# Refactoring Clones: An Optimization Problem

Giri Panamoottil Krishnan, Nikolaos Tsantalis
Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
giri.krishnan@concordia.ca, nikolaos.tsantalis@concordia.ca

***Abstract*—The refactoring of software clones is achieved by extracting their common functionality into a single method. Any differences in identifiers and literals between the clones have to become parameters in the extracted method. Obviously, a large number of differences leads to an extracted method with limited reusability due to the large number of introduced parameters. We support that minimizing the differences between the matched statements of clones is crucial for the purpose of refactoring and propose an algorithm that treats the matching process as an optimization problem.**

## I. Introduction

The problem of source code matching or differencing has been investigated within the context of various applications, including change evolution analysis, plagiarism detection, code reuse, aspect mining, clone detection and refactoring. However, current approaches either do not explore the entire search space of possible matches, and thus may return non-optimal solutions, or face scalability issues due to the problem of combinatorial explosion. To facilitate the refactoring of duplicated code, an optimal solution should not only contain the maximum number of possible mapped statements, but also the minimum number of differences between them. To this end, we propose an algorithm to tackle both optimality and scalability issues. The contributions of the paper are: **(a)** we support that the problem of finding a mapping between the statements of two clones is an optimization problem with two objectives, namely maximizing the number of mapped statements and at the same time minimizing the number of differences between the mapped statements, **(b)** we express this optimization problem as finding the *Maximum Common Subgraph* (MCS) with the minimum number of differences in the *Program Dependence Graphs* (PDGs) of the clones, and **(c)** we apply a bottom-up mapping process based on the control dependence structure of the PDGs. This divide-and-conquer approach breaks the initial mapping problem into smaller sub-problems and avoids the risk of combinatorial explosion that might occur in the initial problem.

## II. Related Work

Komondoor and Horwitz [1] apply slicing on PDGs to find isomorphic subgraphs that represent code clones. The advantage of this approach is the detection of non-contiguous clones (i.e., clones with gaps), clones with re-ordered statements, and clones intertwined with each other. Two nodes are matched if the corresponding statements are syntactically identical (i.e., their AST representation has the same structure) allowing only for differences in variable names and literal values. Krinke [2] proposed an approach to identify code clones by finding the maximal similar subgraphs in two PDGs by induction from a pair of starting vertices. To reduce the complexity of the algorithm, he considers only a subset of vertices (i.e., predicate vertices) as starting points, and restricts the maximum length of the explored paths using a $k$-limit. One important limitation is that the running time of the algorithm explodes as $k$-limit increases. Another limitation is that the use of $k$-limit may lead to an incomplete solution (i.e., the selected $k$-limit is insufficient for detecting all possible matching vertices). Shepherd et al. [3] implemented an automated aspect mining technique exploiting the PDG and AST representations of a program. The proposed algorithm, inspired by [2] and [1], starts by matching the control dependence subgraphs of two compared PDGs to extract all possible matching solutions. Next, it filters out the undesirable matching solutions based on data dependence information. A limitation is that the algorithm always starts from the method entry nodes, and thus will fail to match control dependence subgraphs nested in different levels. More recently, Higo and Kusumoto [4] improved Komondoor's technique [1] by extending the PDG representation and introducing some heuristics to enhance code clone detection. The common limitation of all aforementioned techniques is that they do not explore the entire search space of possible solutions and therefore may return a non-optimal solution. In contrast to MCS approach that builds a search tree examining all possible combinations in the case of multiple node matches, the aforementioned techniques always select one match for each node, essentially exploring only a single path of the entire search tree.

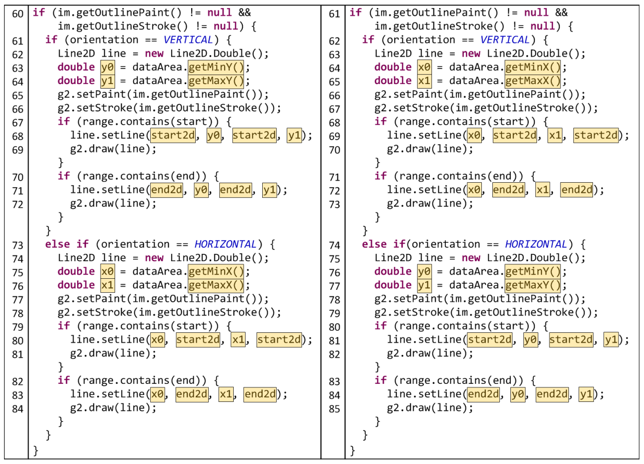
Balazinska et al. [5] focus on the extraction of differences between cloned methods and their contextual dependencies as a means to help the developer make a decision on the actual refactoring to be performed. The comparison of the cloned methods is based on the Dynamic Pattern Matching algorithm, which is applied on the sequences of tokens forming the code fragment being compared and finds an optimal distance between them (i.e., the minimum amount of tokens that have to be inserted, deleted, or substituted to transform one code fragment into the other). A limitation of the algorithm is that it does not take into account the control dependence structure of the cloned code fragments during the token alignment process. Liu et al. [6] developed a software plagiarism detection tool called GPLAG. They support that the PDG structures of the original and the plagiarized code remain invariant and exploit this property to find plagiarism through relaxed subgraph isomorphism testing i.e., by checking if a PDG is $\gamma$-isomorphic to another, where $\gamma$ is a relaxation parameter. To increase the efficiency of the algorithm, they prune the search space (i.e., reduce the number of PDG pairs to be checked) by applying some filters. Fluri et al. [7] describe an approach to extract the fine-grained changes that occur across different versions

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.47

of a program. Their method is based on the tree alignment algorithm proposed by Chawathe et al. [8], which takes as input two trees and produces a minimum edit script that can transform one tree into the other. A limitation of the proposed approach is that string-based similarity matching is not resilient to extensive renaming of identifiers. In addition, the best match approach applied for leaf level nodes may match reoccurring statements that are not at the same position in the method body. Cottrell et al. [9] present an approach to help developers integrate reusable source code. Their algorithm takes as input two ASTs and tries to produce the best correspondences between the nodes. A limitation is that the approach is semi-automated, since user intervention is required to resolve the conflicts when multiple matches are found. Additionally, it tries to find a best fit in a greedy way, which may lead to a non-optimal solution for the entire problem.

## III. Motivating Example

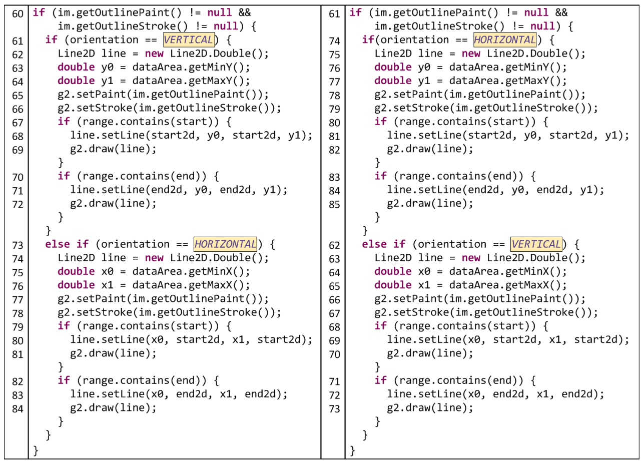
In this section, we will present an example that motivated our research and at the same time demonstrates the limitations of previous approaches. Figure 1 illustrates two code fragments taken from methods `drawDomainMarker` and `drawRangeMarker`, respectively, found in class `AbstractXYItemRenderer` of the *JFreeChart* open-source project (version 1.0.14). These two methods contain over 90 duplicated statements extending through their entire body. However, for the sake of simplicity, we have included only a small portion of the duplicated code. Figure 1 depicts a possible mapping of the statements as obtained from the PDG-based clone detection approaches discussed in section II. These techniques always select one match in the case of multiple possible node matches (e.g., statement 67 on the left side can be mapped to statements 68, 71, 80, and 83 on the right side), which, in the solution of Figure 1, coincides with the 'first' match according to the actual order of the statements. As it can be observed from Figure 1, the solution is maximum, since all 25 statements have been successfully mapped; however, it contains a large number of differences between the mapped statements. The minimization of the differences is of key importance for the refactoring of clones, since it directly affects the number of parameters that have to be introduced in the extracted method containing the common functionality, as well as the feasibility of the refactoring transformation. Figure 2 depicts the optimal mapping solution, which is again maximum in terms of the number of mapped statements, but it has also the minimum number of differences between the mapped statements. Clearly, the bodies of the `if`/`else if` statements in the left and right side of Figure 2 are 'symmetrical' to each other. Consequently, parameterizing the differences in the conditional expressions of the 'symmetrical' `if`/`else if` statements makes easier the refactoring of the clones and introduces less parameters to the extracted method.



```
60 if (im.getOutlinePaint() != null &&
       im.getOutlineStroke() != null) {
61   if (orientation == VERTICAL) {
62     Line2D line = new Line2D.Double();
63     double y0 = dataArea.getMinY();
64     double y1 = dataArea.getMaxY();
65     g2.setPaint(im.getOutlinePaint());
66     g2.setStroke(im.getOutlineStroke());
67     if (range.contains(start)) {
68       line.setLine(start2d, y0, start2d, y1);
69       g2.draw(line);
       }
70     if (range.contains(end)) {
71       line.setLine(end2d, y0, end2d, y1);
72       g2.draw(line);
       }
     }
73   else if (orientation == HORIZONTAL) {
74     Line2D line = new Line2D.Double();
75     double x0 = dataArea.getMinX();
76     double x1 = dataArea.getMaxX();
77     g2.setPaint(im.getOutlinePaint());
78     g2.setStroke(im.getOutlineStroke());
79     if (range.contains(start)) {
80       line.setLine(x0, start2d, x1, start2d);
81       g2.draw(line);
       }
82     if (range.contains(end)) {
83       line.setLine(x0, end2d, x1, end2d);
84       g2.draw(line);
       }
     }
   }
```

```
61 if (im.getOutlinePaint() != null &&
       im.getOutlineStroke() != null) {
62   if (orientation == VERTICAL) {
63     Line2D line = new Line2D.Double();
64     double x0 = dataArea.getMinX();
65     double x1 = dataArea.getMaxX();
66     g2.setPaint(im.getOutlinePaint());
67     g2.setStroke(im.getOutlineStroke());
68     if (range.contains(start)) {
69       line.setLine(x0, start2d, x1, start2d);
70       g2.draw(line);
       }
71     if (range.contains(end)) {
72       line.setLine(x0, end2d, x1, end2d);
73       g2.draw(line);
       }
     }
74   else if(orientation == HORIZONTAL) {
75     Line2D line = new Line2D.Double();
76     double y0 = dataArea.getMinY();
77     double y1 = dataArea.getMaxY();
78     g2.setPaint(im.getOutlinePaint());
79     g2.setStroke(im.getOutlineStroke());
80     if (range.contains(start)) {
81       line.setLine(start2d, y0, start2d, y1);
82       g2.draw(line);
       }
83     if (range.contains(end)) {
84       line.setLine(end2d, y0, end2d, y1);
85       g2.draw(line);
       }
     }
   }
```

Fig. 1. Non-optimal solution with 25 mapped nodes and 24 differences.



```
60 if (im.getOutlinePaint() != null &&
       im.getOutlineStroke() != null) {
61   if (orientation == VERTICAL) {
62     Line2D line = new Line2D.Double();
63     double y0 = dataArea.getMinY();
64     double y1 = dataArea.getMaxY();
65     g2.setPaint(im.getOutlinePaint());
66     g2.setStroke(im.getOutlineStroke());
67     if (range.contains(start)) {
68       line.setLine(start2d, y0, start2d, y1);
69       g2.draw(line);
       }
70     if (range.contains(end)) {
71       line.setLine(end2d, y0, end2d, y1);
72       g2.draw(line);
       }
     }
73   else if (orientation == HORIZONTAL) {
74     Line2D line = new Line2D.Double();
75     double x0 = dataArea.getMinX();
76     double x1 = dataArea.getMaxX();
77     g2.setPaint(im.getOutlinePaint());
78     g2.setStroke(im.getOutlineStroke());
79     if (range.contains(start)) {
80       line.setLine(x0, start2d, x1, start2d);
81       g2.draw(line);
       }
82     if (range.contains(end)) {
83       line.setLine(x0, end2d, x1, end2d);
84       g2.draw(line);
       }
     }
   }
```

```
61 if (im.getOutlinePaint() != null &&
       im.getOutlineStroke() != null) {
74   if(orientation == HORIZONTAL) {
75     Line2D line = new Line2D.Double();
76     double y0 = dataArea.getMinY();
77     double y1 = dataArea.getMaxY();
78     g2.setPaint(im.getOutlinePaint());
79     g2.setStroke(im.getOutlineStroke());
80     if (range.contains(start)) {
81       line.setLine(start2d, y0, start2d, y1);
82       g2.draw(line);
       }
83     if (range.contains(end)) {
84       line.setLine(end2d, y0, end2d, y1);
85       g2.draw(line);
       }
     }
62   else if (orientation == VERTICAL) {
63     Line2D line = new Line2D.Double();
64     double x0 = dataArea.getMinX();
65     double x1 = dataArea.getMaxX();
66     g2.setPaint(im.getOutlinePaint());
67     g2.setStroke(im.getOutlineStroke());
68     if (range.contains(start)) {
69       line.setLine(x0, start2d, x1, start2d);
70       g2.draw(line);
       }
71     if (range.contains(end)) {
72       line.setLine(x0, end2d, x1, end2d);
73       g2.draw(line);
       }
     }
   }
```

Fig. 2. Optimal solution with 25 mapped nodes and 2 differences.

## IV. Proposed Solution

### A. Maximum Common Subgraph algorithm

The detection of the *Maximum Common Subgraph* (MCS) is a well known NP-complete problem for which several optimal and suboptimal algorithms have been proposed in the literature. Conte et al. [10] compared the performance of the three most representative optimal algorithms, which are based on depth-first tree search. All three algorithms have an exponential (more precisely, factorial) worst case time complexity with respect to the number of nodes in the graphs, in the order of $\frac{(N_2+1)!}{(N_2-N_1+1)!}$, where $N_1$ and $N_2$ is the number of nodes in graphs $G_1$ and $G_2$, respectively [10]. The differences among the three algorithms actually lie only in the information used to represent each state of the search space, and in the kind of the heuristic adopted for pruning search paths [10]. We have adopted the McGregor algorithm [11] because it is simpler to implement and has a lower space complexity, in the order of $O(N_1)$, since only the states associated to the nodes of the currently explored path need to be stored in memory. The other two algorithms require the construction of the association graph between the two given graphs, which in the worst case can be a complete graph with a space complexity in the order of $O(N_1 \cdot N_2)$. Algorithm 1 is an adaptation of the McGregor algorithm to the particular characteristics of the PDGs. More specifically, given two PDGs, namely $PDG_i$ and $PDG_j$, Algorithm 1 enforces the following constraints:

1) An edge of $PDG_i$ is traversed only once in each path of the search tree (line **5**).
2) A node from $PDG_i$ is mapped to only one node from $PDG_j$ (and vice versa) in each path of the search tree (lines **13** and **14**).

```
Function search(pState, nodeMapping)
    Data: pState represents a parent state in the tree
    nodeMapping represents a pair of PDG nodes
    ($node_i$, $node_j$) that have been already mapped
    Result: Builds recursively a search tree.
    The leaf nodes in the deepest level are states
    corresponding to maximum common subgraphs
    /* get incoming & outgoing edges */
    $Edges_i \leftarrow node_i$.inEdges $\cup$ $node_i$.outEdges
    $Edges_j \leftarrow node_j$.inEdges $\cup$ $node_j$.outEdges
    foreach $edge_i \in Edges_i$ do
        if $edge_i \notin$ pState.visitedEdges then
            add $edge_i \rightarrow$ pState.visitedEdges
            foreach $edge_j \in Edges_j$ do
                if compatibleEdges($edge_i$, $edge_j$) then
                    $vN_i \leftarrow edge_i$.otherEndPoint
                    $vN_j \leftarrow edge_j$.otherEndPoint
                    if compatibleAST($vN_i$, $vN_j$) and
                    mappedCtrlParents($vN_i$, $vN_j$) and
                    not alreadyMapped($vN_i$) and
                    not alreadyMapped($vN_j$) then
                        mapping $\leftarrow$ ($vN_i$, $vN_j$)
                        state $\leftarrow$ createState(mapping)
                        add state $\rightarrow$ pState.children
                        search(state, mapping)
                    end if
                end if
            end foreach
        end if
    end foreach
end
```

**Algorithm 1:** Recursive function building a search tree.

3) The control dependence structure of $PDG_i$ and $PDG_j$ is preserved throughout the mapping process. This means that if two control predicate nodes $cp_i$ and $cp_j$ have been mapped at a given stage of the search process, then a node nested under $cp_i$ can only be mapped to nodes nested under $cp_j$ (and vice versa) at later stages of the search process (line **12**).

Algorithm 1 builds recursively a search tree by visiting the pairs of mapped PDG nodes in depth-first order. Each node in the search tree is created when a new pair of PDG nodes is mapped and represents a state of the search space. Each state keeps track of all visited edges and mapped PDG nodes in its path starting from the root state (function *createState* copies the visited edges and mapped nodes from the parent state to the child state). The leaf states in the deepest level of the search tree correspond to the maximum common subgraphs.

## B. PDG Node and Edge Compatibility

Two PDG nodes are considered compatible if they have a compatible AST structure. AST structural compatibility requires an identical AST tree structure and allows only for the replacement of expressions that return values of a given type (e.g., method call, field/variable/array access, class instance creation, array creation, literal, infix expressions) with other expressions (in the aforementioned list) as long as they return the same type or types being subclasses of a common superclass (excluding `Object`). In the case of control predicate nodes (e.g., `if`, `for` statements), the part of the AST structure being compared is only their conditional expression.

Two PDG edges are considered compatible if they connect nodes which are compatible (i.e., the nodes in the starting and ending points of the edges, respectively, should be compatible with each other) and they have the same dependence type (i.e., they are both control or data flow dependences). In the case of control dependences, both should have the same control attribute (i.e., True or False). In the case of data dependences, the data attributes should correspond to variables having the same name, or to variables detected as renamed during the AST compatibility check of the connected nodes. Finally, if both data dependences are *loop-carried*, then the loop nodes through which they are carried should be compatible too.

## C. Divide-and-Conquer Based on Control Structure

Despite the constraints that we have set for our MCS search algorithm, it is still subject to the combinatorial explosion effect. As the number of possible matches for the nodes increases, the width of the search tree grows rapidly as a result of the numerous combinatorial considerations to be explored. In order to reduce the risk of combinatorial explosion, we decided to take advantage of the control dependence structure of the two compared PDGs. More specifically, we first build the *Control Dependence Tree* (CDT) of each PDG. The CDT has exactly the same structure with the *Control Dependence Graph* (CDG) with the only difference being that it includes only the control predicate nodes of the PDG. Figure 3 shows the CDTs for the duplicated code fragments of Figure 1. In this particular example, the CDTs are isomorphic. In the general case, we have to find the largest common bottom-up subtrees [12] in the CDTs, since only complete AST-subtrees having the same structure can be valid candidates for refactoring.

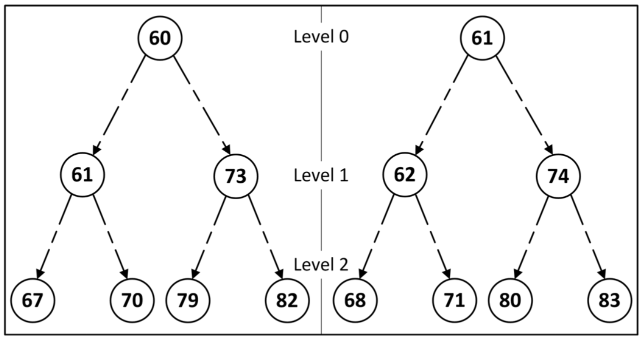


Fig. 3. The Control Dependence Trees for the code fragments of Figure 1.

Assuming that we have two isomorphic CDTs, namely $CDT_i$ and $CDT_j$ we can apply Algorithm 2 as a divide-and-conquer approach to the problem of PDG mapping. Starting from the deepest level of the CDTs, at each level the algorithm uses all possible pairwise combinations of the control predicate nodes nested at that level as starting points for Algorithm 1. Assuming that node $cp_i$ of $CDT_i$ is examined in the current level, a maximum common subgraph is generated and added in $mcsStates$ for each starting point that $cp_i$ participates in. After the examination of all possible matching combinations for node $cp_i$, the best solution in $mcsStates$ (i.e., the solution with the maximum number of mapped nodes and the minimum number of differences between them) is appended to the final solution. In level 2 of the CDTs (Figure 3), node 67 on the left side can be mapped to nodes 68, 71, 80, and 83 on the right side. Consequently, there are four possible matching nodes for node 67 and four node pairs to be used as starting points. All maximum common subgraphs resulting from the aforementioned starting points have the same number of mapped nodes, but only the subgraph generated from starting point (67, 80) has the minimum number of differences (equal to zero).

```
1  Function PDGMapping(ctrlDepTree_i, ctrlDepTree_j)
       Data: Two isomorphic CDTs
       Result: The final mapping solution as finalSolution
2      $level_i \leftarrow$ ctrlDepTree_i.maxLevel
3      $level_j \leftarrow$ ctrlDepTree_j.maxLevel
       /* an initially empty solution    */
4      finalSolution $\leftarrow \varnothing$
5      while $level_i \geq 0$ and $level_j \geq 0$ do
6          $cpNodes_i \leftarrow$ nodes at $level_i$ of ctrlDepTree_i
7          $cpNodes_j \leftarrow$ nodes at $level_j$ of ctrlDepTree_j
8          foreach $cp_i \in cpNodes_i$ do
9              $mcsStates \leftarrow \varnothing$
10             foreach $cp_j \in cpNodes_j$ do
11                 if compatibleAST($cp_i$, $cp_j$) then
12                     mapping $\leftarrow (cp_i, cp_j)$
13                     root $\leftarrow createState$(mapping)
14                     search(root, mapping)
15                     get the maximum common subgraph
                       from root & add it to $mcsStates$
16                 end if
17             end foreach
18             select the best state from $mcsStates$ &
               append it to finalSolution
19         end foreach
20         decrement $level_i$
21         decrement $level_j$
22     end while
23 end
```

**Algorithm 2:** A divide-and-conquer PDG mapping process based on control dependence structure.

## V. Evaluation

To evaluate the efficiency and scalability of our algorithm we computed the number of distinct node comparisons required for the optimal mapping of the largest method-level clones detected by ConQAT in 4 open-source systems, namely *JFreeChart*-1.0.14, *Ant*-1.9, *JMeter*-2.9, and *JRuby*-1.7.3. As it can be observed from Table I, the number of node comparisons performed by our algorithm is always significantly smaller than the maximum number of possible node comparisons (equal to $N_1 \times N_2$, where $N_1$ and $N_2$ is the number of nodes in each PDG) indicating that our approach is more sophisticated and efficient compared to exhaustive search approaches.

## VI. Future Work

Our research vision is to improve the state-of-the-art in the refactoring of clones by discovering optimal refactoring strategies. The refactoring tool we envision should be able to explain which clone differences can be parameterized or not, through a sophisticated and comprehensive visualization, suggest the changes required to make clones refactorable, detect sub-clones within larger clones that can be directly refactored, and finally perform the corresponding refactoring transformations.

TABLE I. Number of Node Comparisons for Optimal Mapping

| ID | Clone | # PDG nodes | | CDT | # CDT | # distinct node |
|---|---|---|---|---|---|---|
| | Type | PDG$_1$ | PDG$_2$ | depth | leaves | comparisons |
| 1 | Type-3 | 55 | 55 | 1 | 1 | 936 |
| 2 | Type-2 | 50 | 50 | 3 | 7 | 603 |
| 3 | Type-2 | 68 | 68 | 3 | 12 | 1040 |
| 4 | Type-1 | 50 | 50 | 1 | 2 | 806 |
| 5 | Type-1 | 67 | 67 | 1 | 25 | 1889 |
| 6 | Type-3 | 93 | 94 | 4 | 11 | 1659 |
| 7 | Type-3 | 51 | 50 | 4 | 7 | 455 |
| 8 | Type-2 | 45 | 45 | 4 | 5 | 420 |
| 9 | Type-2 | 70 | 70 | 7 | 8 | 577 |
| 10 | Type-3 | 36 | 36 | 3 | 7 | 212 |
| 11 | Type-2 | 42 | 42 | 7 | 7 | 235 |
| 12 | Type-2 | 87 | 87 | 4 | 12 | 1812 |
| 13 | Type-3 | 57 | 62 | 4 | 14 | 360 |
| 14 | Type-3 | 42 | 43 | 5 | 6 | 136 |
| 15 | Type-3 | 53 | 50 | 2 | 8 | 1021 |

## Acknowledgment

The authors would like to thank NSERC and the Faculty of Engineering and Computer Science at Concordia University for their generous support.

## References

[1] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, 2001, pp. 40–56.

[2] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering*, 2001, pp. 301–307.

[3] D. Shepherd, E. Gibson, and L. L. Pollock, "Design and evaluation of an automated aspect mining tool," in *Proceedings of the International Conference on Software Engineering Research and Practice*, 2004, pp. 601–607.

[4] Y. Higo and S. Kusumoto, "Code clone detection on specialized PDGs with heuristics," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 75–84.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in *Proceedings of the Seventh Working Conference on Reverse Engineering*, 2000, pp. 98–107.

[6] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 872–881.

[7] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[8] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 493–504.

[9] R. Cottrell, R. J. Walker, and J. Denzinger, "Semi-automating small-scale source code reuse via structural correspondence," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 214–225.

[10] D. Conte, P. Foggia, and M. Vento, "Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs," *J. Graph Algorithms Appl.*, vol. 11, no. 1, pp. 99–143, 2007.

[11] J. J. McGregor, "Backtrack search algorithms and the maximal common subgraph problem," *Software: Practice and Experience*, vol. 12, no. 1, pp. 23–34, 1982.

[12] G. Valiente, *Algorithms on Trees and Graphs*. Springer-Verlag, 2002.