Autom Softw Eng
DOI 10.1007/s10515-014-0170-2

AutoQuery: automatic construction of dependency queries for code search

Shaowei Wang · David Lo · Lingxiao Jiang

Received: 24 December 2013 / Accepted: 10 September 2014

Abstract Many code search techniques have been proposed to return relevant code for a user query expressed as textual descriptions. However, source code is not mere text. It contains dependency relations among various program elements. To leverage these dependencies for more accurate code search results, techniques have been proposed to allow user queries to be expressed as control and data dependency relationships among program elements. Although such techniques have been shown to be effective for finding relevant code, it remains a question whether appropriate queries can be generated by average users. In this work, we address this concern by proposing a technique, AutoQuery, that can automatically construct dependency queries from a set of code snippets. We realize AutoQuery by the following major steps: firstly, code snippets (that are not necessarily compilable) are converted into program dependence graphs (PDGs); secondly, a new graph mining solution is built to return common structures in the PDGs; thirdly, the common structures are converted to dependency queries, which are used to retrieve results by using a dependence-based code search technique. We have evaluated AutoQuery on real systems with 47 different code search tasks. The results show that the automatically constructed dependency queries retrieve relevant code with a precision, recall, and F-measure of 68.4, 72.1, and 70.2 %, respectively. We have also performed a user study to compare the effectiveness of AutoQuery with that of human generated queries. The results show that queries constructed by

S. Wang · D. Lo (✉) · L. Jiang
School of Information Systems, Singapore Management University,
80 Stamford Road, Singapore, Singapore
e-mail: davidlo@smu.edu.sg

S. Wang
e-mail: shaoweiwang.2010@smu.edu.sg

L. Jiang
e-mail: lxjiang@smu.edu.sg

Published online: 25 September 2014

AutoQuery on average help to retrieve code fragments with comparable F-measures to those retrieved by human constructed queries.

Keywords Code search · Dependency query · Query construction · Graph mining

1 Introduction

Many software projects today contain a large amount of source code. Searching through this mass of source code manually would take much developer time and resources. Many software engineering tasks can benefit from efficient and effective automated code search. One scenario where code search can be employed is fixing a bug during maintenance: developers often need to propagate a change to many other similar locations; without the aid of a code search tool, developers may need to tap on their experience to browse relevant source code files and manually find the code fragments requiring changes. This is not only tedious but also error-prone. Some code location requiring changes may be missed out, causing further bugs or even security loop holes. To address the need of finding relevant code, a number of code search tools have been proposed. These tools accept user queries and return code fragments that match the query.

Many code search tools are text-based. They accept user queries as texts and search code fragments that contain identifier names that match or are related to the words in the query (Chan et al. 2012; McMillan et al. 2011). A comprehensive survey of 89 feature location and code search studies is presented by Dit et al. (2013). Quite a few studies view source code more than just texts. It also contains structures and dependency relations among program elements. To leverage these dependencies to improve search accuracy, dependence-based code search techniques have been proposed. They accept queries expressed as dependency relationships among program elements of interest (Wang et al. 2010, 2011a), and return code fragments whose constituent program elements satisfy the dependency relationships. It has been shown that dependence-based code search could outperform text-based code search (Wang et al. 2011a).

However, there is one drawback that potentially hampers the usage of dependence-based code search. Often it is hard to construct dependency queries. Users need to be able to visualize the dependency relationships among program elements, select relevant ones and express these as queries. This process might be daunting for many potential users. In this work, we aim to address the drawback of dependence-based code search by automatically constructing dependency queries from code examples. Using our tool, developers could input a set of example code fragments, which correspond to snippets of code that users need to change to address a particular need (e.g., new feature implementation, bug fix, etc.). By taking multiple code fragments as input, our tool can learn important dependencies shared by the examples and filter unimportant dependencies that are peculiar to an individual example. Our tool eventually constructs a dependency query, which can then be used to identify other code fragments that need to be changed in a similar way (to address the same need) by leveraging a dependence-based code search tool. Of course, if needed, users can make changes to the dependency

query before inputting them to the code search tool. Our setting supports automation while still allowing users to be in control in the code search process and thus important domain knowledge can be leveraged for effective code search.

We propose a tool named AutoQuery. Our tool first converts a given set of code fragments (not necessarily compilable) into program dependence graphs (PDGs). These program dependence graphs are then analyzed and their commonalities are highlighted. We develop a new graph mining solution that could mine for these commonalities from the PDGs expressed as multi-labeled graphs with textual and node type labels. The resultant mined sub-graph is then converted to a dependency query. We use the dependency search tool proposed by Wang et al. as the backend code search tool (Wang et al. 2010). AutoQuery builds upon the dependence-based code search tool (Wang et al. 2010) by automating the process of generating dependence queries from sample code fragments. Previously, users of the dependence-based code search tool needs to manually construct dependence queries.

Our tool can be used to help software engineers in various scenarios. For illustration purpose, consider Alex a software engineer who is responsible to perform a corrective maintenance task which may affect a number of files. Alex has localized two buggy code fragments. However, it is likely that there are many other buggy code fragments that are distributed across many source code files and need to be fixed in a similar way. Our tool AutoQuery can be used to help Alex finds the remaining buggy code fragments. Alex simply needs to input the buggy code fragments that he has localized to AutoQuery and AutoQuery in turn will construct a DQL query and invoke an underlying code search tool to return the other buggy code fragments. Alex can save much time since he does not need to search for the buggy code fragments manually.

There are several challenges that we need to solve to build AutoQuery. First, the code fragments are not necessarily compilable. Many tools that construct PDGs from code, e.g., Codesurfer (2013), require compilable code. Thus, we need to process code fragments into compilable code units. Next, most graph mining solutions, e.g., Yan and Han (2002, 2003) and Zhu et al. (2011), only work on simple graphs whose nodes and edges are labelled with simple types. PDGs are not simple graphs; each node in a PDG is a program element and contains not only node type information and also textual contents describing the fragment of the source code corresponding to the program element. Thus we need to build a new graph mining solution that handles our special graph representation that captures information in a PDG.

We evaluate our query generation approach on 47 realistic code search examples we extract from repositories of four software projects (Apache Http Server, Inkscape, Apache Subversion, and Libmpeg2). We show that using AutoQuery we can generate good queries that could be used to retrieve relevant code with precision, recall, and F-measure of 68.4, 72.1 and 70.2 % respectively. We have also conducted a user study to compare automatically generated queries with manually generated queries. We find that our automatically generated queries can perform as well as human generated queries.

Our contributions are as follows:

1. We are the first to propose an approach that can automatically generate dependency queries from several code examples.

2. We generate PDGs from non-compilable code fragments and propose a new graph mining technique that could search for common substructures in specialized multi-label graphs with nodes containing both node type information and textual contents.
3. We have experimented our approach on 47 realistic code search scenarios. We show that AutoQuery plus dependence based code search tool could return relevant codes with precision, recall, and F-measure of 68.4, 72.1, and 70.2 %. We have performed a user study that shows that our generated queries are comparably as good as human generated queries in returning relevant code.

The structure of this paper is as follows. In Sect. 2, we present PDG and dependence-based code search in more details. In Sect. 3, we describe our overall framework at high level. We zoom-in into the PDG generation engine of our framework in Sect. 4. We elaborate our query generation engine of our framework in Sect. 5. We present our experiments that evaluate the effectiveness of AutoQuery in Sect. 6. We discuss related studies in Sect. 7. We conclude and mention future work in Sect. 8.

2 Preliminaries

In this section, we give a brief description of program dependence graphs. We also describe dependence-based code search briefly.

2.1 Program dependence graphs

A code base comprises of program elements (e.g., expressions, statements, and declarations) that are related to one another via control and data dependencies. An element is control dependent on another if the execution result of the latter determines if the former is executed or not. An element is data dependent on another if the former may use a variable whose value is determined by the latter. A program dependence graph (PDG) captures all these dependencies, including call relations. Each node in the graph corresponds to a program element in the code. Each edge corresponds to either data or control dependence. Program dependence graph has been shown to represent certain semantic aspects of code and useful for various purposes (Liu et al. 2006; Gabel et al. 2008).

In this paper, we represent a PDG as a graph $G = (N, E)$, where N is a set of nodes and E is a set of edges. N is defined as the set $\{n_1 = (ntype_1, text_1), \ldots n_i = (ntype_i, text_i), \ldots\}$. Each node has two labels: $ntype_i$ which is the node type,[1] and $text_i$ which is the textual representations of the corresponding program element in a piece of source code. Let $n_i.ntype$ and $n_i.text$ denote the node type and text label of node n_i. E is defined as the set $\{e_1 = (n_1^L, n_1^R, etype_1), \ldots, e_i = (n_i^L, n_i^R, etype_i), \ldots\}$. Each edge contains two nodes n_i^L and n_i^R; these are the nodes connected by the edge. Each edge also has a type $etype$, $etype$ can either be data

[1] We use the node types defined by CodeSurfer. There are 33 different node types for C/C++, e.g., function call, expression, etc.

Table 1 *DQL* syntax

query	::=	(*ndecl*)*; (*ndesc*)*; (*rdesc*)*; *target*;
ndecl	::=	*tlist id*
tlist	::=	*tlist* | *type*
type	::=	*func* | *var* | *assgn* | *decl* | *ctrlPoint* | *stmt*
ndesc	::=	*id* (*cond*)*
cond	::=	[not] *ucond*
ucond	::=	contains *string* | inFile *string* | inFunc *string* | atLine *number* | ofControlType *ctype* | ofType *string* | ofType *native*
ctype	::=	for | while | switch | if
rdesc	::=	*id op id*
op	::=	[oneStep] *dependOp*
dependOp	::=	dataDepends | controls
target	::=	(*id*)*
id	::=	*string*
string	::=	(A–Z, a–z, 0–9)+
number	::=	(0–9)+

dependency or control dependency (i.e., $etype \in \{data, control\}$). Most graph mining algorithms, only accept *simple* graphs, i.e., graphs with one categorical label per node (edge). We create a simple graph representation of a PDG G by dropping the text information from the node labels. We denote the resultant simple graph G^{notext}.

2.2 Dependence-based code search

Dependence-based code search accepts as input queries expressing dependencies among various program elements (Wang et al. 2010, 2011a). To help users formulate queries and provide inputs for dependence-based code search, the Dependence Query Language (DQL) was proposed in Wang et al. (2010). Its syntax is shown in Table 1.

DQL has four parts: node declaration (*ndecl*), node description (*ndesc*), relationship description (*rdesc*), and targets (*target*). *Ndecl* declares node variables and their types. *Ndesc* specifies constraints on declared node variables. *Rdesc* specifies constraints on the relations among declared node variables. *Target* specifies the variables specified in *ndecl* that are desired search targets. When a *DQL* query is processed on a PDG, nodes in the PDG that match the node variables specified in *target* and satisfy the constraints specified in *ndecl*, *ndesc* and *rdesc* would be returned.

Node declaration This part of a query is to declare some node variables that are to be mapped to nodes in a PDG. We assign types to node variables; each type is a PDG node type, e.g., function call, expression, declaration, etc.

Node description This part of the query specifies further constraints on declared node variables (*cond* and *ucond*). To specify constraints, developers can use the following

unary operators: $contains$, $inFile$, $inFunc$, $atLine$, $ofControlType$, and $ofType$. The operator $contains$ allow developers to specify that a particular node needs to contain a particular text. The operators $inFile$ and $inFunc$ allow developers to specify a node that is located inside a particular file or function respectively. The operator $ofControlType$ allows user to specify a control node of a particular type (i.e., for, while, switch, or if). The operator $ofType$ allows user to specify a node of a particular type. If the type is specified as $native$ it corresponds to built-in types of a programming language, e.g., "char", "float", "int", etc.

Relationship description This part of the query specifies constraints governing the dependency relationships (data and control) between two declared node variables. They are expressed as operators: $dataDepends$, $controls$, and $onestep$. $Onestep$ can be used together with either $dataDepends$ or $controls$; it specifies that a node is *directly* data or control dependent on another node.

Targets This part of a DQL query specifies the target node variables that would be returned as the output of the query. This set of variables is a subset of all declared variables. The other variables serve as contexts for the target nodes.

Example An example DQL query which has been shown useful to find code fragments of interest in a real code search task by Wang et al. (2010) is shown below:

Node declarations:	*decl A, ctrlPoint B*;
Node descriptions:	*A not ofType native*;
Relationship descriptions:	*B oneStep dataDepends A*;
Targets:	*B*;

The DQL query specifies a declaration A and a control point B. The declaration must not be one of the built-in types. B is directly data dependent on A, and we are interested to find all such Bs.

3 Overall framework

In this section, we present the overall framework of our automatic query generation approach. In Sects. 4 and 5, we elaborate the core components of our approach.

The structure of our automatic query generation approach is shown in Fig. 1. It consists of two major processing components (shown as big rounded rectangles) namely PDGs generation engine and query generation engine. The outputs of the PDGs generation engine, which are the PDGs of the code fragments given by a user, is feeded to the query generation engine. The sequence of interactions among the user and the various components of the framework is described in the following paragraphs.

First, a user posts several code snippets to the PDGs generation engine. Inside it, the non-compilable code fragments would be extended to compilable code by adding missing type definitions, missing variables, and missing methods. We support non-compilable code as users could find examples online (e.g., from question and answer sites, from software forums, etc.); these examples are often only code fragments which cannot be compiled by itself. Next, the PDGs generation engine uses Codesurfer

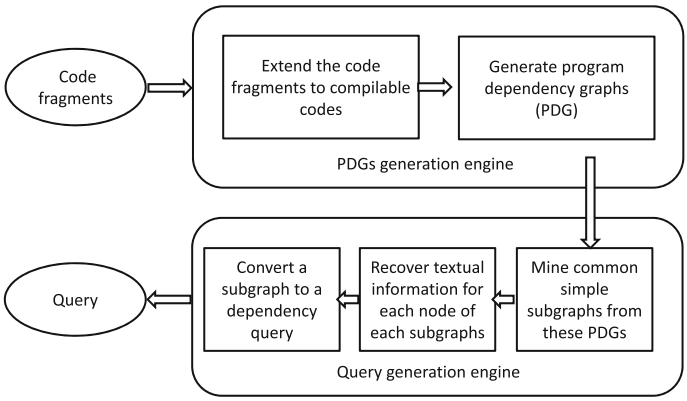

Fig. 1 Overall framework

(2013) to convert the compilable codes to their corresponding program dependence graphs. Each PDG node corresponds to a program element and contains two pieces of information: node type, and textual representation of the program element in the source code.

Next, the PDGs generated by the PDGs generation engine would be sent to the query generation engine. The query generation engine would first transforms PDGs to simple graphs by dropping the textual representation information from the nodes. A maximal common subgraph is then mined from these simple graphs using an existing graph mining algorithm. Then, each node in the mined simple subgraph would be mapped to its original node in the PDGs generated by the PDG generation engine, to extract corresponding textual representation information. A node in a subgraph could be mapped to many nodes in the original PDGs. We develop a heuristics to decide the most appropriate node. After getting a common subgraph and text information for each node in the subgraph, the engine converts the subgraph to a dependency query acceptable by the dependence-based code search tool developed by Wang et al. (2010) (see Sect. 2).

4 PDGs generation engine

In this section, we present the steps to generate PDGs for given code fragments (or incomplete lines of code). Code fragments given by a user might not be compilable due to various reasons, e.g., missing type definitions, missing method declarations, etc. Many examples, especially those available in software forums and question and answer sites, are in this format. In this work, we are able to add the missing type definitions, variable declarations, and function definitions to make a non-compilable code fragment compilable. There have been studies in literature (cf. the Related Work Sect. 7.1) that make incomplete code compilable, but our goal is not to preserve the full semantics of the code; rather, we just would like to preserve the dependencies among the various program elements in the code fragment. Our implementation and evaluation focuses on C/C++ code.

4.1 Code extension

Algorithm 1 Code extension algorithm

```
1: CodeExtension(RPT)
2: Input:
3: RPT: Root of the parse tree of a code fragment
4: Output: Extended compilable code
5: Method:
6: Initialize DV, UV, UF, UT, EX, RV to {}
7: Traverse(RPT, DV, UV, UF, UT, EX ,RV)
8: Let MAP = Mappings from m in (DV ⋃ UV ⋃ EX) to its type
9: repeat
10:    if s1 and s2 in MAP in a type-equivalence relation in RV then
11:       Union s1 and s2 into MAP
12:    end if
13: until No more change in MAP
14: Replace Unknown type in MAP with int
15: Add necessary (global) variable declarations based on UV
16: Add necessary function definitions based on UF
17: Add necessary classes based on UT
18: Add other necessary code (e.g., include statements, etc.)
```

To infer the types of variables and signatures of invoked functions in a code fragment, we first obtain the parse tree of the code fragment. We create this parse tree by using a python library called pycparser.[2] From this parse tree we infer the types of undeclared variables and the signatures of invoked functions (including information of their parameters and return types). Undefined types appearing in the code fragment are inferred as well. We would infer relevant data fields and functions of an undefined type that are used in the code fragment. After the above information is gathered, we extend the code fragment by adding:

1. Declarations of undeclared variables
2. Definitions of undefined functions
3. New classes (data types) that specify undefined types

Our code extension algorithm is shown in Algorithm 1. It takes a parse tree as an input and outputs a piece of compilable code. The major task of the algorithm is to infer a list of undeclared/undefined variables, function arguments, and function return values along with their inferred types, a list of undefined functions along with their signatures, and a list of undefined types along with their relevant attributes and functions.

We first initialize several sets to store a list of declared variables and constants (i.e., DV), a list of undeclared/undefined variables, function arguments, and return values (i.e., UV), a list of undefined functions (i.e., UF), a list of undefined types (i.e., UT), a list of expressions (i.e., EX), and a list of type-equivalence relations among variables and expressions (i.e., RV) (line 6). Note that each expression could be of various types, e.g., function invocation, operator, variable reference, etc. The heuristics in Table 2 are used to infer type-equivalence relations and update the RV list. For example, a variable/expression at the left side of an assignment has "assignment"

[2] https://bitbucket.org/eliben/pycparser.

Table 2 Inference heuristics

Name	Heuristic
Assignment	If two expressions are on the two sides of an assignment, then they are of the same type
Operator	If two expressions are connected with an operator in an expression, then they are of the same type
Switch-case condition	If two expressions are either the condition expression of a switch statement or the expression in a case statement of the same switch statement, then they are of the same type
Function definition and invocation	A parameter of a function must be of the same type as the corresponding argument of its invocation. The return type of a function must be of the same type as the corresponding return value of its invocation

relationship with a variable/expression at the right side of the assignment. We then update, add or remove items into these lists by traversing the parse tree using the procedure `Traverse` defined in Algorithm 2 (Line 7). The `Traverse` procedure walks each node in a parse tree to discover (un)declared/(un)defined variables, expressions, functions, types and store them in corresponding lists. During the traversal, all type-equivalence relations between among variables and expressions are stored for type inference later. We elaborate procedure `Traverse` in latter paragraphs. After the lists are initialized, we create a set MAP that contains mappings from a set of variables, arguments, return values, and expressions to their type (line 8). MAP is initialized as follows:

$$MAP = \{\{m\} \mapsto \mathtt{m's\ type} | m \in DV\} \cup \{\{m\} \mapsto \mathit{Unknown} | m \in UV\} \cup \{\{e\} \mapsto \mathit{Unknown} | e \in EX\} \quad (1)$$

For each member of DV, we have its type, and thus we can insert a mapping between itself and its type in MAP. For each member of UV, initially we set its type in MAP to *Unknown*. Next, we try to infer variables, arguments, return types, and expressions stored in MAP that have the same type (lines 9–13). If two members of MAP have a type-equivalence relation, we combine their corresponding mappings in MAP (lines 10–12). Note that we consider two members, which have a type-equivalence relation if there is type-equivalence relation between two elements from each member (i.e., s_1, s_2), as type-equivalence. We perform this unification iteratively by subsequent applications of the inference heuristics Milner (1978). We try to put all variables of the same type together and infer the types of undeclared/undefined variables, arguments, and return values based on variables of known types. As the unification proceeds, the number of remaining mappings reduces. The unification process ends when the number of mappings does not change anymore (i.e., a fixed point is reached) (line 13). Note that this fixed point would eventually be reached since every time line 11 is executed, we would reduce the number of mappings in MAP by one. Since there is a finite number of mappings in MAP, the repeat-until structure (lines 9–14) would eventually terminate. When this happen, a fixed point is reached. At the end of the process, we have inferred

```
switch(x){
    case 1:
        return true;
    default:
        return false;
}
```

Fig. 2 An example code snippet for the switch-case condition

equivalence classes of variables, arguments, and return values that must be of the same type. For some of these equivalence classes, we can infer the type since one member of the class is defined in the code fragment. For other equivalence classes, the type is still `Unknown`. For these equivalence classes, we replace `Unknown` with int (line 14). This operation (replacing `Unknown` with int) does not affect the compilability of the resultant code as there is no conflicting type assignment. Please note that we are only interested in recovering dependency relationships among program elements in the code fragment and not the full semantics of the code fragment.

At the end of the above steps, we would be able to infer the types of all variables, function arguments and return values. We also know the signatures of the missing function (including types of arguments and return value.) and the type definition (e.g., name of attribute, member function, and their corresponding types, etc.) . Using these pieces of information, we then add necessary code (e.g., variable declarations, function definitions, new classes, include library, etc.) to make the code fragment compile (lines 15–18).

In Algorithm 2, we traverse the parse tree node by node starting from its root. The goal is to update a list of declared variables and constants (DV), undeclared/undefined variables, arguments, return values (UV), undefined functions (UF), undefined types (UT) and relations of variables and expressions (RV) based on the parse tree. We consider several cases where we need to update our lists: start of a local block (e.g., if, for, while, switch-case blocks, etc.), variable declaration or constant, expression, assignment, operator, function invocation, and undeclared variable reference. When we encounter a start of a local block, we might find a set of declared variables defined locally for that block in the parse tree (line 17). We add all these local variables to DV when we enter the local block. We then recursively call the procedure `Traverse` to visit the children of the local block's start node in the parse tree (line 18). We remove these local variables from DV when we exit the local block (line 19). If the local block is a switch-case block, we update RV with "switch-case condition" (cf. Table 2) relationship of expressions in switch statement and case statement (lines 14–16). For example, in Fig. 2, we assign "switch-case condition" relation to variable x and constant 1 and store it in RV.

When we visit a variable declaration or constant node, we update our list of declared variables DV (lines 22–24). When we visit a reference to an undeclared variable, we update UV (lines 26–27). If the undeclared variable is an attribute of an object, we also update UT, including updating the attribute for the object (lines 28–30). When we visit an expression node, we add it to EX and perform `Traverse` procedure recursively (lines 33–37).

When we visit an assignment node, we update RV with "assignment" relationship of the return value of expressions at left and right sides (line 39). Next, the left side and right side of the assignment will be traversed recursively (lines 40–41). For

Algorithm 2 Traverse procedure

Traverse(RPT, DV, UV, UF, UT, EX, RV)
Inputs:
RPT: Root of the parse tree of a code fragment
DV: List of declared variables and constants
UV: List of undeclared/undefined variables, arguments, return values
UF: List of undefined functions
UT: List of undefined types
EX: List of expressions
RV: List of relations of variables and expressions
Outputs: DV, UV, UF, UT, EX, RV
Method:
for each child c of RPT **do**
  if c is a start of a local block **then**
    if c is a switch case condition block **then**
      Add into RV the type-equivalence relations between the expressions in the switch statement and the expressions in the case statements based on heuristic in Table 2
    end if
    Add locally declared variables into DV
    Traverse(c, DV, UV, UF, UT, RV)
    Remove locally declared variables from DV
    Continue
  end if
  if c is a declared variable or a constant **then**
    Add into DV the variable/constant with its type
    Continue
  end if
  if c is an undeclared variable reference **then**
    Add into UV this new variable
    if c is an attribute of an object **then**
      Add into UT this new type with its member
    end if
    Continue
  end if
  if c is an expression **then**
    Add into EX this new expression
    Traverse(c's left side, DV, UV, UF, UT, EX, RV)
    Continue
  end if
  if c is an assignment **then**
    Add into RV the type-equivalence relations between the expressions of c's left side and c's right side based on heuristic in Table 2
    Traverse(c's left side, DV, UV, UF, UT, EX, RV)
    Traverse(c's right side, DV, UV, UF, UT, EX, RV)
    Continue
  end if
  if c is an operator **then**
    Add into RV the type-equivalence relations between the expressions connected with the operator based on heuristic in Table 2
    Traverse(c's left side, DV, UV, UF, UT, EX, RV)
    Traverse(c's right side, DV, UV, UF, UT, EX, RV)
    Continue
  end if
  if c is a function invocation **then**
    Update UF
    for each argument of its invocation a of c **do**
      Add into RV the type-equivalence relations between the argument expression a and its corresponding parameter of c based on heuristic in Table 2
      Traverse(a, DV, UV, UF, UT, EX, RV)
    end for
    Add into RV the type-equivalence relations between the return expression a and of c's return type based on heuristic in Table 2
    Traverse(c's return, DV, UV, UF, UT, EX, RV)
    if c is called from an object **then**
      Update UT
    end if
    Continue
  end if
  Traverse(c, DV, UV, UF, UT, EX, RV)
end for

```
if(tag->check(a,b,c)){
    tag->a = a;
    b = 0;
    c = b + tag->getSize();
    tag1 = tag;
    d = tag1->getSize();
}else{
    return a;
}
```

Fig. 3 An example code fragment

example, for an assignment $c = b + tag \rightarrow getSize()$ in Fig. 3, we apply the "assignment" relationship to variable c and expression $b + tag \rightarrow getSize()$ and update it to RV. After the `Traverse` procedure, with this information, we can infer c and $b + tag \rightarrow getSize()$ have the same type (see step 6 in Table 4). When we visit an operator node, similarly to assignment nodes, we update RV with "operator" relationship of expressions connected with the operator and call the procedure `Traverse` to visit the two expression nodes recursively (lines 45–47). For example, operator node $b + tag \rightarrow getSize()$ in above example, we assign "operator" relationship to the variable b and the expression $tag \rightarrow getSize()$ and store it in RV and used it for inference (see step 4 in Table 4).

When we visit a function invocation node, we also consider the arguments and return values of the function invocations. First, we update RV with "function definition and invocation" relationship of expressions in argument nodes and the parameter of a function (line 52). Second, we invoke the procedure `Traverse` to visit the argument nodes (line 53). We deal with the return value of the function in the same way as argument nodes (lines 55–56). For example, for $tag \rightarrow check(a, b, c)$, variable a and argument $check_{arg1}$[3] are assigned the same type based on the "function definition and invocation" heuristics in Table 2 (see step 1 in Table 4). Furthermore, if the function is called from an object, we also add the object variable to UV, update the list of undefined types (UT), and revise the set of member functions of the object (lines 57–59). For example, for $tag \rightarrow getSize()$, we add tag to UT as TAG,[4] meanwhile we add the function $getSize()$ as TAG's member function. We also assign "function definition and invocation" relationship to the expression $tag \rightarrow getSize()$ and the return value of function $TAG \rightarrow getSize()$ and update it to RV.

Example We use an example to illustrate how the overall code extension algorithm works. Consider a code fragment shown in Fig. 3.

We would like to extend the code fragment and make it compile. First, we generate a parse tree for it and traverse the tree to find the undeclared/undefined types, functions, variables, expressions, and type-equivalence relations. We present the lists of undefined types (UT), undefined functions (UF), undefined variables (UV), and expressions (EX) that can be found from the example code fragment in Table 3.

We notice that tag is an object, thus we create a class named TAG and it has a as an attribute and two methods $check$ and $getSize$. We simply name a type by using its corresponding variable's name in uppercase letters. We deal with tag1 in the same way.

[3] First argument of function $tag \rightarrow check()$.

[4] We simply name a type by using its corresponding variable name in uppercase letters.

Table 3 Lists UT, UF, UV, and EX for the code fragment in Fig. 3

List name	Items in list
UT	TAG[a], $TAG1$
UF	$TAG \rightarrow check()$[b], $TAG \rightarrow getSize()$, $TAG1 \rightarrow getSize()$
UV	$tag, a, b, c, d, tag \rightarrow a, tag \rightarrow check()$, $TAG \rightarrow check_{return}$[c], $check_{arg1}$[d], $check_{arg2}, check_{arg3}, TAG \rightarrow getSize_{return}, TAG1 \rightarrow getSize_{return}$
EX	$tag \rightarrow getSize(), tag1 \rightarrow getSize(), b + tag \rightarrow getSize()$

[a] The signature of the type of variable tag
[b] The member function of TAG
[c] The return value of $TAG \rightarrow check()$
[d] The first argument of $TAG \rightarrow check()$

If during type inference, we find two objects are the same, we merge the corresponding types into one, and keep the attribute (i.e., signature and type) and member functions (i.e., signature, types of arguments and return value) of the new type consistent with older ones. For example, in step 7 in Table 5, we infer that the types of tag and $tag1$ are the same. We merge the two corresponding classes and make sure the types of function $getSize()$ in both of them are consistent. In this case, the type of return values $TAG \rightarrow getSize_{return}$ and $TAG1 \rightarrow getSize_{return}$ is the same. Next, we infer the types of the variables, arguments, and return values.

The inference process proceeds following the steps listed in Tables 4 and 5. Initially, each variable, arguments, and return values either declared or undeclared has its own mapping. So far, we only know the type of tag, $tag1$, and 0. In step 1, the return value $TAG \rightarrow check_{return}$ and expression $tag \rightarrow check()$, a and $check_{arg1}$, b and $check_{arg2}$, and c and $check_{arg3}$ are merged by following function definition and invocation heuristic (see Table 2). In step 2, the mappings for a and $tag \rightarrow a$ are merged following the assignment heuristic. However, their types are still unknown since the types of a and $tag \rightarrow a$ are both unknown. The same heuristic is applied in step 3. Next, the function definition and invocation heuristic is applied. In step 5, the operator heuristic is applied and we merge b and $tag \rightarrow getSize()$. The assignment heuristic is applied again in step 6 and 7. In the following step, the function definition and invocation heuristic is applied. Next, after the final application of the assignment heuristic, we are able to infer the types of $d, b, check(arg2), checkarg3, 0, c, tag \rightarrow getSize()$, $TAG \rightarrow getSize_{return}$,$tag1 \rightarrow getSize()$ are all int since the type of 0 is int. We can not merge any other mappings. Finally, we replace all unknown types with int.

After the above steps, we have the needed information to add variable declarations, a new class, and other needed pieces of code. The code fragment (in gray background) is extended to the compilable code (excluding #*include* part) in Fig. 4.

Finally, we feed the extended code to CodeSurfer and get a PDG. We then remove some nodes from the PDG that correspond to the added code and only keep those that correspond to the input code fragment.

5 Query generation engine

In this section, we present how we find commonalities among multiple PDGs generated from a set of example code fragments, and convert these commonalities into

Table 4 Illustration of our type inference process: part I (steps 1–5)

Step	Mappings	Inference heuristic
1	$\{tag\} \mapsto TAG$, $\{a, check_{arg1}\} \mapsto$ Unknown $\{tag \rightarrow a\} \mapsto$ Unknown, $\{b, check_{arg2}\} \mapsto$ Unknown, $\{d\} \mapsto$ Unknown, $\{c, check_{arg3}\} \mapsto$ Unknown, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown, $\{0\} \mapsto int$, $\{tag1\} \mapsto TAG1$, $\{tag \rightarrow getSize()\} \mapsto$ Unknown, $\{tag1 \rightarrow getSize()\} \mapsto$ Unknown, $\{TAG \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{b + tag \rightarrow getSize()\} \mapsto$ Unknown	Function definition and invocation (tag→check(a,b,c))
2	$\{tag\} \mapsto TAG$, $\{check_{arg1}, a, tag \rightarrow a\} \mapsto$ Unknown, $\{b, check_{arg2}\} \mapsto$ Unknown, $\{d\} \mapsto$ Unknown, $\{c, check_{arg2}\} \mapsto$ Unknown, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown, $\{0\} \mapsto int$, $\{tag1\} \mapsto TAG1$, $\{tag \rightarrow getSize()\} \mapsto$ Unknown, $\{tag1 \rightarrow getSize()\} \mapsto$ Unknown, $\{TAG \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{b + tag \rightarrow getSize()\} \mapsto$ Unknown	Assignment (tag→a = a)
3	$\{tag\} \mapsto TAG$, $\{a, check_{arg1}, tag \rightarrow a\} \mapsto$ Unknown, $\{b, check_{arg2}, 0\} \mapsto int$, $\{d\} \mapsto$ Unknown, $\{c, check_{arg3}\} \mapsto$ Unknown, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown, $\{tag1\} \mapsto TAG1$, $\{tag \rightarrow getSize()\} \mapsto$ Unknown, $\{tag1 \rightarrow getSize()\} \mapsto$ Unknown, $\{TAG \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{b + tag \rightarrow getSize()\} \mapsto$ Unknown	Assignment (b = 0)
4	$\{tag\} \mapsto TAG$, $\{a, check_{arg1}, tag \rightarrow a\} \mapsto$ Unknown, $\{b, check_{arg2}, 0\} \mapsto int$, $\{tag \rightarrow getSize()$, $TAG \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{d\} \mapsto$ Unknown, $\{c, check_{arg3}\} \mapsto$ Unknown, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown, $\{tag1\} \mapsto TAG1$, $\{tag1 \rightarrow getSize()\} \mapsto$ Unknown, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{b + tag \rightarrow getSize()\} \mapsto$ Unknown	Function definition and invocation (tag→getSize())
5	$\{tag\} \mapsto TAG$, $\{a, check_{arg1}, tag \rightarrow a\} \mapsto$ Unknown, $\{b, check_{arg2}, 0, tag \rightarrow getSize()$, $TAG \rightarrow getSize_{return}\} \mapsto int$, $\{d\} \mapsto$ Unknown, $\{c, check_{arg3}\} \mapsto$ Unknown, $\{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown, $\{tag1\} \mapsto TAG1$, $\{tag1 \rightarrow getSize()\} \mapsto$ Unknown, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{b + tag \rightarrow getSize()\} \mapsto$ Unknown	Operator (b + tag→getSize())

a dependency query. As Fig. 1 shows, there are three steps in our query generation process:

1. Mine simple subgraphs from a set of PDGs
2. Recover text information for each node in the common subgraphs

Table 5 Illustration of our type inference process: part II (steps 6–9)

Step	Mappings	Inference heuristic
6	$\{tag\} \mapsto TAG, \{a, check_{arg1}, tag \rightarrow a\} \mapsto$ Unknown, $\{check_{arg2}, check_{arg3}, b, 0, c, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}, b + tag \rightarrow getSize()\} \mapsto int, \{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown, $\{d\} \mapsto$ Unknown, $\{tag1\} \mapsto TAG1, \{tag1 \rightarrow getSize()\} \mapsto$ Unknown, $\{TAG1 \rightarrow getSize_{return}\} \mapsto$ Unknown	Assignment (c = b + tag→getSize())
7	$\{tag, tag1\} \mapsto TAG, \{a, check_{arg1}, tag \rightarrow a\} \mapsto$ Unknown, $\{b, check_{arg2}, check_{arg3}, 0, c, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}, b + tag \rightarrow getSize()\} \mapsto int, \{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown, $\{tag1 \rightarrow getSize(), TAG \rightarrow getSize_{return}\} \mapsto$ Unknown, $\{d\} \mapsto$ Unknown	Assignment (tag1 = tag)
8	$\{tag, tag1\} \mapsto TAG, \{a, check_{arg1}, tag \rightarrow a\} \mapsto$ Unknown, $\{b, check_{arg2}, check_{arg3}, 0, c, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}, tag1 \rightarrow getSize(), b + tag \rightarrow getSize()\} \mapsto int, \{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown, $\{d\} \mapsto$ Unknown	Function definition and invocation (tag→getSize())
9	$\{tag, tag1\} \mapsto TAG, \{a, check_{arg1}, tag \rightarrow a\} \mapsto$ Unknown, $\{d, b, check_{arg2}, check_{arg3}, 0, c, tag \rightarrow getSize(), TAG \rightarrow getSize_{return}, tag1 \rightarrow getSize(), b + tag \rightarrow getSize()\} \mapsto int, \{TAG \rightarrow check_{return}, tag \rightarrow check()\} \mapsto$ Unknown	Assignment (d = tag→getSize())

```
class test{
    int testmain(){
        int a, b, c, d;
        TAG* tag, tag1;
    if(tag->check(a,b,c)){
        tag->a = a;
        b = 0;
        c = b + tag->getSize();
        tag1 = tag;
        d = tag1->getSize();
    }else{
        return a;
    }
        return 0;
    }
     static void main(){
        test *t = new test();
        t->testmain();
     }
}

class TAG{
    public int check(int a0, int a1, int a2){}
    public int getSize(){}
    public int a;
    TAG(){}
}
```

Fig. 4 Compilable code extended from a code snippet

3. Construct a query from text-enriched subgraphs

We describe the first, second, and third steps in Sects. 5.1, 5.2, and 5.3 respectively.

5.1 Mine simple maximal common subgraph

First, we just focus on the node and edge types, i.e., $ntype$ and $etype$. We convert each PDG G into their *simple* graph representation G^{notext}. Next we mine for *maximal* subgraphs that appear on *all* G^{notext}. We get one such simple maximal common subgraph. We mine this maximal common subgraph using an existing graph mining tool called Gaston.[5]

5.2 Recover textual information

At this step, we have a simple subgraph S^{notext} that is common for the set of all PDGs $PDGSet$. Our next job is to recover textual information for each node of this subgraph. The textual information of a node captures semantic and structural information. If the $ntype$ of a node is function, its textual information represents the name of the function. If the $ntype$ of a node is control statement, its textual information represents the type of the control statement (e.g., if, while, etc). We extract the textual information to create "contain *string*" constraints for a node in a DQL query. This makes the query more specific and accurate.

Note that this step is not trivial since each node in S^{notext} can match multiple nodes containing different textual information in a particular PDG, and thus there is a need to choose the best matching node. The best matching nodes among different PDGs may contain different textual information, and thus there is a need to unify these textual information together by removing peculiar information specific to a matching node of a PDG.

High-level description The algorithm performing textual content recovery is shown in Algorithm 3. The idea is to perform graph matching operation. We match each simple subgraph S^{notext} to each PDG G_j in `PDGSet` based on the labels $ntype$ and $etype$ (we ignore the textual labels) (lines 8–11). The graph matching operation returns a set of candidate nodes in G_j that potentially match each node of S_i^{notext}. Notation-wise, given a node n in S_i^{notext}, we denote its set of candidate nodes in G_j as $Cand_n^{G_j}$. We select one representative candidate per PDG and extract its textual label (lines 12–28). A candidate node of a PDG can only represent one node in S^{notext}. Thus we delete the representative nodes from the sets of candidate nodes of other nodes in S^{notext} (line 29). We then unify this set of labels to recover the label for node n (line 30). We elaborate our approach to select representative candidates and unify text labels in the following paragraphs.

[5] http://www.liacs.nl/~snijssen/gaston/.

Algorithm 3 Textual information recovery procedure

1: **recoverText(**$PDGSet$**,** G_{sub}^{notext}**,**$Candidate$**)**
2: **Input:**
3: $PDGSet$: A set of PDGs
4: S^{notext}: A common subgraph of PDGs in $PDGSet$ ignoring $text$ labels
5: **Output:** Text enriched S^{notext} (i.e., S)
6: **Method:**
7: Let $Cand = Pool = \{\}$
8: **for** each G_i in $PDGSet$ **do**
9: Perform a graph matching operation between S^{notext} and G_i
10: Let $Cand_n^{G_i}$ stores all candidates for node n in S^{notext}
11: **end for**
12: **for** each node n in S^{notext} **do**
13: Let $Rep = \{\}$ // Representative nodes from each PDGs
14: **if** $\forall G_i \in PDGSet.|Cand_n^{G_i}| = 1$ **then**
15: $Rep = \{c | \exists_{G_i \in PDGSet}.c \in Cand_n^{G_i}\}$
16: **else**
17: Let $Reference = \{\}$ //First representative node
18: Let $Others = \{\}$ //Candidates from other PDGs
19: **if** $\exists G_i \in PDGSet.|Cand_n^{G_i}| = 1$ **then**
20: $Reference = \{c \in Cand_n^{G_i} | \, |Cand_n^{G_i}| = 1\}$
21: $Others = \{Cand_n^{G_i} | G_i \in PDGSet \wedge |Cand_n^{G_i}| > 1\}$
22: **else**
23: $Reference$ = an arbitrary $Cand_n^{G_i}$
24: $Others = \{Cand_n^{G_i} | G_i \in PDGSet\} \setminus Reference$
25: Delete all but (random) node node in $Reference$
26: **end if**
27: Let Rep = selectRepCand($Reference$,$Others$)
28: **end if**
29: Remove nodes in Rep from $Cand_{n'}^{G_i}$, $n' \neq n$ and $G_i \in PDGSet$
30: $n.Text$ = Unify text labels of nodes in Rep
31: **end for**

Selecting representative candidates We would like to select one representative candidate per PDG for a node n. If all candidate sets are of size 1, then we simply take all candidate nodes as the representative nodes (lines 14–15). Otherwise, we would like to pick representative nodes such that they are similar to one another. The rationale for this would be explained further by an example described in Sect. 5.4. To pick similar representative nodes, first we pick a set of reference nodes (lines 20, 23, and 25). If there are candidate set of size 1, we take the nodes in these sets as the reference nodes (lines 19–20). Otherwise, we pick an arbitrary candidate node as a reference node (lines 23, 25). Next we pick representative nodes from other PDGs that are similar to the reference nodes (line 27). Algorithm 4 describe this last step in more detail. In Algorithm 4, we first take the input reference nodes as the representative nodes (line 7). Next, we visit each candidate node set of the other PDGs and pick a node that is the most similar to the selected representative nodes (lines 8–12).

At line 9 of Algorithm 4, we need to measure the similarity between nodes. We measure the similarity between nodes by considering their text labels. Each node is represented by one vector which captures its textual information. Each element of the vector corresponds to a word token that appears in the corresponding node's text label. In our dataset, the sizes of these vectors are from 1 to 11 with a mean of 4.2.

Algorithm 4 Selection of representative candidates

1: **selectRepCand**($Reference$, $Others$)
2: **Input:**
3: $Reference$: Selected representative candidates
4: $Others$: Candidates from other PDGs
5: **Output:** All representative candidates
6: **Method:**
7: Let $Rep = \{n | n \in Reference\}$
8: **for** each set $Cand_n^{G_i}$ in $Others$ **do**
9: Select a node n' in $Cand_n^{G_i}$ which is the most similar to all nodes in $Reference$
10: $Rep = Rep \bigcup \{n'\}$
11: $Others = Others \setminus Cand_n^{G_i}$
12: **end for**
13: **return** Rep

The vocabulary of these vectors contains all words that appear in the textual labels of all nodes in the PDGs. We then weigh each word by using a TF-IDF weighting scheme (Manning et al. 2008). Here, the term frequency (TF) of a word refers to the number of times the word appear in the text label. Inverse document frequency (IDF) refers to 1/DF (document frequency), where DF is the number of representative nodes with text labels containing the word. Each node is then represented as a vector of weights. Similarity between two nodes n_1 and n_2 could then be measured by the similarity between their corresponding weight vectors v_1 and v_2. We use standard cosine similarity for this purpose (Manning et al. 2008):

$$cos(v_1, v_2) = \frac{\sum_{i \in (v_1 \cup v_2)} v_1[i] \times v_2[i]}{\sqrt{\sum_{i \in v_1} v_1[i]^2} \times \sqrt{\sum_{i \in v_2} v_2[i]^2}}$$

Finally, we take an average on the similarity scores of the node n in $Cand_n^{G_i}$ and each node in $Reference$ and select the node with the highest score as the most similar one.

Unifying textual labels For this process, for a node n in a subgraph S^{notext}, we have as input a set of representative nodes with their text labels. Our goal is to create a single unified textual label for n (line 30). To do this, we perform the following steps:

1. For each representative node text label, we pre-process it as follows: If $n.ntype$ = function call, arguments (including parentheses) are removed and the name of the function is kept. If $n.ntype$ = expression, only keep the right side of the expression. For all other types, all text is kept.
2. Get the longest common text from the pre-processed text labels. In this step, we find the longest consecutive sequence of characters (Gusfield 1997) (we don't consider case-sensitivity) that appears in all the text labels.
3. Split the resultant text and remove special symbols. We first split the resultant text with white space. We then perform Camel Case splitting on each token to split the consecutive identifier (Antoniol et al. 2002). For example, we split "getString" to "get" and "string". We also split each token with some special symbols (i.e., operators, number). At last, we remove some special symbols (i.e., operators, number).

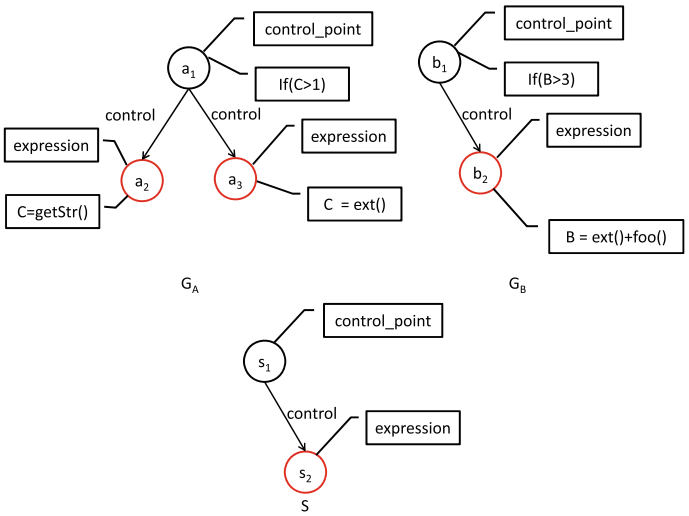

Fig. 5 Query generation example

Example Assume two representative nodes n_1 and n_2 with labels "a = getExtString (para1) + 123 + var12" and "b = ExtString(para2) + 123 + var13" respectively. We first perform step 1 on these two nodes. This step removes "para1" from n_1 and "para2" for n_2 as their ntype = function call. We also remove "a =" and "b =" as we only keep the right hand side of an expression. After step 1, we get "getExtString + 123 + var12" and "ExtString + 123 + var13" for n_1 and n_2, respectively. Following step 2, we get the longest common text "ExtString + 123 + var1" from "getExtString + 123 + var12" and "ExtString + 123 + var13". At the last step, we split the longest common text "ExtString + 123 + var1" to "ext", "string", "+", "123", "+", "var", and "1" and then remove "+", "123" and "1". Finally, we get "ext", "string" and "var".

5.3 Construct query from enriched subgraphs

For this step, we have as input a text-enriched common maximal subgraph S. We create a dependency query (expressed in DQL which is described in Sect. 2) from this subgraph by following these steps:

1. Output the nodes in S and their types as *node declarations* of the query
2. Output the edges in S as *relationship descriptions* of the query
3. Output the text labels of nodes in S as *node descriptions* of the query
4. Identify all nodes defined in the query as *target* nodes

5.4 Example

We use Fig. 5 to illustrate the query generation process. PDGs G_A and G_B are the inputs and a query written in DQL is the output. First, we perform graph mining on G_A and G_B to get a simple maximal common subgraph which is S.

Table 6 Programs analyzed in this study

Name	Description	Size (KLOC)	#Versions
Apache Http Server	HTTP Web Server	264.5	12
Apache Subversion	Open source version control system	483.5	8
Inkscape	Open source vector graphics editor	458.2	9
Libmpeg2	Library for decoding MPEG-2 and MPEG-1 video streams	37.3	3

Next, we run Algorithm 3 on G_A, G_B, and S to recover the textual information for the nodes in S. For node s_1, its representative nodes in G_A and G_B can be easily identified as there is only one candidate node for each PDG; They are nodes a_1 and b_1. After we unify the text labels of a_1 and b_1, we get "if" as the text label of s_1. For node s_2, there are two candidate nodes in G_A (i.e., nodes a_2 and a_3), and one candidate node in G_B (i.e., node b_2). For this case, our algorithm first selects node b_2 as a representative node. It then finds the node in $\{a_2,a_3\}$ which is the most similar to b_2. It selects a_3 as it is more similar to b_2 than a_2. After we unify the text labels of a_3 and b_2, we obtain "ext" by getting the longest common text "ext()" and then removing the parentheses.

Finally, we convert the text-enriched S to a dependency query expressed in DQL. The resultant query is as follows:

Node declarations: *ctrlPoint A, func B*;
Node descriptions: *A contains if, B contains ext*;
Relationship descriptions: *A oneStep controls B*;
Targets: *A,B*;

6 Evaluation

6.1 Experimental settings

To evaluate our proposed approach, we extract code search scenarios from repositories of real software systems. We are especially interested on recovering real-life changes that need to be applied to various locations in a code base. We use these changes to simulate code search scenarios. Developers might know some of these locations but would like to know other relevant locations. Code search could help developers to find these other locations. This experimental setting follows the setting described in Wang et al. (2010). The code bases that we use for our experiments are from four realistic programs written in C and C++ namely Apache Http Server (12 versions), Inkscape (9 versions), Apache Subversion (8 versions), and Libmpeg (3 versions), which have undergone at least thirteen years of continuous development, improvement, and optimization. The details of these programs are shown in Table 6.

In Wang et al. (2013), we perform an empirical study on widespread changes. To identify widespread changes, we look for commits that touch many files and these files are modified in a similar way structurally and semantically. Files involved in a

widespread change are modified for the same purpose, e.g., fixing the same bug, etc. To check whether the files are modified in a semantically similar way, we manually inspect the files to find whether they are indeed changed for the same purpose. We follow this procedure to identify widespread changes on the 4 programs that we use in this study. Applying the procedure, we get 47 widespread changes. Each widespread change involves 5–53 code locations, with an average of 10 locations. Let us refer to the code requiring change at each location as a fragment. In total, we have 478 fragments. Each fragment is of 2–20 lines of code. To simulate code search scenarios, for each widespread change, we randomly pick two fragments as the input to a code search task. We test the effectiveness of the code search task using the remaining fragments, i.e., the remaining fragments become the gold set or standard. Since we have 47 widespread changes, we simulate 47 code search tasks.

We implement our AutoQuery approach in Python and Java. AutoQuery converts code examples into dependency queries. We use the approach by Wang et al. (2010) as the backend code search engine that would process the dependency queries and return relevant pieces of code. We use a desktop with an Intel Core i5 3.2GHz CPU installed with 4GiB of memory and 2TiB of hard disks to run experiments.

We compare the performance of AutoQuery with manually constructed queries (we refer to them as UserQuery). We perform a user study involving 10 participants and the 47 code search tasks. Among the 10 participants, 9 of them are PhD students who have at least two years of C and C++ programming experience. One of them is a professional software engineer who has three years of C and C++ programming experience. All of them know are familiar with Program Dependency Graph (PDG)—many of them have taken a course on program analysis.

We give each participant tasks in the following format: given a set of code fragments, generate a DQL query that can capture the code fragments in this set. Each participant is assigned four or five tasks. For each task, a participant is given two fragments (i.e., code examples) along with the corresponding PDGs of the fragments. Note that we give users PDGs to ease the tasks for users. A participant needs to look at the code fragments as well as their corresponding PDGs, and construct a dependency query expressed in DQL to find other similar codes. We record the queries users created and the time each of them takes to complete the construction of a query. Before users start with the tasks, each of them is given a 20 min tutorial about dependence-based code search and DQL. They are also given a 10 min exercise to construct a query from simple code fragments. These tutorial and exercise are meant to familiarize users with the tasks.

6.2 Experiment results

We aim to answer the following research questions:

- RQ1 Can AutoQuery generate good dependency queries that can retrieve relevant search results?
- RQ2 Can AutoQuery perform comparably well as developers in constructing good dependency queries?
- RQ3 Can AutoQuery improve the time it takes to construct queries?

The first research question investigates the overall effectiveness of our proposed approach. The second research question investigates the effectiveness of AutoQuery as compared to manually constructed queries (UserQuery). The third research question investigates the efficiency of our approach as compared to UserQuery.

6.2.1 RQ1: effectiveness of AutoQuery

To answer this research question, for each of the 47 tasks, we run the dependency query generated by AutoQuery on the dependence-based code search engine of Wang et al. (2010). We count the number of code fragments in the gold set that are retrieved. Based on this, we compute the average precision, recall, and F-measure which are standard information retrieval measures (Manning et al. 2008). For a given task, precision, recall, and F-measures are defined as follows:

$$precision = \frac{\#\ retrieved\ code\ fragments\ in\ the\ gold\ set}{\#\ retrieved\ code\ fragments}$$

$$recall = \frac{\#\ retrieved\ code\ fragments\ in\ the\ gold\ set}{\#\ code\ fragments\ in\ the\ gold\ set}$$

$$F\text{-}measure = \frac{2 \times precision \times recall}{precision + recall}$$

F-measure is the harmonic mean of precision and recall and it is often used as a summary measure. It quantifies if an increase in precision outweighs a decrease in recall (and vice versa). Table 7 shows the result. The average precision, recall, and F-measure for the 47 tasks are 68.4, 72.1, and 70.2 %, respectively. For 25 automatically constructed queries, we are able to find all fragments in the gold set (recall = 1). For 21 automatically constructed queries, all retrieved fragments are in the gold set (precision = 1). For 12 automatically constructed queries, all fragments in the gold set are retrieved and all retrieved fragments are in the gold set (F-measure = 1).

In the default setting, for each task, AutoQuery is given two code fragments to generate a query. We would like to test the effectiveness of AutoQuery with different numbers of code fragments as input. In this experiment, we give k randomly selected code fragments to AutoQuery. Since the number of relevant code fragments for each task ranges from 5 to 53, we vary the value of k in set {1, 2, 3, 4} and measure effectiveness in terms of recall, precision and F-measure. In Fig. 6, we present the effectiveness of AutoQuery for different numbers of code fragments. We notice that the recall value increases as the number of code fragments increases, and the precision value decreases as the number of code fragments increases. As more code fragments are used to generate a query, a more general query is generated. Searching using a more general query returns a larger number of code fragments. As the number of code fragments increases, more false positives are introduced, which leads to a lower precision value. On other hand, as the number of code fragments increases, less false negatives are introduced, which leads to a higher recall value. In terms of F-measure, the harmonic mean of precision and recall, AutoQuery performs the best when three code fragments are given.

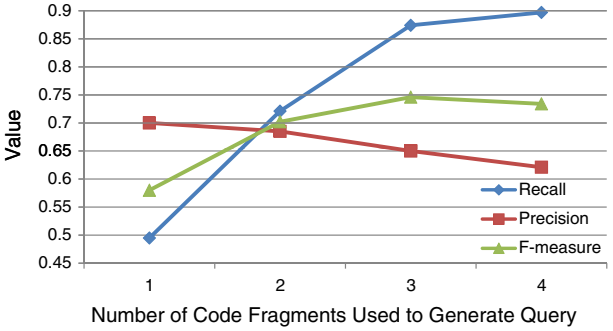

Fig. 6 Results (*y* axis) vary as the number of code fragments (*x* axis) increase

Table 7 AutoQuery versus UserQuery: precision, recall, and F-measure

	AutoQuery	UserQuery
Precision	0.684	0.584
Recall	0.721	0.767
F-measure	0.702	0.664

We also perform a fivefold cross validation like experiment to evaluate AutoQuery. We randomly split 1/5 of the code fragments in each gold set of our code search tasks into 5 buckets. We then perform 5 iterations. In each iteration, we use fragments in one of the buckets to generate a query and use the other four buckets to test the effectiveness of AutoQuery. For this experiment, AutoQuery achieves a recall, precision, and F-measure of 0.861, 0.64, and 0.734, respectively.

6.2.2 RQ2: AutoQuery versus UserQuery

To answer this research question, we compare the results of AutoQuery with those of UserQuery. For each of the 47 code search tasks, we use the same pairs of code fragments as input to AutoQuery and UserQuery.

As shown in Table 7, the precision, recall, and F-measure of UserQuery are 58.4, 76.7, and 66.4 %, respectively. In general, users' queries are more general (contain less constraints), while queries generated by AutoQuery are more specific (contain more constraints). This makes the results obtained from the queries generated by AutoQuery more precise (resulting in higher precision). However, some relevant results are missed (resulting in lower recall) because of the more specific queries. It is a trade-off between precision and recall. To measure the trade-off between precision and recall, we also compute F-measure which is a summary measure of precision and recall. In terms of F-measure, AutoQuery achieves a 5.7 % improvement over UserQuery.

We also perform a Wilcoxon signed-rank test Wilcoxon (1945) to check the significance of the differences in the performance of AutoQuery and UserQuery measured in terms of recall, precision and F-measure. The results show that the differences in terms of F-measure (p value = 0.49) and recall (p value = 0.17) are not significant, while the difference in terms of precision is significant (p value = 0.02). This shows that

Table 8 AutoQuery versus UserQuery: number of winning queries

	AutoQuery wins	*UserQuery* wins
Precision	18	7
Recall	4	5
F-measure	16	9

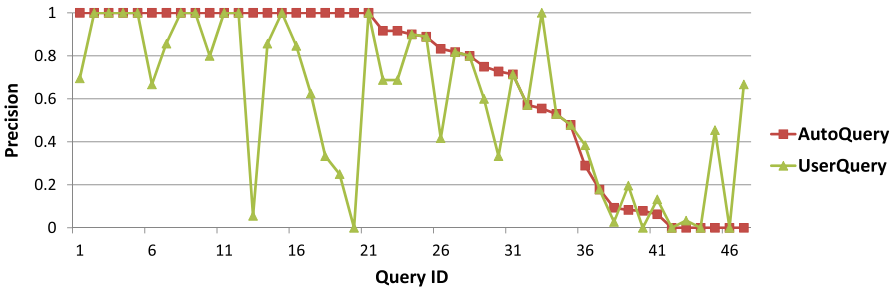

Fig. 7 AutoQuery versus UserQuery: precision per task

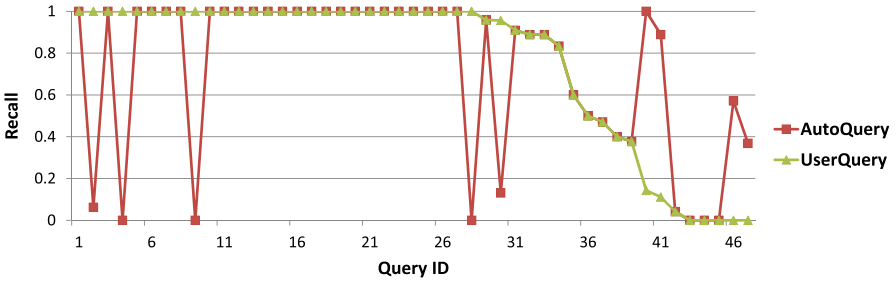

Fig. 8 AutoQuery versus UserQuery: recall per task

AutoQuery is comparably as good as developers in constructing dependency queries in terms of recall and F-measure, and the improvement in terms of precision achieved by AutoQuery in constructing dependency queries over developers is statistically significant.

Table 8 presents the number of tasks where AutoQuery outperforms UserQuery (and vice versa). In terms of precision, AutoQuery wins on 18 queries and loses on 7 queries. In terms or recall, AutoQuery wins on 4 queries and loses on 5 queries. In terms of F-measure, which is the harmonic mean of precision and recall, AutoQuery wins on 16 queries and loses on 9 queries. AutoQuery and UserQuery do not produce the exact same search results for the remaining queries, however their precision (or recall, or F-measure) values for these queries are the same. Figures 7, 8, and 9 present precision, recall, and F-measure of AutoQuery and UserQuery for each of the 47 tasks. The above results show that AutoQuery is comparably as good as developers in constructing dependency queries.

We can notice that for Query 20 in Fig. 7, AutoQuery achieves a precision value of 1, while UserQuery can not find any correct results, which leads to poor precision. We manually check the query generated by the user, and we notice that the user

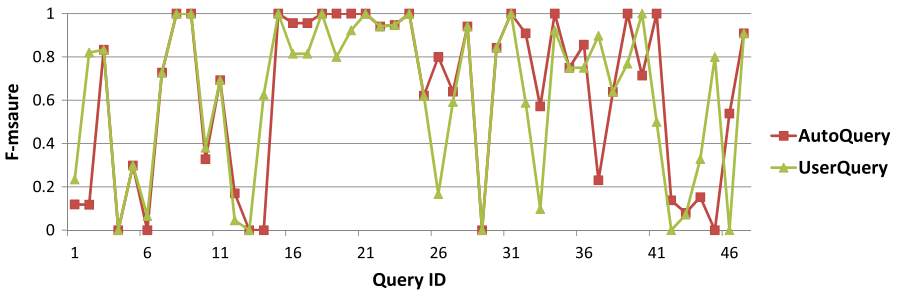

Fig. 9 AutoQuery vs. UserQuery: F-measure per task

misses important constraints which make the query ineffective to find relevant code fragments. The precisions of queries formed by users for tasks 13, 18–21 are low. We have checked the queries we got from the participants that perform tasks 13, 18–21. We find that they are generated by participants that are able to create relatively good queries for other tasks. In Fig. 8, for some queries (e.g., Query 41) UserQuery outperforms AutoQuery. We check the queries generated by both of them, we find that queries generated by users are more general with less constraints, while queries obtained from AutoQuery are more specific with more constraints. More general queries return more results which lead to UserQuery achieving higher recall than AutoQuery. Based on this observation, in the future, to improve the effectiveness of AutoQuery, we plan to extend it by developing a machine learning technique that can remove or weaken some of the generated constraints automatically.

6.2.3 RQ3: efficiency of AutoQuery compared with UserQuery

Static analysis and data mining techniques can take much time and resource to run to completion. Some static analysis and data mining techniques can run for hours. Our approach makes use of both static analysis and data mining. Graph mining in particular can be a time consuming operation. Thus, in this research question we want to investigate whether our approach is efficient enough as compared to the time it takes for developers to manually construct queries. If our approach is slower than the time developers take to manually construct queries, then it might not be practical. Another side goal of this research question is to investigate the effort developers need to construct queries as measured by the time they take to construct them.

To answer this question, we compare the time it takes for AutoQuery to construct queries with that of UserQuery. The total time it takes for AutoQuery to construct the 47 queries is 27.5 s. Thus, the average time per query is 0.6 s which is reasonable short. The total time for developers (UserQuery) to construct the 47 queries is 10,509 s, with an average of 223.6 s. Compared with the time it takes for developers to construct query, our approach is much more efficient. Many developers are likely to be reluctant to use a code search tool if they need to think hard for 3–4 min to construct a query. AutoQuery addresses this concern.

Figure 10 shows the time it takes for AutoQuery and a developer to construct each of the 47 queries. Almost all queries can by constructed by AutoQuery in less

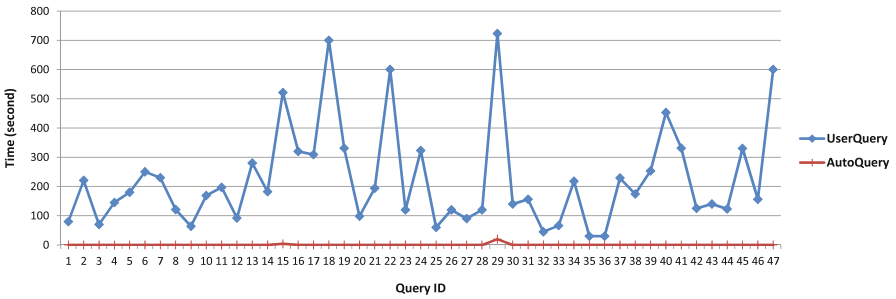

Fig. 10 AutoQuery versus UserQuery: efficiency

than a second except for two queries: one of them costs 3.9 s and another costs 19.8 seconds. For these two most complicated queries, developers spend 521 and 723 s, respectively. From these experiment results, we can see that AutoQuery is able to save much developer time.

In Figs. 11 and 12, we present the code fragments for those two queries, which contain 19 and 10 lines, respectively. Correspondingly, their dependence graphs are bigger and more complex (they have 23 nodes and 10 edges, and 21 nodes and 27 edges, respectively). Thus, for these fragments, because AutoQuery needs to perform graph extraction from the PDGs, whose complexity is dependent on the size of the PDGs (i.e., number of nodes and edges in the PDGs), it takes more time to generate queries. Correspondingly, it is also more difficult for users to formulate queries from these code fragments. One way to alleviate this problem, which we plan to investigate in the future, is to compress the dependence graph by removing some unimportant nodes and edges.

6.2.4 Threats to validity

Threats to internal validity include experimenter bias. There might be subjectivity in the queries that a participant formulates. Similar to past studies, e.g., McMillan et al. (2011), Wang et al. (2012) and Tian et al. (2014), we do not have a large pool of participants such that each query can be formulated by many people. It could be the case that some participants may have formulated very bad queries, which may not be representative of a typical/trained user. Still, we consider 47 widespread changes and take averages, and we believe averaging would help reduce the threats. Also, we have given user study participants a 20 min tutorial and 10 min exercise (the exercises we gave to the users are similar to the actual tasks) to familiarize them with writing queries in DQL. We have also provided helps and hints when users faced difficulties during the tasks. The results that we get may be different if we train them much more.

Threats to external validity relate to the generalizability of our approach. We have investigated 47 widespread changes and experimented on four realistic programs written in C and C++. Admittedly, our programs are just a fraction of the collection of all programs out there. Also, our tasks do not cover all kinds of widespread changes. In the future, we plan to reduce the threats to external validity further by investigating more widespread changes and more programs written in other languages.

```
(1)
    if (!type) {
      GTypeInfo info = {
           sizeof(SPAttributeTableClass),
           0, // base_init
           0, // base_finalize
           (GClassInitFunc)sp_attribute_table_class_init,
           0, // class_finalize
           0, // class_data
           sizeof(SPAttributeTable),
           0, // n_preallocs
           (GInstanceInitFunc)sp_attribute_table_init,
           0 // value_table
       };
       type = g_type_register_static(GTK_TYPE_VBOX,
           "SPAttributeTable",
           &info,
           static_cast<GTypeFlags>(0));
    }
    return type;
(2)
        static GType type = 0;

    if (!type) {
        GTypeInfo info = {
            sizeof(SPCtrlRectClass),
           0, // base_init
           0, // base_finalize
           (GClassInitFunc)sp_ctrlrect_class_init,
           0, // class_finalize
           0, // class_data
           sizeof(CtrlRect),
           0, // n_preallocs
           (GInstanceInitFunc)sp_ctrlrect_init,
           0 // value_table
        };
        type = g_type_register_static(SP_TYPE_CANVAS_ITEM,
           "SPCtrlRect",
            &info,
            static_cast<GTypeFlags>(0));
    }
    return type;
```

Fig. 11 Two complex code fragments—example I

7 Related work

In this section, we describe closely related studies on type inference and code comprehension for incomplete code fragments, code search, program dependence graphs, and graph mining for software engineering.

7.1 Type inference and code comprehension for incomplete code fragments

Studies in the literature have addressed many of the issues related to type inference and code comprehension for incomplete code fragments in various languages. For example, Milner's unification algorithm and variants have been widely used to assign types to program elements and facilitate program comprehension (Baker 1990; O'Callahan and Jackson 1997). Dagenais and Hendren's work (2008) uses heuristics to enable partial type inference and more accurate static analysis for Java programs. They focus

```
(1)
         if (item) {
        ec->shape_knot_holder = sp_item_knot_holder(item, ec->desktop);
        Node *shape_repr;
        shape_repr= SP_OBJECT_REPR(item);
        if (shape_repr) {
            ec->shape_repr = shape_repr;
            sp_repr_ref(shape_repr);
            sp_repr_add_listener(shape_repr, &ec_shape_repr_events, ec);
            sp_repr_synthesize_events(shape_repr, &ec_shape_repr_events, ec);
         }
(2)
    if (item) {
        ec->shape_knot_holder = sp_item_knot_holder(item, ec->desktop);
        Node *shape_repr;
        shape_repr= SP_OBJECT_REPR(item);
        if (shape_repr) {
            ec->shape_repr = shape_repr;
            sp_repr_ref(shape_repr);
            sp_repr_add_listener(shape_repr, &ec_shape_repr_events, ec);
            sp_repr_synthesize_events(shape_repr, &ec_shape_repr_events, ec);
         }
```

Fig. 12 Two complex code fragments—example II

on low error rates during the type inference. Parseweb by Thummalapenta and Xie (2007) also uses partial type inference for Java programs to facilitate the construction of code examples that can convert an object from one type to another. Partial type inference also helps EqMiner Jiang and Su (2009) to make code fragments in C language compilable and executable for the purpose of semantic code clone detection. In this paper, we focus on preserving program dependencies for dependency queries and are less concerned with accurate semantic preservation. We adapt similar, but more lightweight, heuristic-based dataflow analysis and type inference for making C and C++ code fragments compilable so that we can easily construct PDGs for code search.

7.2 Code search

Many code search techniques accept user queries in the form of free text (Chan et al. 2012; McMillan et al. 2011). There are also related studies on feature localization and bug localization that takes a description of a feature or a bug and return source code files that implement the feature or need to be fixed to address the bug (Dit et al. 2013; Wang et al. 2011b). Dit et al. present a systematic literature survey of 89 feature location and code search articles from 25 venues published between November 1992 and February 2011 (Dit et al. 2013). They provide a comprehensive, structured overview of those articles based on various dimensions, such as the types of the analysis techniques (e.g., static, dynamic, textual, historical change analyses), the types of user inputs (e.g., natural language queries, execution scenarios, source code artifacts), the types of derived inputs (e.g., dependence graphs, execution traces), the types of outputs (e.g., code fragments, ranked lists, visualization), among others. The idea of our approach may also be applied to studies that would take source code artifacts and dependence graphs as input.

Source code contains more than identifiers. It also contains data and control dependencies among the identifiers. To leverage dependency relationships among program elements, dependence-based code search technique was proposed by Wang et al. (2010). The approach is further extended by the incorporation of topic modeling to dependence-based code search (Wang et al. 2011a). It has been shown that dependence queries can outperform text queries (Wang et al. 2010). Different from text queries where relationships between terms in the queries are unspecified, in a dependence queries relationships between terms (i.e., program elements) can be specified in terms of control and data dependencies. Although, dependence-based code search has been shown to be accurate if good dependency queries are given, it is not clear if users could construct good dependency queries. Indeed, some dependencies queries require users to visualize dependence relations among program elements of interest and this might be a daunting task to many users. AutoQuery addresses this problem by automatically recovering dependency queries from some code examples.

There are other techniques that accept a code example and returns other similar code examples (Li and Ernst 2012; Lee et al. 2010). These techniques are often based on code clone mining (Jiang et al. 2007; Roy et al. 2009; Kim et al. 2010; Jang et al. 2012). Wang et al. has shown that dependence-based code search could outperform clone-based code search as users could specify dependencies of interest (Wang et al. 2010). With AutoQuery, dependencies of interest could be inferred from a set of program examples. Furthermore, different from those code-clone-based studies, our approach unifies multiple example code snippets into a single query. By analyzing multiple code snippets, AutoQuery can differentiate relevant dependencies that are observed in multiple code snippets from peculiar dependencies that are only observed in an individual code snippet. By producing a human-readable query (rather than directly searching the code base using the common subgraph of the PDGs), AutoQuery allows developers to retain control in the code search process. Developers can modify and tweak the generated query based on their domain knowledge to result in more effective code search.

7.3 Program dependence graph, its construction and usages

Program dependence graphs was proposed by Horwitz and Reps (1992). Data and control dependencies can be detected more accurately with better pointer analysis and string analysis algorithms. Many studies propose new algorithms for pointer analysis (Hardekopf and Lin 2007; Lattner et al. 2007) and string analysis (Ganesh et al. 2011). In this work, we generate program dependence graphs from code fragments. We address the challenge of generating PDGs from non-compilable code fragments.

PDGs have been utilized by many past studies. Komondoor and Horwitz use PDGs for detecting duplicated code, aka. clones (Komondoor and Horwitz 2001). Baah et al. build a probabilistic PDG and use it to localize bugs in programs given a set of failing and correct executions (Baah et al. 2010). In this work, we focus on the usage of PDGs for automatic construction of queries for dependence-based code search.

7.4 Graph mining for software engineering

In this work, we make use of graph mining to capture the commonalities between the PDGs. There are many other studies that also make use of graph mining algorithms.

Tien et al. investigate the use of graph mining for software specification mining Nguyen et al. (2009). They characterize usages of an API as a graph and mine for frequent graphs. Hong et al. create graphs from failing and correct program execution traces Cheng et al. (2009). They then extract discriminative graphs that differentiate failing from correct program executions. Chang et al. use graph mining to detect for implicit programming rules from system dependence graphs and use these rules to find bugs which correspond to violations of the rules Chang et al. (2008). In a latter work, Sun et al. improve the above work by incorporating supervised learning algorithm Sun et al. (2010). In this work, we employ graph mining for a different problem. We also extend existing graph mining algorithm to support a unique graph where each node contains multiple labels: node type information (categorical), and textual content information (text). Past graph mining algorithms, e.g., Yan and Han (2002), only support simple graphs that contain one categorical label per node.

7.5 Program transformation

There are also some studies on program transformation that are related to our work (Meng et al. 2011a, b; Andersen et al. 2012; Andersen and Lawall 2010). Andersen et al. propose generic patch inference (Andersen and Lawall 2010). Their approach take a set of example program transformations and generate a simple patch (aka. generic patch) from it. A patch specifies changes that need to be made given a context. The expressiveness of generic patch is not high though, and thus they extend their study further in Andersen et al. (2012) to infer semantic patch. These studies of Andersen et al. only focus on API usage changes and context information can only be expressed as API method invocations and dependencies among them. Meng et al. (2011a) generalize the work by Andersen et al. to support more than API usage changes. In Meng et al. (2011a), they are only able to generalize from one example. In their later work, Meng et al. are able to generalize from multiple examples (Meng et al. 2013).

In this work, similar to the past studies by Meng et al., we can also handle more than API usage changes. The closest work of Meng et al. with ours is their latest work, i.e., Meng et al. (2013), which generalizes from a set of examples. Meng et al.'s work generalize transformation examples, while we generalize code examples. In their work, Meng et al. find commonalities among transformation examples by running a token-based clone mining algorithm, i.e., CCFinder. Token-based clone mining algorithm treats code as a set of tokens with no dependency relationships among them. In our work, since we are supporting dependence-based code search, we need to capture commonalities in dependency relationships among program elements of interest. While they analyze a set of tokens, in our work, we analyze a set of graphs. We develop a new graph mining solution that is able to handle multi-label graphs with categorical and textual node labels to do this.

8 Conclusion and future work

Searching through a large base of source code is common activities performed by developers. Without the aid of automated code search tools, developers need to tap on their experience to browse relevant source code files and manually find the code fragments that are related to their tasks at hand. This is not only be time consuming but also error-prone. A number of code search tools have been proposed to address this problem. Many of them accept textual descriptions as user queries and return relevant code fragments.

Source code contains not only text but also dependencies. Dependence-based code search tools accept queries expressed as dependency relationships among program elements of interest and return code fragments whose constituent program elements satisfy the dependency relationships. Dependence-based code search tools have been shown to be effective and improve the accuracy of search results than text-based code search tools. However, there is one drawback that potentially hampers the usage of dependence-based code search. It may be hard for users who have no or little knowledge of PDGs to construct queries.

To address this drawback, we propose an automatic approach to construct a query based on common dependency structures and textual information extracted from a set of sample code fragments. We have evaluated our approach with 47 realistic code search tasks on 4 real systems, and show that the automatically constructed dependency queries recover relevant code with a precision, recall, and F-measure of 68.4, 72.1, and 70.2 %, respectively. We have also performed a user study that shows that our automatically constructed queries are comparable to human constructed queries in retrieving relevant codes.

In the future, we plan to experiment with more code search tasks and systems. We also plan to support other programming languages besides C and C++ and to publicly release the tool.

References

Andersen, J., Lawall, J.L.: Generic patch inference. Autom. Softw. Eng. **17**(2), 119–148 (2010)

Andersen, J., Nguyen, A. C., Lo, D., Lawall, J. L., Khoo, S.-C.: Semantic patch inference. In: ASE, pp. 382–385 (2012)

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002)

Baah, G.K., Podgurski, A., Harrold, M.J.: The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans. Softw. Eng. **36**(4), 528–545 (2010)

Baker, H.G.: Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In: ACM Conference on LISP and Functional Programming, pp. 218–226 (1990)

Chan, W.-K., Cheng, H., Lo, D.: Searching connected api subgraph via text phrases. In: SIGSOFT FSE, p. 10 (2012)

Chang, R.-Y., Podgurski, A., Yang, J.: Discovering neglected conditions in software by mining dependence graphs. IEEE Trans. Softw. Eng. **34**(5), 579–596 (2008)

Cheng, H., Lo, D., Zhou, Y., Wang, X., Yan, X.: Identifying bug signatures using discriminative graph mining. In: ISSTA, pp. 141–152 (2009)

Dagenais, B., Hendren, L.: Enabling static analysis for partial java programs. In OOPSLA, pp. 313–328 (2008)

Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. J. Softw. **25**(1), 53–95 (2013)
Dit, B., Revelle, M., Poshyvanyk, D.: Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. Empir. Softw. Eng. **18**(2), 277–309 (2013)
Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: ICSE, pp. 321–330 (2008)
Ganesh, V., Kiezun, A., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a string solver for testing, analysis and vulnerability detection. In: CAV, pp. 1–19 (2011)
Codesurfer, Grammatech (2013)
Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York (1997)
Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: PLDI, pp. 290–299 (2007)
Horwitz, S., Reps, T.W.: The use of program dependence graphs in software engineering. In: ICSE, pp. 392–411 (1992)
Jang, J., Agrawal, A., Brumley, D.: Redebug: Finding unpatched code clones in entire os distributions. In: IEEE Symposium on Security and Privacy (S&P), pp. 48–62 (2012)
Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: ICSE, pp. 96–105 (2007)
Jiang, L., Su, Z.: Automatic mining of functionally equivalent code fragments via random testing. In: ISSTA, pp. 81–92 (2009)
Kim, J., Lee, S., Hwang, S.-W., Kim, S.: Towards an intelligent code search engine. In: AAAI (2010)
Komondoor, R., Horwitz, S.: Tool demonstration: finding duplicated code using program dependences. In: ESOP, pp. 383–386 (2001)
Lattner, C., Lenharth, A., Adve, V.S.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: PLDI, pp. 278–289 (2007)
Lee, M.-W., Roh, J.-W., won Hwang, S., Kim, S.: Instant code clone search. In: SIGSOFT FSE, pp. 167–176 (2010)
Li, J., Ernst, M.D.: Cbcd: Cloned buggy code detector. In: ICSE, pp. 310–320 (2012)
Liu, C., Chen, C., Han, J., Yu, P.S.: GPLAG: Detection of software plagiarism by program dependence graph analysis. In: KDD, pp. 872–881 (2006)
Manning, C., Raghavan, P., Schütze, H.: Introduction to Information Retrieval, vol. 1. Cambridge University Press, Cambridge (2008)
McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., Fu, C.: Portfolio: finding relevant functions and their usage. In: ICSE, pp. 111–120 (2011)
Meng, N., Kim, M., McKinley, K.S.: Sydit: creating and applying a program transformation from an example. In: SIGSOFT FSE, pp. 440–443 (2011)
Meng, N., Kim, M., McKinley, K.S.: Systematic editing: generating program transformations from an example. In: PLDI, pp. 329–342 (2011)
Meng, N., Kim, M., McKinley, K.S.: Locating and applying systematic edits by learning from examples. In: ICSE (2013)
Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**, 348–375 (1978)
Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Graph-based mining of multiple object usage patterns. In: ESEC/SIGSOFT FSE, pp. 383–392 (2009)
O'Callahan, R., Jackson, D.: Lackwit: a program understanding tool based on type inference. In: ICSE, pp. 338–348 (1997)
Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (2009)
Sun, B., Podgurski, A., Ray, S.: Improving the precision of dependence-based defect mining by supervised learning of rule and violation graphs. In: ISSRE, pp. 1–10 (2010)
Thummalapenta, S., Xie, T.: Parseweb: a programmer assistant for reusing open source code on the web. In: ASE, pp. 204–213 (2007)
Tian, Y., Lo, D., Lawall, J.L.: Automated construction of a software-specific word similarity database. In: CSMR-WCRE, pp. 44–53 (2014)
Wang, S., Lo, D., Jiang, L.: Code search via topic-enriched dependence graph matching. In: WCRE, pp. 119–123 (2011)

Wang, S., Lo, D., Jiang, L.: Inferring semantically related software terms and their taxonomy by leveraging collaborative tagging. In Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM), ICSM '12, pp. 604–607 (2012)
Wang, S., Lo, D., Jiang, L.: Understanding widespread changes: a taxonomic study. In: CSMR (2013)
Wang, S., Lo, D., Xing, Z., Jiang, L.: Concern localization using information retrieval: an empirical study on linux kernel. In: WCRE, pp. 92–96 (2011)
Wang, X., Lo, D., Cheng, J., Zhang, L., Mei, H., Yu, J.X.: Matching dependence-related queries in the system dependence graph. In: ASE, pp. 457–466 (2010)
Wilcoxon, F.: Individual comparisons by ranking methods. Biom. Bull. **1**(6), 80–83 (1945)
Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In ICDM, pp. 721–724 (2002)
Yan, X., Han, J.: Closegraph: mining closed frequent graph patterns. Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. KDD '03, pp. 286–295. NY, USA, ACM, New York (2003)
Zhu, F., Qu, Q., Lo, D., Yan, X., Han, J., Yu, P.S.: Mining top-k large structural patterns in a massive network. PVLDB **4**(11), 807–818 (2011)