

Edgar: the Embedding-baseD GrAph MineR

Marc Wörlein,¹ Alexander Dreweke,¹ Thorsten Meinl,² Ingrid Fischer², and Michael Philippsen¹

¹ University of Erlangen-Nuremberg, Computer Science Department 2
Martensstr. 3, 91058 Erlangen, Germany
{woerlein,dreweke,philippsen}@cs.fau.de

² ALTANA Chair for Bioinformatics and Information Mining
University of Konstanz, BOX M712, 78457 Konstanz, Germany
{Thorsten.Meinl,Ingrid.Fischer}@inf.uni-konstanz.de

Abstract. In this paper we present the novel graph mining algorithm Edgar which is based on the well-known gSpan algorithm. The need for another subgraph miner results from procedural abstraction (an important technique to reduce code size in embedded systems). Assembler code is represented as a data flow graph and subgraph mining on this graph returns frequent code fragments that can be extracted into procedures. When mining for procedural abstraction, it is not the number of data flow graphs in which a fragment occurs that is important but the number of all the non-overlapping occurrences in all graphs. Several changes in the mining process have therefore become necessary. As traditional pruning strategies are inappropriate, Edgar uses a new embedding-based frequency; on average, saves 160% more instructions compared to classical approaches.

1 Introduction

Nowadays embedded systems are used in many fields such as automotive, cell-phones, and various consumer electronics. As consumers demand more and more functionality, the programs that run on these systems grow. To reduce the per piece costs the manufacturers have to reduce the memory, space, and energy requirements. So they invest a great deal of effort in reducing the amount of code. Procedural abstraction (PA) is the most important technique to deal with code repetitions: frequent code fragments are extracted and substituted with jumps or procedure calls. Only one fragment remains in the resulting code.

This paper demonstrates that frequent graph mining can be applied successfully to enhance procedural abstraction. This section briefly introduces procedural abstraction and frequent graph mining. Section 2 discusses related work. Section 3 describes our new subgraph miner satisfying all requirements for procedural abstraction in detail. An evaluation is given in section 4.

1.1 Procedural Abstraction

Although there are compiler flags to avoid code bloat and smart linkers that reduce code size, still many space-wasting code duplications remain in the com-

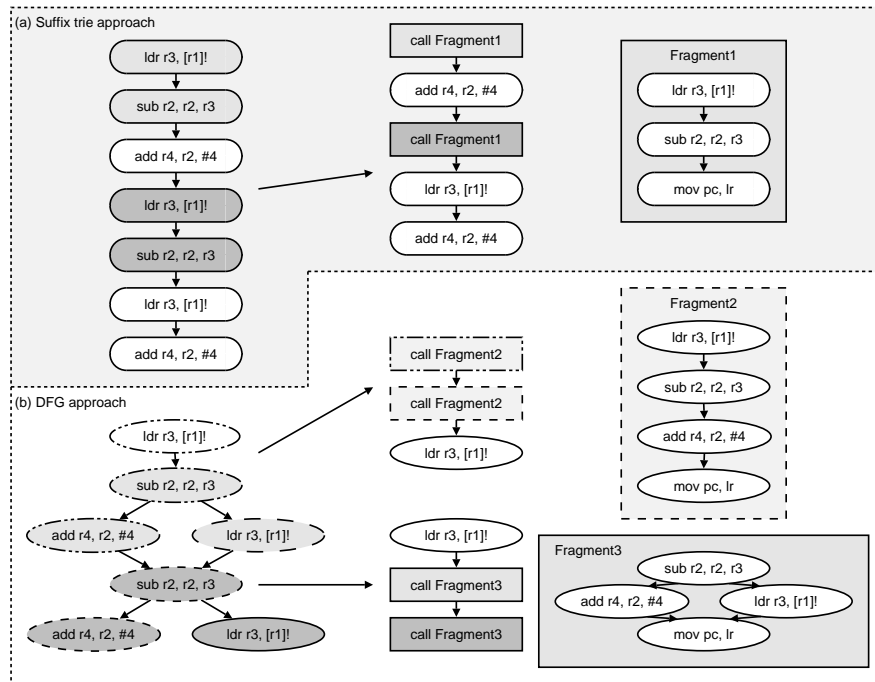


Fig. 1: Running PA example: (a) sequential PA on a basic block of ARM code and (b) the corresponding data flow graph extractions.

piled executables [4] (resulting from code templates, copy-and-paste programming, etc.). Code-size optimization is important for embedded systems [1] as cost and energy consumption depends on the size of the built-in memory and because more functionality fits into memory of a given size.

The main focus of code compaction based on PA is to detect repeated code fragments that can be extracted afterwards. Early compaction approaches considered the whole code as a sequence of instructions and used suffix tries to detect common subsequences [7]. Therefore fast algorithms that keep the compile-time short are used. For embedded systems, a larger time budget is available since the cost per piece is more relevant than compile time.

Optimizing compilers have passes that reorder the code sequence to e.g. keep the instruction pipeline perfectly filled. Therefore the traditional linear suffix trie approach cannot detect all semantically equivalent code, because the instruction order may differ from occurrence to occurrence. Our new graph-based PA approach transforms the instruction sequences of basic blocks³ into data flow graphs (DFG) which are independent of the actual scheduling.

³ A basic block is a piece of code that ensures that if one instruction is executed all following instructions will also be executed under all circumstances.

We explain the basic ideas of our approach to PA with the (synthetic) piece of ARM code and its corresponding DFG shown on the left side of Fig. 1. Our example steps through the values of an array (register `r1` points to the current element in the array) and performs some calculations. The upper part of Fig. 1 shows how the suffix trie based approaches detect the code fragment `ldr r3, [r1]! → sub r2, r2, r3` twice. The lower part shows that graph-based PA is more successful, even for this tiny example, because it finds two different fragments of size three that both appear twice and that both result in a smaller number of instructions. The varying order of instructions prevents suffix tries from detecting these fragments.

1.2 Frequent Subgraph Mining

Several frequent subgraph mining algorithms have been published that allow graph databases to be searched for frequent graph fragments by traversing the lattice of subgraphs [14]. An example of such a search lattice is given in Fig. 2. The empty subgraph (*) at

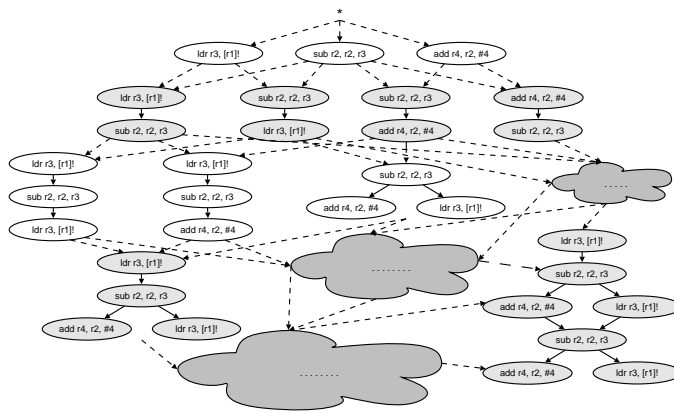


Fig. 2: Parts of the search lattice for the example.

the top is followed by the subgraphs with just one node (i.e. 1 instruction). Each step downwards adds a new edge (and node, if necessary) to a subgraph. Real search lattices are much more complex than this example as the underlying databases consist of many different and much bigger graphs.

A graph fragment is usually considered *frequent* if it appears in a given number of database graphs. Only the number of database graphs in which it occurs is counted, no matter how often a fragment is found in a single database graph.

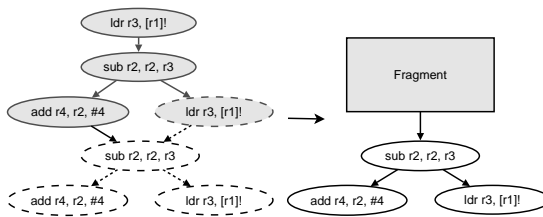


Fig. 3: Two overlapping embeddings

This counting scheme is inappropriate for PA where the total number of occurrences (called *embeddings*) of the graph fragment matters. In Fig. 1 the frequent fragment has a frequency of 1 since it appears twice but only in one graph.

However, not all embeddings are relevant for PA. If 2 embeddings overlap (as in Fig. 3) only the code for one of them can be extracted by PA. When one is extracted, the other embedding disappears. So only the number of non-overlapping embeddings matters.

2 Related Approaches

Within the graph mining community only a few algorithms have been published that satisfy the requirements of PA. The well-known algorithms such as MoFa, gSpan, Gaston, or FFSSM [15] use inappropriate graph-based frequency counting. Among the embedding-based algorithms, there are heuristic algorithms such as SubDue [3], GBI [12], or SEuS [8]. In contrast to Edgar, they only check heuristic subsets of fragments, and do not find the optimal candidate in general. The algorithms of Vanetik [13] and Kuramochi [11] are similar to our approach as they calculate the maximal independent set to determine edge-disjoint embeddings. But since two edge-disjoint embeddings can share a node (i.e. one instruction in our application) these algorithms are unsuitable for PA.

In the area of circuit synthesis on FPGAs (field-programmable gate arrays) similar problems arise [17]. High-level synthesis compilers produce recurring patterns that can be extracted to reduce the number of functional units. This so called *template generation* is similar to frequent graph mining. But there seems to be no attempt to do a complete detection as in our approach. Similar to us, [2, 12, 17] use a data flow graph-based approach to identify frequent patterns for a hard-logic implementation on embedded systems. However, their heuristics or restrictions to fragments with just one source node⁴ do not ensure that all relevant patterns are detected.

3 Edgar (Embedding-based GrAph mineR)

For mining DFGs we have developed a new mining algorithm called Edgar that is based on the well known gSpan algorithm [16]. We decided to start from gSpan because it is the best algorithm for mining with graph-based frequency [15].

3.1 gSpan

Fig. 2 shows parts of a search lattice. There are multiple paths to most subgraphs, which have to be filtered out to avoid duplicate detection of the corresponding fragments. Therefore gSpan builds a so-called *dfs-code* that represents the edges in the order in which they are added to the fragment. As there are different paths to each fragment, different codes exist. A global order on these codes allows the smallest one to be defined as canonical [16]. gSpan only extends fragments with canonical representation, because the non-canonical ones have already been created before.

⁴ A source node has just outgoing and no incoming edges.

To detect non-canonical codes, a straightforward but expensive graph isomorphism test can be used. To minimize these tests, gSpan groups all possible extending edges into forward edges that insert a new node, and backward edges that just close cycles. Now, gSpan only extends fragments with forward edges starting at nodes on the *right most path*⁵ and backward edges starting at the last added node. This heuristic method eliminates many unnecessary paths. Canonical tests are only required for the paths that remain.

For each fragment gSpan only stores a list of database graphs in which the fragment appears (a so-called *appearance list*). Its size represents the frequency of the fragment. To extend a fragment, only the graphs of that list have to be scanned (as opposed to the whole dataset).

3.2 Extending gSpan for PA

For Edgar, we have adopted the basic structures of the search lattice and gSpan's basic dfs-code including the right most extension rule. However, since the original gSpan algorithm works on undirected graphs, we had to extend the dfs-code, the extension process, and the test for being canonical to work on directed graphs needed for PA.

Edgar's frequency-based pruning significantly differs from gSpan's. For graph-based miners, the frequency of a fragment decreases monotonically with growing fragment size. Hence the search can be pruned as soon as an infrequent fragment is found. For embedding-based pruning *all* embeddings and not only the supported graphs are relevant. Therefore, first all embeddings have to be stored (not just the appearance lists). This leads to higher memory requirements.⁶ Second, since there are cases in which the frequency *increases* with growing fragment size (see Fig. 4) a corresponding frequency pruning is wrong.

Fortunately for PA it is not the set of all embeddings that matters, but the non-overlapping subset (see section 1.2). The ones with maximal size (the maximal number of embeddings) are the best, because extracting the embeddings of such maximal non-overlapping sets results in best code compression. Additionally, the maximal size decreases monotonically as the graph-based frequency does.

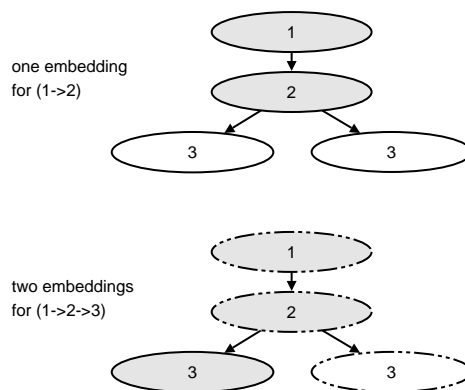


Fig. 4: Embedding counts.

⁵ This is the path of forward edges, between the first and the last inserted node.

⁶ It is better to keep embeddings rather than recomputing occurrences.

This is obvious, because a non-overlapping subset for each fragment can be generated by the embeddings of one of its successors. For each parent a non-overlapping subset with the same size can be generated out of the maximal subset of a child.⁷ This subset does not need to be a maximal subset, which therefore can contain the same number or more embeddings. For each successor such a subset can be found, thus the *maximal* subset for each fragment is greater than or equal to the sets of its children, which allows pruning like the original frequency.

Edgar builds a so-called collision graph to determine the size of a maximal non-overlapping subset. In the collision graph the embeddings of a fragment are represented as nodes which are connected if the corresponding embeddings overlap.

A maximum independent set in the collision graph then represents a subset of maximal non-overlapping embeddings on the graph database. Alternatively, a maximum clique in the inverted collision graph⁸ also represents the desired subset. For computing the maximum independent set/maximum clique – which is an NP-complete problem – standard algorithms (like [6]) can be used. Edgar adapts the maximal clique algorithm of Kumlander [10] which at the moment is the fastest known algorithm.

4 Evaluation

We have evaluated our approach with a subset of the MiBench suite [9]. For embedded systems it is sufficient to statically link against the small *dietlibc*⁹ [5]. Additionally all binaries were compiled with `-Os` to optimize for size. Table 6 compares the traditional suffix trie approach (SFX) with DgSpan, the directed graph-based gSpan algorithm, and with the embedding-based Edgar. Repeatedly for each

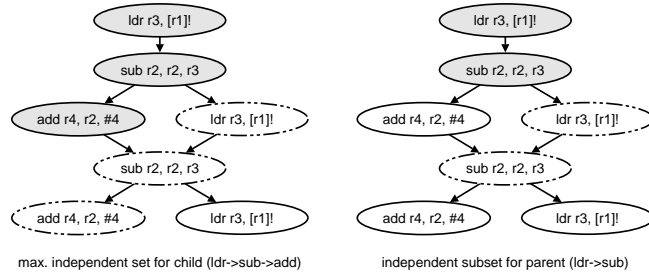


Fig. 5: A parent's subset induced by a child's maximal one.

Program	# Inst.	# of saved inst.		
		SFX	DgSpan	Edgar
bitcnts	3946	53	81	83
crc	3584	46	68	119
dijkstra	4632	70	102	164
patricia	5039	70	105	189
qsort	4770	70	105	197
rijndael	7113	70	132	256
search	3717	46	74	110
sha	3897	55	82	120
total	36698	480	749	1238

Fig. 6: Saved instructions

⁷ In the example in Fig. 5 only the unused `add r4, r2, #4` has to be removed

⁸ The inverted collision graph has an edge, if two nodes do not overlap

⁹ Dietlibc is a cross platform C run-time library, supporting x86, ARM, Sparc, Alpha, PPC, Mips, and s390 architectures, compatible with SUVv2 and POSIX.

approach, in each iteration the best fragment has been extracted until no further shrinking was possible. The fragments are weighted by the number of instructions they save. In total, Edgar saves 1238 as opposed to 480 instructions by SFX which is an improvement by a factor of 2.6. Edgar even saves more than DgSpan, because DgSpan misses many fragments that appear several times in the same basic block.

From the related work (section 2) only SubDue is available for download. Therefore we have simply compared Edgar and SubDue quantitatively. In both algorithms we used the same weighting function $(graphSize - 1) * (frequency - 1)$ to estimate the instruction savings for a given fragment.

edges	200	300
Connectivity	3	2
Coverage	0.7	0.8
Overlap	0.7	0.2
Deviation	0.9	1.0
SubVertexLabel	0.30	0.25
SubEdgeLabel	0.20	0.25
DelVertex	0.15	0.10
DelEdge	0.35	0.40
Substruct. nodes	3	6
Substruct. edges	6	5

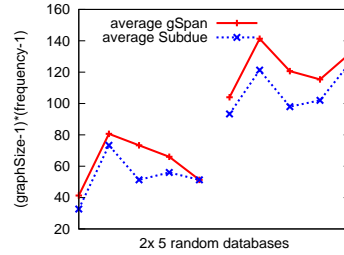


Fig. 7: SubDue vs. gSpan: best found fragments

For Edgar the support is set to two to find the overall best fragments. A real substitution of the found fragment is not trivial, so just the first run for both algorithms is taken for the results in Fig. 7. For the comparison ten different random graphs with uniformly distributed node and edge labels¹⁰ - five with the parameters of the first column of the given table and the other half with the second values - are generated with the *subgen* tool of the SubDue-system. Fig. 7 shows the average weight over the best three fragments (default configuration of SubDue) for each algorithm. In most cases SubDue found the same fragments as Edgar but not the optimal number of independent embeddings, which lead to lesser ranking of those fragments.

5 Conclusion

Frequent graph mining renders PA more effective and helps to build cheaper and/or more powerful embedded systems. By using DFGs instead of code sequences as a basis of frequent code detection more potential for outlining is found. The better the candidates are with respect to size and frequency, the smaller the resulting code. Compared to other sequential approaches our prototype Edgar saves an average of 160% more instructions.

In contrast to Edgar, common graph-based miners cannot be applied, since they do not work on directed graphs, they use heuristics that prune the search space prematurely, stumble over fragments that have nodes in common, or lack embedding-based frequency counting. The paper presented a new embedding-based frequency definition that is based on the maximal non-overlapping subset of all embeddings and results in a valid pruning strategy.

¹⁰ 25 node and 4 edge labels

References

1. J. Bentley. Programming Pearls: Squeezing Space. *Communications of the ACM*, 27(5):416–421, May 1984.
2. P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 262–269, New York, 2002.
3. D. J. Cook and L. B. Holder. Substructure Discovery Using Minimum Description Length and Background Knowledge. *Artificial Intelligence Research*, 1:231–255, 1994.
4. S. K. Debray, W. Evans, R. Muth, and Bjorn de Sutter. Compiler Techniques for Code Compaction. *ACM Trans. Programming Languages and Systems*, 22(2):378–415, 2000.
5. dietlibc - a libc optimized for small size. <http://www.fefe.de/dietlibc/>.
6. F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and Conquer: A Simple $o(n^{0.288n})$ Independent Set Algorithm. In *Proc. of 17th ACM-SIAM Symp. on Discrete Algorithms*, pages 18–25, Miami, FL, January 2006.
7. C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and Compressing Assembly Code. In *Proc. ACM Symp. on Compiler Construction*, pages 117–121, Montréal, C, 1984.
8. S. Ghazizadeh and S. S. Chawathe. SEuS: Structure extraction using summaries. In *Discovery Science*, volume 2534, pages 71–85. Springer, 2002.
9. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. 4th IEEE Workshop on Workload Characterization*, pages 3–14, Austin, TX, 2001.
10. D. Kumlander. A new exact Algorithm for the Maximum-Weight Clique Problem based on a Heuristic Vertex-Coloring and a Backtrack Search. In *Proc. 5th Int'l Conf. on Modelling, Computation and Optimization in Information Systems and Management Sciences*, pages 202–208, Metz, France, July 2004.
11. M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Min. Knowl. Discov.*, 11(3):243–271, November 2005.
12. P. C. Nguyen, K. Ohara, H. Motoda, and T. Washio. Cl-gbi: A novel approach for extracting typical patterns from graph-structured data. In *Advances in Knowledge Discovery and Data Mining, 9th Pacific-Asia Conference Proc.*, volume 3518, pages 639–649, Hanoi, Vietnam, May 2005. Springer.
13. N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proc. IEEE Int'l Conf. on Data Mining ICDM*, page 458, Maebashi City, Japan, November 2002.
14. T. Washio and H. Motoda. State of the Art of Graph-based Data Mining. *SIGKDD Explorations Newsletter*, 5(1):59–68, July 2003.
15. M. Wörlein, T. Meinel, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In *Knowledge Discovery in Database: PKDD 2005*, volume 3721, pages 392–403, Berlin, 2005. Springer.
16. X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proc. IEEE Int'l Conf. on Data Mining ICDM*, pages 721–723, Maebashi City, Japan, November 2002.
17. D. Zaretsky, G. Mittal, R. P. Dick, and P. Banerjee. Dynamic template generation for resource sharing in control and data flow graphs. In *19th Int' Conf. on VLSI Design*, pages 465–468, Hyderabad, India, January 2006.