Performance Tuning of PDG-based Code Clone Detection

Torsten Görg
University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
torsten.goerg@informatik.uni-stuttgart.de

Abstract: *This paper provides several ideas how to improve the performance of PDG-based code clone detection techniques. We suggest an efficient way to handle the subgraph isomorphism problem without losing precision and present an algorithm that avoids the unnecessary matching effort for subclones of larger clones.*

1 Introduction

To reach a high recall in code clone detection, PDGs (Program Dependency Graphs) can be used as an intermediate representation of the analyzed code. Komondoor and Horwitz [1] have introduced this approach. Clone pairs are calculated by matching subgraphs along two synchronously constructed slices. The main advantage of PDG-based clone detection is that it abstracts from several structural code differences which are semantically irrelevant, i.e., from reordering independent statements. PDG-based clone detection is able to recognize many more semantically equivalent clones that are not structurally equivalent than structural approches like token-based or AST-based clone detection [2]. To express program semantics in detail directly in the graph structure, Krinke [3] has suggested fine-grained PDGs. The nodes of fine-grained PDGs are at a granularity level similar to 3-address-code statements. But a fine granularity results in PDGs with many nodes so that performance issues become relevant. Because of the subgraph isomorphism problem, graph matching is principally NP-complete. Krinke tackles this problem with an approximation that groups similar paths through the graph into equivalence classes. The tradeoff is a loss of precision.

Our intention is to construct a PDG-based clone detector that provides high precision along with acceptable performance. A high detection precision is important to recognize clones that are sematically equivalent and not only similar. The contributions of this paper are to suggest a precise high-performance approach to handle the subgraph isomorphism problem specifically in the context of PDG-based clone detection and to provide an algorithm that avoids unnecessary matching effort for clones that are part of another clone.

2 Handling of the Subgraph Isomorphism Problem

During the PDG matching process two nodes v_1 and v_2 are compared based on their node types and node attribute values. If v_1 and v_2 are equal, their outgoing edges are matched. We assume that both nodes have n outgoing edges. Without further distinguishing the edges there are n! possibilities to map the outgoing edges of v_1 to the outgoing edges of v_2. Each step to an adjacent node provides a similar multitude of possibilities. To check all these possibilities requires a backtracking algorithm with exponential complexity. But NP-completeness does not necessarily make an algorithm unusable, if the problem size is small.

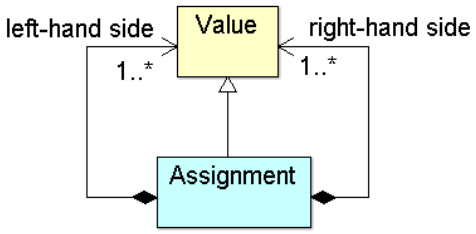

Fig. 1. Data dependency categories of assignments

In a PDG the edges represent dependencies of different types. The main categories are data dependencies and control dependencies. And there are many subcategories of data dependencies for different purposes specifically for different node types. It does not make sense to mix up these categories. E.g., an assignment is data dependent on the expression for the new value on the right-hand side and on a l-valued expression on the left-hand side that determines where the new value is stored. These two dependencies have completely different purposes. Fig. 1 shows the situation as a composite pattern in a UML class diagram. Further examples are the operands of non-commutative operators and the data dependencies of Φ-nodes. If the outgoing edges of each node are grouped into sections for the different categories, the matching process can handle each section separately and has to take into account only the inner permutations of the sections. Let k be the number of sections and n_i the number

of outgoing edges in section i: $n_1!+n_2!+...+n_k! < n!$, $\Sigma n_i=n$, $n_i>0$. Based on this observation the size of the PDG matching problem can be reduced significantly. Our approach is a generalization of the equivalence classes suggested by Krinke [3].

A special case is program code that does not use commutative operators and does not dereference pointers. In this case, $n_i=1$ for all sections and just one unique mapping is possible. We use a PDG variation that combines the PDG approach with SSA form by integrating Φ-nodes into the PDG. In conventional PDGs, $n_i>1$ at all join points, even without commutative operators and dereferenced pointers. The operands of commutative operators, like add or multiply operators, have the same purpose and are therefore grouped into the same section. Dereferencing pointers may cause n_i values greater than 1 as pointers usually have multiple target objects. Each object in the target set establishes a data dependency. To keep the n_i values small, it is important to use a precise pointer analysis as a basis of the clone detection.

The handling of sections with multpile entries can be further improved by introducing a partial order on expressions [4]. E.g., for two binary add operations x and y their summands are sorted according to the partial order on expressions: (x_1, x_2), $x_1 \leq x_2$ and (y_1, y_2), $y1 \leq y2$. If the summands can be ordered in such a way, no permutation is needed and x_1 is uniquely mapped to y_1 and x_2 to y_2.

If the performance is still not sufficient, the entries of a section can be merged and not distinguished any more in the further process, at the cost of reduced precision, as suggested in Krinke's approach [3]. But this is not necessary in general. A performance improvement is already gained by grouping dependencies into sections, without any loss of precision.

3 Avoiding Subclones

Another important issue is the selection of suitable start nodes for the subgraph matching. A naive approach would start a comparison at each node with every other node. A consequence of this quadratic scheme is that comparisons are started at nodes which are inside of larger clones. We call the resulting clones subclones of the encompassing clones. But usually one is interested in the maximal clones only. Several clone detection techniques filter subclones in a postprocessing step, e.g., the AST-based technique of Baxter [2]. In contrast, our approach avoids a repeated matching of many subclones in advance.

At first, we start with any node $r_1 \in V$ that represents an output value at the exit of a procedure. The backward slice S_1 spanned by this node is matched against the backward slices $S_{2,i}$ spanned by all nodes $r_{2,i}$ of the same node type that have not been processed yet. Then we mark r_1 as processed. The result of a succesful match between S_1 and $S_{2,i}$ is a clone $C = (C_1, C_{2,i})$ with $C_1 \subseteq S_1$, $C_{2,i} \subseteq S_{2,i}$ and a relation $M \subset V \times V$ that describes which node matches which other node. A matching starting with any $v_1 \in C_1$, $v_1 \neq r_1$ and $v_{2,i} \in C_{2,i}$, $v_{2,i} \neq r_{2,i}$ with $M(v_1, v_{2,i})$ provides a subclone of C. The unnecessary matching process starting at the root nodes v_1 and $v_{2,i}$ is skipped. Instead, the next match attempt starts with each $q_1 \in S_1$, $q_1 \notin C_1$, where q_1 is a direct successor of a node in C_1. q_1 is processed in the same way as described above. After all nodes reachable from r_1 are processed, the algorithm continues with the next procedure-output-value node. The set of output values encompasses return values, output parameters, and writing side effects. Nodes that are not processed, in the end, indicate dead code.

Although the worst case complexity is still quadratic, we expect the average performance of this algorithm to be much below that.

4 Conclusion

Although algorithms with exponential complexity are often viewed as unusable, in the context of PDG matching several performance improvements are possible. As PDGs provide a chance to push clone detections further towards the detection of semantic clones, it should be examined in more detail. A prototypical implementation of the ideas presented in this paper is currently under construction.

References

[1] Raghavan Komondoor and Susan Horwitz, "Using Slicing to Identify Duplication in Source Code," in Proc. of the 8th International Symposium on Static Analysis (SAS '01), London, UK, Springer-Verlag, 2001

[2] Chanchal Kumar Roy and James R. Cordy, "A survey on software clone detection research," technical report, Queen's University, Canada, 2007.

[3] Jens Krinke, "Identifying Similar Code with Program Dependence Graphs," in Proc. Eight Working Conference on Reverse Engineering (WCRE 2001), Stuttgart, Germany, pp. 301-309, October 2001

[4] Torsten Görg and Mandy Northover, "A canonical form of Arithmetic and Conditional Expressions," in Proc. of the 16th Workshop Software Reengineering & Evolution (WSRE 2014), Bad Honnef, Germany, 2014