">
    <owlx:DataValue
       owlx:datatype="&xsd;string">pop_stack</owlx:DataValue>
    </owlx:DataPropertyValue>
</owlx:Individual>
<owlx:DatatypeProperty
         owlx:name="#has_stack_push_operation_signal">
   <owlx:domain owlx:class="#StackPushOperation"/>
</owlx:DatatypeProperty>
<owlx:Individual owlx:name="#StackPushOperation_individual_1">
      <owlx:type owlx:name="#StackPushOperation"/>
      <owlx:DataPropertyValue
              owlx:property="#has_stack_push_operation_signal">
          <owlx:DataValue
         owlx:datatype="&xsd;string">push</owlx:DataValue>
    </owlx:DataPropertyValue>
</owlx:Individual>
<owlx:Individual owlx:name="#StackPushOperation_individual_2">
<owlx:type owlx:name="#StackPushOperation"/>
  <owlx:DataPropertyValue
  owlx:property="#has_stack_push_operation_signal">
    <owlx:DataValue
       owlx:datatype="&xsd;string">push_stack</owlx:DataValue>
    </owlx:DataPropertyValue>
</owlx:Individual>
```

Figure 3. The descriptive knowledge about the data structure *Stack*.

E. Descriptive Knowledge about Program Codes

We use *SC* to represent the ontology class *SourceCode*, which is the domain of the four kinds of properties indicated by *has_al_signal_object*, *has_ds_signal_object*, *has_al_used_data* and *has_ds_used_data* in TABLE II. We use the following variables to denote these properties.

- *OPa*(*i*, *j*) is one of object properties indicated by *has_al_signal_object*, whose range is the class *Sa*(*i*, *j*);
- *OPd*(*i*, *j*) is one of object properties indicated by *has_ds_signal_object*, whose range is the class *Sd*(*i*, *j*);
- *DPUa*(*i*) is one of data properties indicated *by has_al_used_data*, whose range is boolean (*true* or *false*);
- *DPUd*(*i*) is one of data properties indicated *by has_ds_used_data*, whose range is boolean (*true* or *false*).

Fig.4 depicts the class *SC* (i.e., *SourceCode*) and its properties about the algorithm *DFS* and the data structure *Stack*. The class *SC* has an object property *OPa*(*i*, 1) = *has_depth_first_search*, which corresponds to the only one kind of signals about the algorithm *DFS*. Furthermore, it has another three object properties *has_stack_variable*, *has_stack_pop_operation* and *has_stack_push_operation*, which correspond to the three kinds of signals of the stack, respectively, i.e., *OPd*(*i*, 1) = *has_stack_variable*, *OPd*(*i*, 2) = *has_stack_pop_operation* and *OPd*(*i*, 3) = *has_stack_push_operation*. Finally, the class *SC* has two data properties, i.e., *DPUa*(*i*) = *has_dfs_used* and *DPUd*(*i*) = *has_stack_used*.

```
<owlx:Class owlx:name="#SourceCode"/>
<owlx:ObjectProperty owlx:name="#has_depth_first_search">
   <owlx:domain owlx:class="#SourceCode"/>
   <owlx:range owlx:class="#DepthFirstSearch"/>
</owlx:ObjectProperty>
<owlx:DatatypeProperty owlx:name="#has_dfs_used">
   <owlx:domain owlx:class="#SourceCode"/>
</owlx:DatatypeProperty>
<owlx:ObjectProperty owlx:name="#has_stack_variable">
   <owlx:domain owlx:class="#SourceCode"/>
   <owlx:range owlx:class="#StackVariable"/>
</owlx:ObjectProperty>
<owlx:ObjectProperty owlx:name="#has_stack_pop_operation">
   <owlx:domain owlx:class="#SourceCode"/>
   <owlx:range owlx:class="#StackPopOperation"/>
</owlx:ObjectProperty>
<owlx:ObjectProperty owlx:name="#has_stack_push_operation">
   <owlx:domain owlx:class="#SourceCode"/>
   <owlx:range owlx:class="#StackPushOperation"/>
</owlx:ObjectProperty>
<owlx:DatatypeProperty owlx:name="#has_stack_used">
   <owlx:domain owlx:class="#SourceCode"/>
</owlx:DatatypeProperty>
```

Figure 4. Some properties of the class *SourceCode*.

The procedure to create classes and properties is described briefly as follows.

First, we create the unified ontology class *SC*, and its properties *DPUd*(*i*) and *DPUa*(*i*).

Then, for each kind *j* of signals of each data structure *i* we build a class *Sd*(*i*, *j*) and its property *DPSd*(*i*, *j*), and the property *OPd*(*i*, *j*) of the class *SC*. For each kind *j* of signals of each algorithm *i* we also build a class *Sa*(*i*, *j*) and its property *DPSa*(i, j), and the property *OPa*(i, j) of the class *SC*.

Finally, for each signal k belonging to the kind j of signals of the data structure i we create a member *IDset*(i, j, k) of the class *Sd*(i, j), whose data property *DPSd*(i, j) is assigned with the string of the signal k. Similarly, we create a member *IAset*(i, j, k) of a class *Sa*(i, j) for each signal k of the algorithm i, whose data property *DPSa*(i, j) is assigned with the string of the signal k.

We regard these classes, properties and individuals as the descriptive knowledge in the knowledge base.

F. Inferential Knowledge for Reasoning Rules

The inferential knowledge in the knowledge base refers to the rules to recognize data structures or algorithms used in program codes. A rule usually consists of a conclusion and several preconditions. The conclusion is the reasoning result that we want to obtain, and the preconditions indicate the circumstances under which the conclusion is tenable. The conclusion is associated with one of properties indicated by *has_al_used_data* or *has_ds_used_data* in TABLE II, which would appear in the query for recognizing the corresponding data structure or algorithm. Each precondition is also associated with a respective property.

Fig.5 shows the reasoning rule of the data structure *Stack*. In this rule, there are four properties marked by the tag <swrlx:individualPropertyAtom>. The property named *has_stack_used* between the tag <ruleml:_head> and the tag </ruleml:_head> is the conclusion of the rule. It has two variables with the same name *CODE*. Its meaning is to check up whether or not an individual (indicated by *CODE*) of the class *SC*, which represents a program code, satisfies the following preconditions. If it satisfies all the preconditions, the individual of the class *SC* will be added to the result set of the rule. The properties between the tag <ruleml:_body> and the tag </ruleml:_body> are three preconditions of the rule. Taking the property named *has_stack_variable* for example, it possesses two variables named *CODE* (which is the same as the variable *CODE* appearing in the conclusion) and *STACKVAR*. It means that when the conclusion is tenable, the individual (indicated by *CODE*) of the class *SC* must have an object property *has_stack_variable*. When the individual indicated by *CODE* possesses the three object properties *has_stack_variable*, *has_stack_pop_operation* and *has_stack_push_operation* (which are part of knowledge depicted in Fig.4), this rule infers that the program code represented by the individual indicated by *CODE* use the data structure *Stack*.

```
<ruleml:imp>
  <ruleml:_head>
    <swrlx:individualPropertyAtom swrlx:property="#has_stack_used">
      <ruleml:var>CODE</ruleml:var>
       <ruleml:var>CODE</ruleml:var>
      </swrlx:individualPropertyAtom>
  </ruleml:_head>
  <ruleml:_body>
    <swrlx:individualPropertyAtom
               swrlx:property="#has_stack_variable">
      <ruleml:var>CODE</ruleml:var>
      <ruleml:var>STACKVAR</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom
               swrlx:property="#has_stack_pop_operation">
      <ruleml:var>CODE</ruleml:var>
      <ruleml:var>POP</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom
                swrlx:property="#has_stack_push_operation">
       <ruleml:var>CODE</ruleml:var>
       <ruleml:var>PUSH</ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:_body>
</ruleml:imp>
```

Figure 5. The reasoning rule of the data structure *Stack*.

We regard the reasoning rules as the inferential knowledge of the knowledge base. For each data structure or each algorithm, there is a unique rule associated with it in the knowledge base. Thus, each rule represents a data structure or an algorithm in the knowledge base. The inferential knowledge in the knowledge base would be used to recognize the data structures or algorithms lying in program codes.

III. RELATIONSHIP DISCOVERY

A. Ontology Individuals for Program Codes

After building the knowledge base, we could create individuals of the class *SC* for program codes, and attach to the created individuals some properties if the program codes contain some corresponding signals about data structures or algorithms in the knowledge base.

We propose an algorithm *CreateIndividual*, which is used to create an individual for a program code. The following variables are needed in this algorithm.

- C is the program code;
- *Ind*(C) is the individual created for the program code C.

The algorithm *CreateIndividual* is described as follows:

1) Read the content of a program code C;
2) Create an ontology individual *Ind*(C) of the class *SC* for the program code C (In this step, the individual *Ind*(C) does not possess any property);
3) For each individual *IDset*(i, j, k) in each signal class *Sd*(i, j) of each data structure D_i, get the signal string k by reading the value of its data property *DPSd* (i, j), and attach the individual *IDSet*(i, j, k) to the code individual *Ind*(C) via the object property *OPd* (i, j) if the program code C contains the signal string k;
4) For each individual *IAset*(i, j, k) in each signal class *Sa*(i, j) of each algorithm A_i, get the signal string k by reading the value of its data property *DPSa* (i, j), and attach the individual *IASet*(i, j, k) to the code individual *Ind*(C) via the object property *OPa* (i, j) if the program code C contains the signal string k.

Fig.6 shows a program code which contains signals of three kinds of the data structure *Stack*, and the signal string “dfs” of the algorithm *DFS*. By using the algorithm *CreateIndividual*, we create a code individual named *ProgramCode_1* for this program code as shown in Fig.7.

```
#include<stdio.h>
#include<stack>
#define maxn 50
using namespace std;
stack < int > mysta;
int lena, lenb, num[maxn];
char a[maxn], b[maxn];
void dfs(int i, int j, int k){
   char ch; int p;
   if (j == lenb){
      for (p = 0; p < k; p++){
         if (num[p])
            printf("i ");
         else
            printf("o ");
      }
      printf("\n");
      return;
   }
   if (i < lena){
      mysta.push(a[i]);
      num[k] = 1;
      dfs(i + 1, j, k + 1);
   }
   if (!mysta.empty() &&
mysta.top() == b[j]) {
      ch = mysta.top();
      mysta.pop();
      num[k] = 0;
      dfs(i, j + 1, k + 1);
      mysta.push(ch);
   }
   return;
}
int main(){
   while (scanf("%s%s", a, b)
         != EOF){
      lena = strlen(a);
      lenb = strlen(b);
      while (!mysta.empty()){
         mysta.pop();
      }
      printf("[\n");
      dfs(0, 0, 0);
      printf("]\n");
   }
   return 0;
}
```

Figure 6. A program code which employs *DFS* and *Stack*.

```
<a:SourceCode rdf:ID="ProgramCode_1">
  <a:has_stack_variable rdf:resource="#
    StackVariable_individual_2"/>
  <a:has_stack_pop_operation rdf:resource="#
    StackPopOperation_individual_1"/>
  <a:has_stack_push_operation rdf:resource="#
    StackPushOperation_individual_1"/>
  <a:has_depth_first_search
    rdf:resource="#DepthFirstSearch_individual_1"/>
</a:SourceCode>
```

Figure 7. The code individual for the program code.

B. Recognition of Programming Knowledge

After creating individuals for program codes, we are now ready to recognize the programming knowledge of data structures or algorithms lying in the program codes by using the inferential knowledge in the knowledge base.

In order to recognize the data structures or the algorithms, we use Java programming language to create an instance of a KAON2 [14] reasoner.

For each rule in the knowledge base, a query is built to recognize its corresponding data structure or algorithm. The query is used to search for all individuals of the class *SourceCode* that satisfy the preconditions of the reasoning rule. The result of the query is a group of individuals of the class *SourceCode*, each of which satisfies the preconditions and therefore indicates that its corresponding program code contains the data structure (or algorithm) associated with the reasoning rule.

Fig.8 shows a query about the data structure *Stack*. The literal *hasStackDS_CODE_STACK* uses the property *has_stack_used*, which is from the rule for the data structure *Stack* in Fig.5. This literal is used by the query *codeContainsDataStrcture*, so that the query about *Stack* could be associated with the rule of *Stack*. The query uses another literal for the class *SourceCode*, which indicates that the scope of the query is the individuals of the class *SourceCode*.

When the query, whose conditions are as the rule

```
Literal hasStackDS_CODE_STACK
=KAON2Manager.factory().literal(true, has_stack_used, new Term[]
{ CODE,CODE });

Query codeContainsDataStrcture
= reasoner.createQuery(new Literal[]
{
      KAON2Manager.factory().literal(true, SourceCode, new Term[]
      { CODE}),
      KAON2Manager.factory().literal(true,
      hasStackDS_CODE_STACK, new Term[] { CODE,
      DATASTRUCTURE }),
 }, new Variable[] { CODE, DATASTRUCTURE });
```

Figure 8. The query about *Stack* in Java language.

defines, is open, the reasoning process would be executed. As a result, the individuals which possess the three object properties *has_stack_variable*, *has_stack_pop_operation* and *has_stack_push_operation* would be selected.

We define some variables as follows.

- Q_{di} indicates the query for a data structure i in the knowledge base;
- R indicates an instance of the KAON2 reasoner, which could execute the queries about the data structures and algorithms;
- C_{di} indicates the result of the query Q_{di}, which is a set of individuals in the class SC that satisfy the preconditions of the reasoning rule.

The algorithm, which is used to recognize a data structure i, is described as follows.

1) For each individual $Ind(C)$ in the knowledge base, attach to it a data property $DPUd(i)$ whose value is assigned with *false*;
2) Create an instance R;
3) Build a query Q_{di} on a basis of the rule about the data structure i in the knowledge;
4) Get the result set C_{di} by executing the query Q_{di};
5) For each individual in the set C_{di}, set *true* to its data property $DPUd(i)$.

For each data structure in the knowledge base, do the above steps to recognize the data structure, and attach its corresponding property to each individual $Ind(C)$ of program codes. The steps used to recognize algorithms are similar to the steps of the above algorithm. Fig.9 depicts the individual of the program code in Fig.6, whose data properties *has_stack_used* and *has_dfs_used* are assigned with *true*. The recognition results would be

```
<a:SourceCode rdf:ID="ProgramCode_1">
  <a:has_stack_variable rdf:resource="#
    StackVariable_individual_2"/>
  <a:has_stack_pop_operation rdf:resource="#
    StackPopOperation_individual_1"/>
  <a:has_stack_push_operation rdf:resource="#
    StackPushOperation_individual_1"/>
  <a:has_depth_first_search
    rdf:resource="#DepthFirstSearch_individual_1"/>
  <a:has_stack_used
    rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">tru
    e</a:has_stack_used>
  <a:has_dfs_used
    rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">tru
    e</a:has_dfs_used>
</a:SourceCode>
```

Figure 9. The code individual with recognition result

used to discover the relationships between data structures and algorithms.

C. Discovery of Programming Knowledge Relationships

We define that the relationships between data structures and algorithms are reflected by the frequencies of the two kinds of programming knowledge used in program codes simultaneously. When a data structure and an algorithm are contained in a program code, there may be a relationship between them. For instance, the data structure *Stack* and the algorithm *Depth First Search* are often contained in a program code, which implies a relationship between them.

We propose a 3-tuple data model R = <A, D, N> to mark the relationships between algorithms and data structures lying in the program codes. Here, R denotes the set of relationships, A denotes the set of algorithms in the knowledge base, D denotes the set of data structures in the knowledge base and N denotes the set of frequencies. We assume that there are m algorithms and n data structures, and there are $m \times n$ different relationships between algorithms and data structures. Thus, each of the 3-tuples can be expressed as $R_{(i,j)} = \langle A_i, D_j, N_{(i,j)} \rangle$. In this tuple, A_i (i = 1, 2, …, m) denotes an algorithm in the set A; D_j (j = 1, 2, …, n) denotes a data structure in the set D; $N_{(i,j)}$ (i = 1, 2, …, m, j = 1, 2, …, n) shows the frequency of the algorithm A_i and the data structure D_j used simultaneously in the same program codes.

The value of each element $N_{(i,j)}$ is the number of all program codes that use both the algorithm A_i (i = 1, 2, …, m) and the data structure D_j (j = 1, 2, …, n). This value can be obtained by counting the number of individuals whose data properties *DPUa*(i) and *DPUd*(j) are both *true*. Thus, the relationship $R_{(i,j)}$ between an algorithm A_i and a data structure D_j is indicated by the 3-tuple $\langle A_i, D_j, N_{(i,j)} \rangle$.

According to the elements of the set R, we can calculate the total frequency A_{iTotal} of each algorithm A_i by the formula in (1), where A_{iTotal} means the total number of the program codes that use the algorithm A_i simultaneously with at least one data structure in the knowledge base.

$$A_{iTotal} = \sum_{j=1}^{n} N_{(i,j)} \tag{1}$$

For each pair of a data structure D_j and an algorithm A_i, we give a definition that the rate $Rate_{(i,j)}$ of the data structure D_j is the ratio of the number $N_{(i,j)}$ to the number A_{iTotal}. The rate $Rate_{(i,j)}$ indicates how much the data structure D_j is related to the algorithm A_i. Its value is between 0 and 1 inclusive. The bigger the value, the more related to the algorithm A_i the data structure D_j is. The rate can be calculated by the formula in (2).

$$Rate_{(i,j)} = \frac{N_{(i,j)}}{A_{iTotal}} \tag{2}$$

IV. LEARNING METRIALS

In this Section, we will introduce the programming problems on OJ systems, and then organize the programming problems and their program codes on the web into learning materials in terms of algorithms and data structures with hyperlink techniques.

A. Information Items of Programming Problems

A programming problem from an OJ system may consist of many information items. We just consider the following items: Pro.ID, problem name, problem URL, problem description, and pass rate.

We define that Pro.ID consists of two parts, the name of the OJ system and the serial number of a programming problem on the OJ system.

The pass rate of a programming problem is defined as the percentage of the accepted program codes out of all the program codes submitted by students for the sake of solving the programming problem. The pass rate of a programming problem reflects its difficulty. If the pass rate of a programming problem is low, it means that the problem is difficult.

The web page of a programming problem contains some information items about this problem.

Fig.10 shows a part of the web page of a programming problem from the HDU[1] system. We could get the information items about this problem from this page as shown in TABLE III. This programming problem has its serial number "3078" on the OJ system named "HDU", so its Pro.ID is "HDU 3078". From this page, we can see that its problem name is "Network". The description of the programming problem is the string "The ALPC company is…..". The total number of program codes submitted by students for the programming problem is 96,

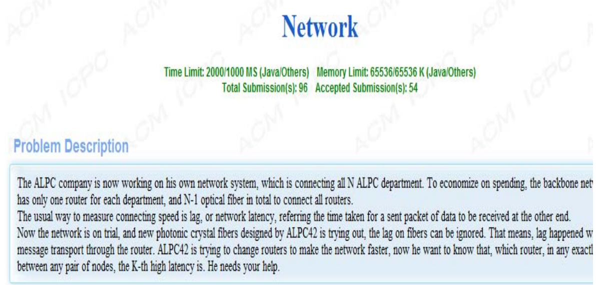

Figure 10. The web page of a programming problem from HDU

TABLE III.
THE ITEMS AND THEIR INFORMATION OF A PROBLEM

Item	Information
OJ System Name	HDU
Serial Number	3078
Pro.ID	HDU 3078
Problem Name	Network
Problem URL	http://acm.hdu.edu.cn/showproblem.php?pid=3078
Problem Description	The ALPC company is…..
Total Submissions	96
Accepted Submissions	54
Pass Rate	56.25%

and the number of the accepted codes is 54. Thus, the pass rate of this programming problem is 56.25%, which

[1] HDU: http://acm.hdu.edu.cn/

is the number of accepted codes divided by the total number of program codes.

B. Obtainment of Information Items

The Pro.ID and the name of a programming problem can be obtained during the process of downloading solution reports from the web if we use the method mentioned in [1].

According to the Pro.ID of a programming problem, we can get the URL of the problem by string concatenation. TABLE IV shows the URLs of programming problems on three different OJ systems, where “xxxx” is the problem serial number. Take the programming problem in TABLE III for example, its Pro.ID is “HDU 3078”, so that its problem URL is “http://acm.hdu.edu.cn/showproblem.php?pid=3078”. By using this URL, we can obtain the web page which contains its name and description as shown in Fig.10.

The pass rate of a problem is contained in the volume page which this problem belongs to. We can get its pass rate from the volume page.

C. Orgainzition of program codes and problems

We download solution reports on the web for programming problems using the method described in [1]. During the process, the URLs of reports, Pro.ID and problem names could be obtained. According to Pro.ID, we can get the URL of a problem, and then obtain its pass rate.

TABLE IV.
THE PROBLEM URLS FROM DIFFERENT OJ SYSTEM

OJ Name	URL of Problem Description
HDU	http://acm.hdu.edu.cn/showproblem.php?pid=xxxx
POJ	http://poj.org/problem?id=xxxx
ZOJ	http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=xxxx

We get program codes from solution reports, and then discover the relationships between algorithms and data structures used in them.

With the discovered relationship, we could organize the program codes on the web and the programming problems on online judges into learning materials in terms of algorithms and data structures.

D. Structure of Learning Metrials

The learning materials possess a hierarchical structure. The highest level of this structure is the algorithms. Below each algorithm, there are several data structures, which are sorted with their percents from high to low. For each data structure, several problems, which are sorted by their pass rates from high to low, are displayed. The materials also provide one or more solution reports for each programming problem.

So these materials can help learners to study programming knowledge in terms of algorithms and data structures systematically.

V. EXPERIMENT AND RESULT

In our experiment, we took program codes written in C/C++ programming language and the programming problems from OJ systems as our research data. The program codes were obtained from solution reports, which were downloaded from many web sites, and the programming problems were from three OJ systems HDU, POJ[2], and ZOJ[3].

We built a knowledge base for seven algorithms and twelve data structures by using the ontology tool protégé [15]. The seven algorithms are *BFS*, *DFS*, *MaxFlow*, *ShortestPath*, *MST*, *Topological* and *Linear DP*. The twelve data structures are *Stack*, *Queue*, *LinkList*, *UnionFindSet* and the eight data structures from the Standard Template Library (STL).

We collected 6643 solution reports of 480 programming problems, and obtained 3752 program codes written in the C/C++ programming language from these reports. We created 3752 individuals of the class *SourceCode* for the 3752 program codes respectively, and recognized data structures and algorithms used in these program codes.

Since some of the program codes did not use any data structure or any algorithm in the knowledge base we created, for the sake of discovering the associated relationships between data structures and algorithms, we selected 1073 suitable program codes, which use both data structures and algorithms according to the recognition results.

Some results of our experiment are shown in TABLE V. In the string “HDU_2101_code_1” from the first column, “HDU” is the OJ system name, “2101” is the serial number of the problem on the HDU system, and “code_1” means that this code is from the first solution report of the programming problem with the Pro.ID “HDU 2101”. The second column is the recognition result of data structures and the last one is the recognition result of algorithms.

After recognizing the data structures and algorithms used in the program codes, we can analyze the relationships between the two kinds of knowledge. We calculated the total frequency A_{iTotal} of each algorithm A_i by the formula in (1), and the rate $Rate_{(i,\ j)}$ of each data structure D_j which is used simultaneously with the algorithm A_i by the formula in (2).

[2] POJ: http://poj.org/
[3] [illegible]

TBALE V.
RECOGNITION RESULT OF DATA STRUCTURES AND ALGORITHMS

Program Code Name	Data Structure	Algorithm
HDU_2101_code_1	STLQueue	BFS
POJ_3522_code_1	UnionFindSet	MST
POJ_2337_code_1	STL Stack	DFS
POJ_3272_code_1	STLQueue	Topological

TABLE VI shows the relationships between the seven algorithms and several data structures used in program codes simultaneously. The second column lists the algorithm names, each followed by the number (in the parentheses) of all program codes that use the corresponding algorithm. The third column displays the data structures, which are used in program codes simultaneously with the corresponding algorithms. The last column shows the rate *Rate*(i, j) of the data structure D_j related to the algorithm A_i. The data structures, which are used in program codes simultaneously with the same algorithm, are sorted by their rate *Rate*(i, j) from high to low.

TABLE VI.
RELATIONSHIPS BETWEEN DATA STRUCTURES AND ALGORITHMS

No	A_i (A_{iTotal})	D_j	$Rate_{(i, j)}$
1	BFS (650)	STL Queue	70.17%
		STL Priority_Queue	12.52%
2	DFS(371)	STL Queue	27.49 %
		STLVector	24.53%
		STL Stack	10.24 %
3	MaxFlow(77)	STL Queue	83.12%
4	ShortestPath(74)	STL Priority Queue	62.16 %
		STL Queue	13.51%
		STL Vector	10.81 %
5	MST(57)	UnionFindSet	42.11 %
		STL Queue	24.56 %
		STL Priority_Queue	7.02 %
6	Topological(32)	STL Queue	40.63 %
		STL Priority Queue	18.75 %
		STLStack	12.50 %
7	Linear DP(24)	STLVector	33.33%
		STL Queue	33.33%
		STLMap	12.50%

Fig.11 shows the hierarchical structure of programming knowledge in terms of algorithms and data structures. At the top level are algorithm names and at the lower level are data structure names. For each algorithm, it displays its correlative data structures which are sorted by the percentage from high to low. Take the algorithm *DFS* as an example, the number 371 in the parentheses which follows the algorithm name means that there are 371 program codes out of 1073 program codes use this

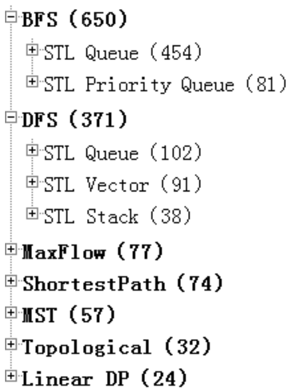

Figure 11. The hierarchical structure of programming knowledge

algorithm. The names of data structures appearing

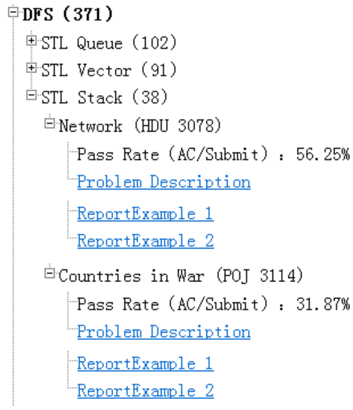

Figure 12. The hierarchical structure of the learning materials

simultaneously with *DFS*, are *STL Queue* (102), *STL Vector* (91) and *STL Stack* (38).

Fig.12 shows the hierarchical structure of the learning materials which consist of programming problems distributed on online judges and their program code resources on the web. The learning materials display the pass rates, the hyperlinks of problem descriptions and some solution reports. For each data structure, the programming problems are ordered by their pass rates from high to low. When students want to learn how to implement data structures and algorithms in solving problems, they can easily find out the suitable problems to practice from this structure. Learners can also choose suitable problems to practice according to the difficulty of problems. For each problem, we give one or more solution reports for students to study. Fig.12 shows two programming problems whose solutions use both *DFS* and *STL Stack*. The Pro.IDs of them are "HDU 3078" and "POJ 3114", respectively. The pass rate of the problem named "Network" is 56.26%, and it is higher than the pass rate 31.87% of the problem named "Countries in War", thus the former problem is displayed above the latter one.

VI. CONCLUSION

We have proposed an approach to discover the relationships between data structures and algorithms with ontology techniques, which consists of constructing the knowledge base about data structures and algorithms, recognizing the two kinds of programming knowledge used in program codes with ontology reasoning technique, and discovering the relationships between them by recording their frequencies. With the discovered relationships, we organized the program codes on the web and the programming problems distributed on OJ systems into learning materials with a hierarchical structure. In this structure, the top level is algorithms, and the lower is data structures. Under the level of data structures, several problems are provided in the descending order of their pass rates. For each problem, several solution reports are given for students to learn. We believe that the materials would be useful for students to learn the programming knowledge of data structures and algorithms.

ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China (NSFC) under Grant No. 60973121.

REFERENCES

[1] G. Zhu and L. Fu, "Automatic Organization of Programming Resources on the Web," in Advances in *Computer Science and Information Engineering*. vol. 168, D. Jin and S. Lin, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 675-681. doi: 10.1007/978-3-642-30126-1_106.

[2] ACM/ICPC. http://cm.baylor.edu/welcome.icpc, 2012-11-20/2012-12-25.

[3] G. Zhu and Z. Zhang, "Knowledge unit discovery for programming tutoring based on Formal Concept Analysis," in *Educational and Information Technology (ICEIT), 2010 International Conference on*, 2010, pp. V3-476-V3-479. doi: 10.1109/ICEIT.2010.5607545.

[4] J. Fagerberg, M. Fosaas and M. Sapprasert, "Innovation: Exploring the knowledge base," Research Policy, vol. 41, pp.1132-1153, 2012. doi: 10.1016/j.bbr.2011.03.031.

[5] H. Gao and X. Chen, "Ontology and CBR-based Dynamic Enterprise Knowledge Repository Construction," *Journal of Software*, vol. 7, pp. 1211-1218, Jun 2012. doi: doi:10.4304/jsw.7.6.1211-1218.

[6] W. Gao and T. Xu, "Characteristics of Optimal Function for Ontology Similarity Measure via Multi-dividing," *Journal of Networks*, vol. 7, pp. 1251-1259, Aug 2012. doi:10.4304/jnw.7.8.1251-1259.

[7] Studer R., Benjamins V.R., and Fensel D., "Knowledge engineering: Principles and methods, " *Data & Knowledge Engineering*, vol. 25, pp.161-197, March 1998. doi: 10.1016/j.bbr.2011.03.031.

[8] C. Brewster and K. O'Hara, "Knowledge representation with ontologies: the present and future," *Intelligent Systems, IEEE*, vol. 19, pp. 72-81, 2004. doi: 10.1109/MIS.2004. 1265889.

[9] M. Vargas-Vera, E. Motta, J. Domingue, S.B. Shum and M. Lanzoni, "Knowledge Extraction Using an Ontology-Based Annotation Tool",Workshop on *Knowledge Markup & Semantic Annotation*, ACM Press, New York, 2001, pp.5-12.

[10] H. Alani, et al., "Automatic ontology-based knowledge extraction from Web documents," *Intelligent Systems, IEEE*, vol. 18, pp. 14-21, 2003. doi: 10.1109/MIS.2003. 1179189.

[11] D. C. Wimalasuriya and D. Dou, "Ontology-based information extraction: An introduction and a survey of current approaches," *J. Inf. Sci.*, vol. 36, pp. 306-323, 2010. doi: 10.1177/0165551509360123.

[12] L. Zhang, M. Zhu and W. Huang, "A Framework for an Ontology-based E-commerce Product Information Retrieval System," *Journal of Computers*, vol. 4, pp. 436-443, Jun 2009. doi: 10.4304/jcp.4.6.436-443.

[13] B. Jiang, M. Zhu and J. Wang, "Ontology-Based Information Extraction of Crop Diseases on Chinese Web Pages," *Journal of Computers*, vol. 8, pp. 85-90, Jan 2013. doi: 10.4304/jcp.8.1.85-90.

[14] KAON2. http://kaon2.semanticweb.org, 2012-11-20/2013-01-10.

[15] Protégé. http://protege.stanford.edu, 2012-11-01/2012-12-20.

Guojin Zhu is an associate professor at the Department of Computer Science, Donghua University (DHU), Shanghai, China. He received his M.S. and Ph.D. degrees from DHU in 1991 and 2007, respectively. He was a visiting scholar at the Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, USA from November 2007 to November 2008. His current research interests include semantic web, knowledge discovery, and neural computing.

Zhiyue Yu is a graduate student of Computer Application Technology at Donghua University. She was born in Hebei province, P.R. China in 1987, and received her Bachelor of Science degree from Shaanxi University of Science and Technology in 2010. Her current main research interest is computer network and artificial intelligence.

Jiyun Li is an associate professor at the Department of Computer Science, Donghua University (DHU), Shanghai, China. She received her M.S. and Ph.D. degrees from DHU in 1996 and 2003, respectively. She has been visiting scholar at the Department of Psychology and Brain Science in Indiana University and Textile and Fashion Institute in Hongkong Polytechnique University. Her current research interests include cognitve modeling, knowledge discovery, and neural computing.