Identifying Similar Code with Program Dependence Graphs

Jens Krinke
Lehrstuhl Softwaresysteme
Universität Passau
Passau, Germany

Abstract

We present an approach to identify similar code in programs based on finding similar subgraphs in attributed directed graphs. This approach is used on program dependence graphs and therefore considers not only the syntactic structure of programs but also the data flow within (as an abstraction of the semantics). As a result, there is no tradeoff between precision and recall—our approach is very good in both. An evaluation of our prototype implementation shows that our approach is feasible and gives very good results despite the non polynomial complexity of the problem.

1. Introduction

Duplicated code is common in all kind of software systems. Although cut-copy-paste (-and-adapt) techniques are considered bad practice, every programmer is using them. Code duplication is easy and cheap during software development, but it makes software maintenance more complicated:

- Errors may have been duplicated together with the duplicated code.
- Modifications of the original code often must also be applied to the duplicated code.

Especially for software renovation projects, it is desirable to detect duplicated code; a number of approaches have been developed [10, 4, 3, 11, 13]. These approaches are graph-based [10], text-based (and language independent) [4], syntax-based [3] or are based on metrics (syntax- and/or text-based) [11, 13]. Some approaches can only detect (textual or structural) identical duplicates, which are not typical in software systems as most duplicates are adapted to the environment where they are used.

In Figure 1 two similar pieces of code in `main.c` from the `agrep` program are shown, which have been detected as duplicates by our prototype tool. Let us assume that the left part is the original and the right part is the duplicate. We can identify some typical modifications to the duplicate:

1. Parts of the code will be executed under different circumstances (lines 742 and 743 have been moved into an `if` statement in lines 473-476).
2. Variables and/or expressions are changed (lines 743/478, 747/483, . . .).
3. Parts of the code are inserted or deleted ("`lasti = i-1`" in line 758).
4. Code is moved to different locations ("`j++`" in line 481/748).

Modifications disturb the structure of the code and duplicated code is more complicated to identify. This causes a tradeoff between precision (amount of false positives) and recall (amount of undiscovered duplicates) in text- or structure-based detection methods. To also detect not identical but similar duplicates (increased recall), the methods have to ignore certain properties. However, this may lead to false positives (reduced precision). This tradeoff has been studied in [11]. Some of the approaches suffer the *splited duplicates* symptom: A simple modification in a duplicate causes a detection of two independent duplicates, one duplicate for the code before the modifications and one after it. If the unmodified parts are to small, the duplicate is not even identified.

We have developed an approach which does not suffer under the tradeoff between recall and precision and where modified duplicates can still be detected. Such an approach cannot just be based on text or syntax, but has to consider semantics too. Our approach is based on *fine-grained program dependence graphs (PDGs)* which represent the structure of a program and the data flow within it. In these graphs, we try to identify similar subgraph structures which are stemming from duplicated code. Identified similar subgraphs can be directly mapped back onto the program code and presented to the user. However, our approach is work in progress and some issues are still to be resolved.

```
if(c != Newline)
{
  r1 = Init1 & r3;
  r2 = (Next[r3] & CMask) | r1;
}
else {
  r1 = Init1 & r3;
  r2 = Next[r3] & CMask | r1;
  j++;
  if(TAIL) r2 = Next[r2] | r2 ;
  if(( r2 & 1) ^ INVERSE) {
    if(FILENAMEONLY) {
      num_of_matched++;
      printf("%s\n", CurrentFileName);
      return;
    }
    r_output(buffer, i-1, end, j);
  }
  lasti = i - 1;
  r3 = Init0;
  r2 = (Next[r3] & CMask) | Init0;
}
c = buffer[i++];
CMask = RMask[c];
...
```

```
if(c != Newline)
{ if(CMask != 0) {
  r1 = Init1 & r3;
  r2 = ((Next[r3>>hh] | Next1[r3&LL]) & CMask) | r1;
 }
 else  {
  r2 = r3 & Init1;
 }
}
else { j++;
  r1 = Init1 & r3;
  r2 = ((Next[r3>>hh] | Next1[r3&LL]) & CMask) | r1;
  if(TAIL) r2 = (Next[r2>>hh] | Next1[r2&LL]) | r2;
  if(( r2 & 1 ) ^ INVERSE) {
    if(FILENAMEONLY) {
      num_of_matched++;
      printf("%s\n", CurrentFileName);
      return;
    }
    r_output(buffer, i-1, end, j);
  }
  r3 = Init0;
  r2 = (Next[r3>>hh] | Next1[r3&LL]) & CMask | Init0;

}
c = buffer[i++];
CMask = Mask[c];
...
```

Figure 1. Two similar pieces of code from `agrep`

The rest of this paper is structured as follows: In the next Section we present *fine-grained* program dependence graphs (in contrast to traditional program dependence graphs). Section three formalize how similar subgraphs can be identified in attributed directed graphs. In Section four we present our specific implementation which is evaluated in Section five. Related work is presented in Section six and finally, in Section seven we discuss some future work.

2. Fine-grained Program Dependence Graphs

The traditional *program dependence graph (PDG)* [8] is a directed attributed graph whose vertices represent the assignment statements and control predicates that occur in a program. Some of the vertices have an attribute that marks them as entry vertices, which represent the entry of procedures. The edges represent the *dependences* between the components of the program. They have two attributes: the first is separating the edges into *control* and *data dependence edges* and the second is `true` or `false` for control dependence edges. A control dependence edge from vertex v_1 to v_2 represents that if the predicate that is represented by v_1 is evaluated to the second attribute of the edge, the component that is represented by v_2 will be executed. A data dependence edge from vertex v_1 to v_2 represents that the component represented by v_1 assigns a value to a variable which may be used at the component represented by v_2. Program dependence graphs can be used for all kind of software engineering and reengineering problems; the main application is program slicing [7, 8].

Our PDG is a specialization of the traditional and is similar to both the AST and the traditional PDG. On the level of statements and expressions, the AST vertices are almost mapped one to one onto PDG vertices. The definitions of variables and procedures have special vertices. The vertices may be attributed with a class, an operator and a value. The class specifies the kind of vertex: statement, expression, procedure call etc. The operator further specifies the kind, e.g. binary expression, constant etc. The value carries the exact operator, like “+” or “–”, constant values or identifier names.

Between vertices that represents components of expressions we have also specialized edges. Between the components of an expression we have a special control dependence which we call *immediate (control) dependence*: the targets of immediate control dependence edges are evaluated before the source is evaluated. The data flow between the expression components is represented by another specialized edge: the *value dependence edge*, which is like a data dependence edge between expression components. Another specialized edge represents the assignments of values to variables. The *reference dependence edges* are similar to the value dependence edges, except that they show that a computed value is stored into a variable.

Definition 1 *(The definitions of data and control dependence are still the same.)*

1. *An expression vertex n is* value dependent *on an expression vertex p, if the value computed at p is needed at vertex n.*

2. *An assignment vertex n is* reference dependent *on an expression vertex p, if the value computed at p is stored into a variable at n.*

Figure 2 shows an example PDG of the following code.

```
void f ( int a, int b, int c ) {
  x = a * (y = b + c);
  z = x + y;
}
```

In this example, vertex 5 is an entry vertex, the vertices 6, 7 and 8 are formal-in vertices and the vertices 13, 14 and 15 are formal-out vertices. They all have the same kind and operator. Vertex 12 is a compound vertex, which groups the subgraphs of the two assignments together. The remaining vertices are all expression vertices with different operators. For example, vertex 26 is an expression vertex with operator kind "binary" and value "+". Note that assignments are expressions, e.g. vertex 25 is an expression vertex with an operator "assign" and no value.

The benefit of fine-grained program dependence graphs is the structural representations of expressions and the richness of the attributes that eases the identification of similar or identical vertices and edges. This is presented in the next section.

3. Identification of similar subgraphs

Program dependence graphs are *attributed directed graphs*, where the vertex attributes are the class, the operator and the value of the vertices and the edge attributes are the class and the label of the edges.

An *attributed directed graph* is a is a 4-tuple $G = (V, E, \mu, \nu)$ where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, $\mu : V \rightarrow A_V$ maps vertices to the vertex attributes and $\nu : E \rightarrow A_E$ maps edges to the edge attributes. Let $\Delta : E \rightarrow A_v \times A_E \times A_v$ be the mapping $\Delta(v_1, v_2) = (\mu(v_1), \nu(v_1, v_2), \mu(v_2))$. A *path* is a finite sequence of edges and vertices $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$ where $e_i = (v_{i-1}, v_i)$ for all $1 \leq i < n$. A *k-limited path* is a path $v_0, e_1, v_1, e_2, \ldots, e_n, v_n$ with $n \leq k$.

Two attributed directed graphs $G_1 = (V_1, E_1, \mu_1, \nu_1)$ and $G_2 = (V_2, E_2, \mu_2, \nu_2)$ are *isomorphic*, if a bijective mapping $\phi : V_1 \rightarrow V_2$ exists with:

$$\begin{aligned}(v_i, v_j) \in E_1 \quad \Longleftrightarrow \quad & (\phi(v_i), \phi(v_j)) \in E_2, \\ & \Delta_1(v_i, v_j) = \Delta_2(\phi(v_i), \phi(v_j))\end{aligned}$$

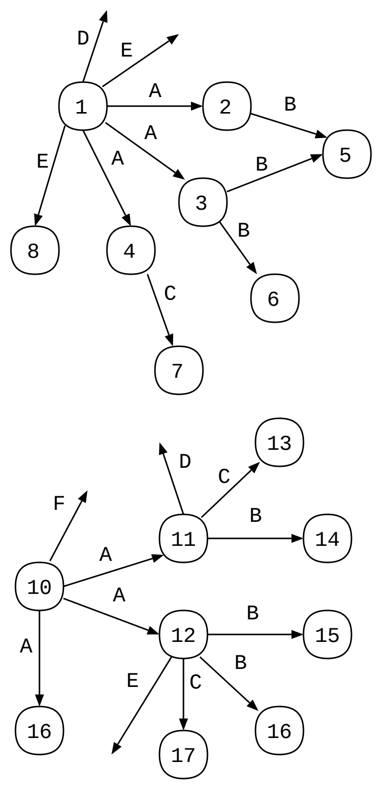

Figure 3. Two simple graphs

This means that two graphs are isomorphic if every edge is bijectively matched to an edge in the other graph and the attributes of the edges and the incident vertices are the same. The question *whether two given graphs are isomorphic* is NP-complete in general.

In Figure 3 two simple attributed graphs are shown, where the edge labels represents the complete attribute-tuple of the vertex and edge attributes. There is no single pair of maximal isomorphic subgraphs, at least two exists with six vertices each. We are also more interested in *similar* subgraphs which do not have to be isomorphic. Defining something to be similar is always tricky, as similarity is nothing precise but something vague. Nevertheless we try to define similarity between graphs by relaxing the mapping between edges: We consider two graphs G and G' as similar, if for every path $v_0, e_1, v_1, e_2, \ldots, e_n, v_n$ in one graph there exists a path $v'_0, e'_1, v'_1, e'_2, \ldots, e'_n, v'_n$ in the other graph and the attributes of the vertices and the edges are identical if the path are mapped against each other ($\forall_{1 \leq i \leq n} e_i, e'_i : \Delta(e_i) = \Delta'(e'_i)$). The second restriction is that all paths have to start at a single vertex v in G and at v' in G' ($v_0 = v, v'_0 = v'$ for all such paths).

A naive approach to identify the maximal similar subgraphs would now calculate all (cycle free) paths starting at

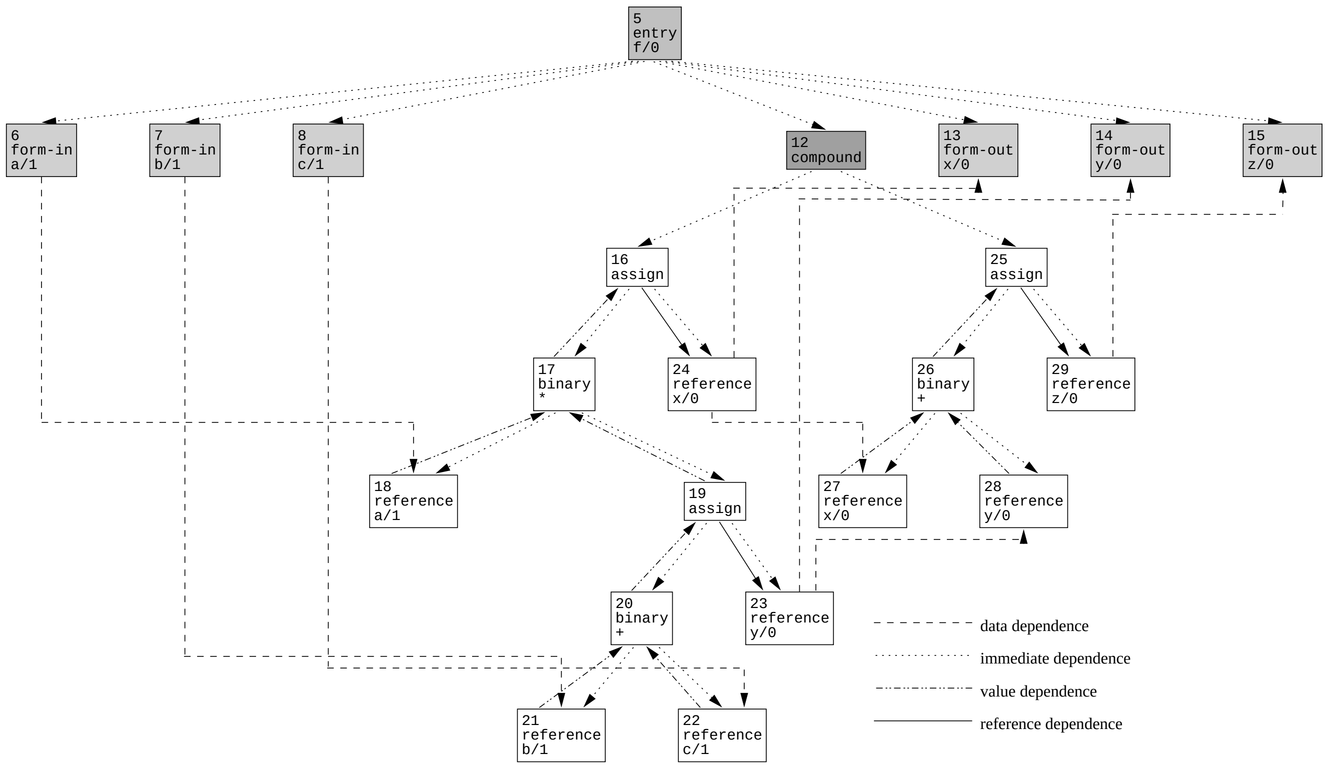

Figure 2. Example for a fine grained PDG

v and v' and would do a pairwise comparison afterwards. Of course, this is infeasible. Even if the paths are length limited, the maximal length would be unusable small.

Our approach is constructing the maximal similar subgraphs by induction from the starting vertices v and v' and is matching length limited similar paths. What makes this approach feasible, is that it considers all possible matchings *at once*. In many cases, an edge under consideration can be matched to more than one edge. Instead of checking every possible pair, we check the complete set of matching edges. This is best seen at the example:

1. The algorithm starts with $v = 1$ and $v' = 10$. These vertices are considered the endpoints of matching paths of the length zero.
2. Now, the matching paths are extended: The incident edges are partitioned into equivalence classes based on the attributes. There is only one pair of equivalence classes that share the same attributes in both graphs: $\{(1,2),(1,3),(1,4)\}_A$ and $\{(10,11),(10,12),(10,16)\}_A$.
3. The reached vertices are now marked as being part of the maximal similar subgraphs and the algorithm is continuing with the sets of reached vertices $\{2,3,4\}$ and $\{11,12,16\}$.
4. Again the incident edges are partitioned into the first pair $\{(2,5),(3,5),(3,6)\}_B$ and $\{(11,14),(12,15),(12,16)\}_B$ and the second pair $\{(4,7)\}_C$ and $\{(11,13),(12,17)\}_C$. For both pairs the algorithm continues recursively.
5. The reached vertices $\{5,6\}$ and $\{14,15,16\}$ are marked as parts of the maximal similar subgraphs. No edges are leaving these vertices.
6. The other set pair of reached vertices $\{7\}$ and $\{13,17\}$ are marked. No edges are leaving these vertices.
7. As no more set pairs exists, the algorithm terminates.

In the end, the algorithm has marked $\{1,2,3,4,5,6,7\}$ and $\{10,11,12,13,14,15,16,17\}$ which induce the maximal similar subgraphs. By accident, this is identical to the union of all maximal isomorphic subgraphs. In general, it will only be similar to maximal isomorphic subgraphs.

A simplified version of the algorithm is is shown in Figure 4. It calculates the maximal similar subgraphs G_1 and G_2 which are induced by k-limited paths starting at the vertices v_1 in G_1 and v_2 in G_2. We call these graphs *maximal similar k-limited path induced subgraphs $G^k_{v_1}$ and $G^k_{v_2}$.*

Before maximal similar k-limited path induced subgraphs G^k_v and $G^k_{v'}$ can be found, the possible pairs (v, v')

```
propagate($V_1, V_2, l$):
If $l \leq k$:
    Let $V_1 \subset V$ and $V_2 \subset V$ be the the endpoints
        of similar paths.
    Let $E_1$ and $E_2$ be the edges that are leaving
        the vertices of $V_1$ and $V_2$.
    Partition $E_1$ and $E_2$ into equivalence classes
        $E_{1_i}$ and $E_{2_i}$ based on $\Delta$.
    For all $E_{1_i}$ with their corresponding $E_{2_i}$:
        Add edges from $E_{1_i}$ and $E_{2_i}$ to $G_{v_1}$ and $G_{v_2}$
        Let $V_{1_i}$ and $V_{2_i}$ be the vertices that are
            reached by the edges in $E_{1_i}$ and $E_{2_i}$
        Call propagate($V_{1_i}, V_{2_i}, l+1$)

generate($v_1, v_2, k$):
Initialize $G_{v_1}$ and $G_{v_2}$ to be empty.
Call propagate($\{v_1\}, \{v_2\}, 1$)
Return $G_{v_1}$ and $G_{v_2}$ as result.
```

Figure 4. Algorithm to generate G_{v_1} and G_{v_2}

have to be detected. A naive approach would be to check all pairs $V \times V$ which leads to a complexity of $O(|V|^2)$ (independent of the complexity of the generation of the subgraphs them self). Even with smarter approaches, this complexity cannot be reduced. Therefore, only a subset of V should be considered as "starting" vertices, as most other vertices are reached during the construction of the maximal subgraphs. This subset should be based on specific features of the vertices which is highly application specific.

4. Implementation

To find similar code based on identifying maximal similar subgraphs in fine-grained PDGs we first had to find the subset of the vertices which are used in the pairwise construction of the subgraphs. One possibility would have been to use entry vertices, which would find similar procedures. We decided to use predicate vertices instead, because we also want to find similar pieces of code independent of procedures. For every pair of predicate vertices (v_1, v_2) the maximal similar $G^k_{v_1}$ and $G^k_{v_2}$ are generated. The generation is basically a recursive implementation of the induction from Figure 4.

4.1. Weighted subgraphs

If we take the subgraphs as direct result, they just represent *structural* similarity which can also be achieved via less expensive techniques like [3]. The subgraphs can be large even if they do not even have a similar semantic. The reason is that the data dependence edges may not match and the subgraphs are only or mostly induced by control dependence edges. For example, two nodes A and B may be included in the subgraph because they are reached by control dependence edges which match in the similar subgraph. It is possible that a data dependence edge from A to B is not included in the subgraph, because there is no matching edge in the similar subgraph. Only if the data dependence edges are considered special it is guaranteed that the subgraphs have a similar semantic.

Therefore the constructed subgraphs have to be weighted. A simple criterion is just the number of data dependence edges in the subgraphs. As our evaluation in the next section shows, this criterion is good enough. However, other, more sophisticated criterions are possible like the percentage of data dependence edges or the amount and the length of paths induced by data, value and reference dependence edges.

Another possibility is to reduce the constructed subgraphs, which are edge induced, to a connected node induced part. In that case no pair of nodes exists in the subgraph, which are connected by an edge not included in the subgraph. This is planned for the future.

4.2. The used infrastructure

The presented technique has been implemented in a prototype on top of our infrastructure to analyze ANSI-C programs [12]. This infrastructure is aimed at the validation of measurement system software and therefore complex data flow analysis techniques like flow sensitive points-to analysis are used to construct the PDGs. For the application of identifying maximal similar subgraphs such complex data flow techniques are not needed, we just used it for our prototype implementation because it was readily available. PDGs for the application of identifying similar subgraphs just need basic data flow information.

5. Evaluation

Like any other k-limited technique, the presented work had to be "tuned" to find an appropriate value for k. We therefore checked a set of test programs stemming from different sources for duplicated code. The results can be seen for some examples in Figure 5. The size of the programs are given in terms of lines of code and the number of vertices and edges in the PDG. For different limits k between 10 and 50 the running times are given (measured in seconds of user time spent). A direct relation between the size of a program and the running time does not exists as the running time is mostly dependent on the size and the amount of similar subgraphs within a program. However, due to the pairwise comparison we expect a quadratic complexity overall. In the same table, the last three columns show the amount of

Project	LOC	Edges	Vertices	Time f. limit k (sec)						Duplicates		
				k=10	k=20	k=30	k=40	k=50	k=100	≥10	≥20	≥50
`agrep`	3968	69032	22588	26.4	207.9	1465	7150	38848	-	155	91	12
`bison`	8303	79030	28071	8.9	47.4	249.2	714.5	920.3	921.6	34	22	0
`cdecl`	3879	40578	12939	0.6	0.6	0.6	0.6	0.6	0.6	0	0	0
`compiler`	2402	99219	16497	226.8	237.6	237.6	237.6	237.6	237.6	94	67	51
`ctags`	2933	45249	12446	0.6	0.8	0.8	0.8	0.8	0.8	0	0	0
`diff`	17485	169508	43518	2.5	9.1	32.0	61.4	63.6	63.6	40	10	0
`fft`	3242	35701	16446	6.0	53.4	297.2	892.9	1292	1296	16	14	8
`flex`	7640	124730	37073	3.3	3.8	4.2	4.3	4.3	4.3	16	0	0
`football`	2261	63833	18718	30.3	49.9	54.7	54.7	54.7	54.7	50	2	0
`larn`	10410	817432	158077	271.4	4242	5878	5905	5867	5876	91	53	6
`patch`	7998	196106	29766	6.27	7.5	8.6	9.2	9.3	9.2	2	0	0
`rolo`	5717	50816	17438	0.7	0.7	0.7	0.7	0.7	0.7	0	0	0
`simulator`	4476	34939	14438	1.4	2.4	2.6	2.6	2.6	2.6	0	0	0
`spim`	19739	1338294	122819	525.9	703.5	798.5	809.1	809.1	807.2	30	16	0
`twmc`	24950	1605532	181281	918.4	24263	-	-	-	-	1383	992	639

Figure 5. Sizes and running times for some test cases

discovered duplicates with a minimum weight of 10, 20 and 50. The limit used was $k = 20$ and only minimal differences exists for larger k (except for `twmc`). Due to lack of time it was impossible to manually verify all reported duplicates. However, all reported duplicates we checked were correct (100% precision).

Due to the complexity of the data flow analysis used in our infrastructure, we are only able to construct PDGs up to a limited size. Therefore all of our test cases are of limited size too. This does not mean that the presented technique has the same limit—we plan a reimplementation on top of a different infrastructure to evaluate for big programs.

5.1. Optimal limit

To insure highest possible recall, a very high k-limit is desirable. However, this is not possible due to the exponential complexity of the graph comparison. Our claim is that a small k is sufficient and that a limit above this small value will not increase recall. We found this claim to be true for almost any test case. A typical case is `bison`, for which the results are shown in Figure 6. All test cases were repeated for limits $0 \leq k \leq 30$ (y-axis). Also shown is how many duplicates (z-axis) are reported that are above a specific minimum weight (y-axis). As we can see, for very small k ($< 5 - 10$) almost no duplicates are reported. For bigger (but still small) k ($< 15 - 20$) the amount of reported duplicates is increasing fast. For bigger k (> 20) the amount of reported duplicates is not changing any more. We have found this to be the same for almost any other test case—a k-limit around 20 seems to be sufficient for highest recall.

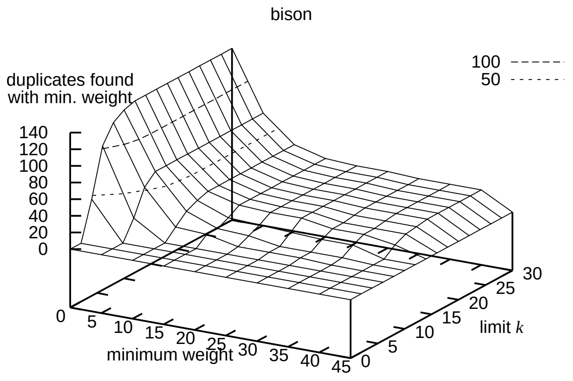

Figure 6. Results for `bison`

5.2. Minimum weight

The other "tunable" parameter in our technique is the minimum weight of a similar subgraph before it is reported. This value is not critical like the k-limit, as it does not influence the comparison itself. Normally, all possible duplicates are identified independent of their weights and the minimum weight just changes the amount of *reported* duplicates. The `bison` test case is an ideal example: for small minimum weights, many duplicates are reported. For bigger minimum weights this changes quickly, which shows that the majority of duplicates are small pieces of codes. For minimum weights between 10 and 40, around 40 duplicates are reported. For minimum weights above 45, no duplicates are reported, which shows that the maximum weight of all duplicates is less than 45.

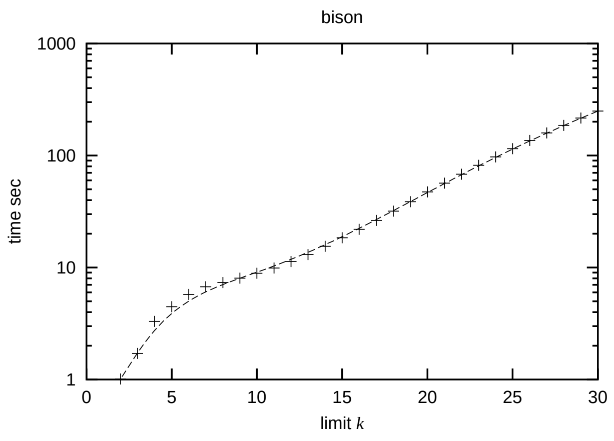

Figure 7. Running times of `bison`

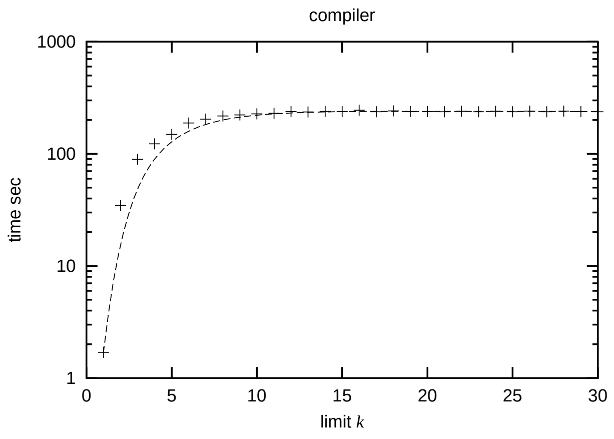

Figure 8. Running times of `compiler`

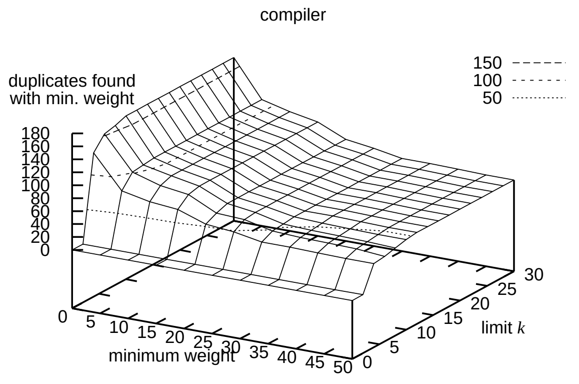

Figure 9. Results for `compiler`

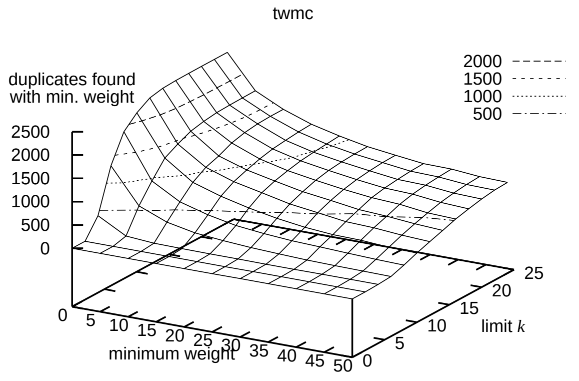

Figure 10. Results for `twmc`

We have found that there is no "ideal" minimum weight, as every test case has different amounts of reported duplicates with varying minimum weights. This is not unexpected, as duplication is different in every program. Unlike k the minimum weigth can be tuned *after* the identification finished during presentation to the user.

5.3. Running time

Figure 7 shows the times for the `bison` example, which are increasing exponential for large k. We claimed that a k-limit around 20 is ideal for recall: we need 47 seconds to analyze `bison` under this limit. For some test cases we have found an interesting behavior—the running time is not increasing exponential but reverse logarithmic for increased k. This is shown in Figure 8 for the test case `compiler`. As you also can see in Figure 9, for k-limits bigger than ten the amount of reported duplicates stays the same: there are more than 50 duplicates with a weight bigger than 50. This means that there are no similar paths longer than 10 edges in that software and the limit is not reached for larger limits. The result is that the time needed to calculate the similar graphs is independent of k for k bigger than 10. Therefore, the overall needed time is not changing above that. The same behavior can be seen for most of the test cases in Figure 5: only two of the test cases have differences in running time for the limits $k = 50$ and $k = 100$. For all others, even the amount of reported duplicates does not change for $k > 20$ (not shown).

One of our test case (see Figure 10) was different than all others: First of all, we could not test for k-limits bigger than 25, as the running time was already at 46 hours. Also, the amount of reported duplicates was incredibly high: more than 500 with a weight bigger than 50 and more than 1000 with a weight bigger than 20. These extreme high numbers are stemming from massive code duplication in that particular software. We have found a high amount of files, which just have been copied and slightly changed for slightly different purpose.

6. Related Work

An approach very similar to ours is [10], which is based on (traditional) program dependence graphs. Starting from *every* pair of matching nodes, they construct isomorphic subgraphs for ideal clones which can be replaced by function calls automatically. Unlike our approach, their subgraphs are only subtrees which are not maximal, as they visit every node only once during subgraph construction. Like us, they cannot analyze big programs due to limitations of the underlying PDG generating infrastructure.

Another structure comparing work is [3], where a program under observation is transformed to an AST first. For every subtree in the AST a hash value is computed and identical subtrees are identified via identical hash values. To also detect similar (not identical) subtrees, the subtrees have to be pairwise compared. The authors suggest many improvements as future work which are similar to our approach.

An approach which obeys but not compares syntactical structure is [13], where metrics are calculated from names, layout, expression and (simple) control flow of functions. Two functions are considered as clones if their metrics are similar. This work can only identify similar functions but not similar pieces of code. A language independent approach is [4] which is looking for specific patterns in a comparison from every line to every other. Another text-based approaches is [2]. These approaches can be used to analyze very large programs, as they are not relying on pairwise comparison.

An application in the same setting is the detection of *plagiarism*: Given two programs, one has to detect if one program is in part or completely duplicated in the other. Most plagiarism detecting systems are comparing the lexical structure of the programs [14, 16]. Other system are again based on metrics; however, studies show that metrics-based systems are only partly successful because of the tradeoff between recall and precision, both for detection of plagiarism [15] and detection of similar code [11].

The opposite problem to identifying similar or identical parts of programs is identifying the differences between programs. [7, 6] is an approach to identify program differences based on program dependence graphs. However, that approach relies on the existence of a mapping ϕ that maps every vertex of one program to a vertex of the other if the representing program components are the same in both programs. The authors suggest a special program editor that keeps such a mapping. Instead, our approach could be used to find such a mapping.

The matching of similar (attributed) graphs is used in other areas like computer vision [9] and graph visualization [1] too.

7. Summary and future work

We have presented a technique for identifying similar code based on finding maximal similar subgraphs in fine-grained program dependence graphs. As this problem is not solvable in polynomial time, a k-limiting technique is used. A prototype implementation shows that this approach is feasible even with the non polynomial complexity of the problem and results in high precision and recall.

This is work in progress and some obstacles remain to be solved: First off all, high amounts of duplicated code cause exploding running times. Secondly, large duplicated code sections cause many duplicates to be reported, as duplicates are basically reported for every predicate within. These duplicates are overlapping and have to be merged before reported to the user.

Our future plans include:

- A reimplementation on top of a simpler infrastructure to enable an evaluation for large programs. Due to the underlying infrastructure for PDG generation, our prototype is only able to analyze programs up to limited size.

- An adaption of our prototype for detection of plagiarism. We are using JPlag [14] in education with great success. However, a manual check is still needed as students are aware of our tool usage and try to hinder the detection through simple modifications. A plagiarism detection tool based on our approach should not be so easily confused.

- A full evaluation of precision and recall. That we did not find false positives does not mean that there aren't any—we were unable to check all reported duplicates. Also, we have not really checked recall as we are not able to check all test programs for code duplication manually. A cross check against other tools is desirable.

- A general implementation for other types of graphs. Graphs are one of the main representation for data in reengineering and a detection of similar subgraphs is often helpful. This implementation could be based on GXL [5], a proposed standard exchange format for graphs.

- An automatic substitution of identified duplicated code through new functions or macros. As the underlying infrastructure contains enough semantic information in the PDGs, the *isomorphic* subgraphs can be identified and replaced by new parameterized function calls which do not change the semantic of the program.

References

[1] S. Bachl. *Erkennung isomorpher Subgraphen und deren Anwendung beim Zeichnen von Graphen*. Dissertation, Universität Passau, 2000. (In German).

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings: Second Working Conference on Reverse Engineering*. IEEE Computer Society Press, 1995.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings; International Conference on Software Maintenance*, 1998.

[4] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings; IEEE International Conference on Software Maintenance*, 1999.

[5] R. C. Holt, A. Winter, and A. Schürr. GXL: Towards a standard exchange format. In *Proceedings 7th Working Conference on Reverse Engineering WCRE*, 2000.

[6] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3), 1989.

[7] S. B. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992.

[8] S. B. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[9] H. Kälviäinen and E. Oja. Comparisons of attributed graph matching algorithms for computer vision. Technical report, Lappeenranta University Of Technology, Finland, 1990.

[10] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Eigth International Static Analysis Symposium (SAS)*, 2001.

[11] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In *Proceedings Fourth Working Conference on Reverse Engineering*, 1997.

[12] J. Krinke and G. Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, 40(11-12), 1998.

[13] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, 1996.

[14] L. Prechelt, G. Malpohl, and M. Philippsen. JPlag: Finding plagiarisms among a set of programs. Technical Report 2000-1, Fakultät für Informatik, Universität Karlsruhe, Germany, 2000.

[15] K. L. Verco and M. J. Wise. Plagiarism à la mode: a comparison of automated systems for detecting suspected plagiarism. *The Computer Journal*, 39(9), 1996.

[16] M. J. Wise. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. In *Proceedings of the 23rd Technical Symposium on Computer Science Education*, volume 24(1) of *SIGSCE Bulletin*, 1992.