ABSTRACT
We present how to efficiently mine a set of directed acyclic graphs (DAGs) for unconnected, both multi- or single-rooted, and induced fragments. With a new canonical form that is based on the nodes' topological levels, our miner is faster and uses less storage than general purpose gSpan (Yan, X. and Han, J., 2002). Moreover, it can base support resp. frequency either on the number of embeddings of a fragment in the database or on the number of graphs a fragment appears in. This is crucial for finding frequent fragments in data flow graphs generated from assembly code. Extracting them into new procedures reduces the total code size. The paper shows that our miner outperforms general purpose mining and demonstrates the quantitative effects of DAG mining in program size reduction.

KEYWORDS
Graph Mining, Compiler Construction

1. DIRECTED ACYCLIC GRAPHS IN PROCEDURAL ABSTRACTION

DAGs are important in code generation, especially in code compaction. Code-size optimization is crucial for embedded systems as cost and energy consumption depend on the size of the built-in memory and since with smaller code more functionality fits into it. Procedural Abstraction (PA) attacks code repetitions in assembly code: frequent code fragments are extracted into new procedures and substituted with jumps or procedure calls. Traditionally, identically repeated code sequences are found by means of suffix tries generated for the textual instruction sequences (Debray, S. et al., 2000). We have shown in (Dreweke, A. et al., 2007) that a graph-based PA that first transforms the instruction sequences of basic blocks1 into data flow graphs (DFG) representing the data dependencies between instructions and then mines for frequent graph fragments can achieve much better code shrinking results because, conceptually, all semantically equivalent instruction reorderings are considered.

A basic block is code that has one entry point (i.e., no code within it is the destination of a jump instruction), one exit point and no jump instructions contained within it.
This domain for graph mining has the following requirements that our DAG miner addresses up front. First, it is designed to find unconnected fragments (i.e. frequent sub-DAGs in the DAG database). To see that this is important for PA, consider the DFG for a piece of synthetic ARM assembly shown on the left side of Figure 1. There is an unconnected fragment that appears twice (gray and dashed). This fragment can be extracted into a procedure as shown on the right. The two unconnected parts in this fragment can be executed in any order, since there are no data dependencies between them. Existing miners for connected graphs do not easily find this fragment. Instead, multiple starting points (Borgelt, C. and Berthold, M., 2002) or an additional node connected to all other nodes (Yan, X. and Han, J., 2002.) have to be used. Second, our miner can calculate support/frequency of fragments embedding-based. The embedding-based support is the minimal absolute number of embeddings. Embeddings are the actual appearances of the fragment in the database. For frequency the minimal number of appearances is given in percent. The support in Figure 1 is 2. Furthermore, our miner searches for induced fragments that preserve the parent-child relationship of each node in the original DAG. This is in contrast to embedded fragments that preserve the more general ancestor-descendant relationships over several nodes and edges. Finally, our miner can handle fragments with multiple roots. A root is a node without incoming edge. The fragment in Figure 1 has two roots.

Whereas there are algorithms for (undirected) mining of trees (Muntz, C. et al., 2005) and graphs (Meinl, T. and Fischer, I., 2008), the situation is different for directed acyclic graphs. The only three other DAG miners known to us (Termier, A. et al., 2006; Chen, Y. et al., 2004; Zaretsky, D. et al., 2006) are not applicable to PA. The first is a miner for gene network data that does not handle induced but embedded fragments. The second and third address single-rooted and connected sub-DAGs only. Furthermore, the Chen et al. overlooked a special case while constructing their pyramid pattern.

The DAG Mining Algorithm (DAGMA) presented in this paper is more general; it cannot only be used for the application domain mentioned above. In addition to mining with embedding-based support, our DAG miner can also mine with graph-based support, i.e., the number of graphs (DAGs) a fragment appears in is counted. Also, it can mine for connected fragments by filtering unconnected ones in an additional step, or it can mine for single-rooted fragments by using only one root node as a starting point.

2. ALGORITHMIC DETAILS OF DAGMA

Systematic graph miners build a search lattice by starting from single-node fragments and expand them by nodes and/or edges in a stepwise fashion. For a general introduction to graph mining see for example (Meinl, T. and Fischer, I., 2008).

2.1 DAG mining is NP-complete

Whereas this search lattice can be enumerated in polynomial time for trees, general graph mining is NP-complete because subgraph isomorphism is NP-complete. Graph isomorphism is supposed to be in a complexity class of its own (Fortin, S., 1996). Unfortunately, (sub-)DAG isomorphism problems are in in the same complexity classes.

As a proof, consider the following transformation of an undirected graph into a DAG: In the middle of each undirected edge, a new node is inserted. The two 'halves' of the undirected edge are replaced by two directed edges that point to the newly inserted node. Obviously, the transformed graph is a DAG since every old node has only outgoing, every new node has only incoming edges. This transformation (and the inverse one) can be done in polynomial time and the increase of nodes and edges remains polynomial. The transformation is a valid reduction: If two graphs are isomorphic to each other, they also are isomorphic after transformation. If a graph contains a subgraph, its transformed graph contains the transformed subgraph. Hence, each (sub-)graph isomorphism problem can be solved by solving the corresponding (sub-)DAG isomorphism problem.

2 Note that in research on code compaction, 'template generation' or 'clone detection' are the key words used to denote related forms of DAG mining.
3 If the source graph is a directed graph, edge labels on the newly introduced edges represent the direction of the original edge.
2.2 A Canonical Form for DAG enumeration

Because of the NP-completeness, one of the challenging problems for DAG mining remains to avoid as many costly (sub-)DAG isomorphism tests as possible. If the application area has special constraints for the input data like distinct labels in each input graph, these tests can be optimized, but for general DAGs avoidance is the best way. Therefore, the enumeration has to quickly detect duplicates, i.e., fragments that are reached through several paths. As usual this is done by encoding fragments in a canonical form that is simple to construct and more amenable to comparison than costly subgraph isomorphism tests.

The fundamental idea of DAGMA is a novel canonical form that exploits a certain DAG property. The main difference between directed graphs with cycles and directed acyclic graphs is that DAGs are topologically sortable in linear time with respect to the number of nodes and edges (Cormen, T. et al., 2001). This way each node has a topological level based on the length of the longest path from a root node to the node itself. The DAGs given in Figure 2 have two topological levels indicated with roman numbers. The main idea is the systematical construction of fragments by inserting nodes and edges only in topological order.

![Figure 2. Two isomorphic subgraphs with different canonical forms.](image)

DAGMA’s canonical form of a DAG contains the full information about the graph structure, the edge directions, and the insertion order according to the enumeration of the search lattice. In Figure 2(a) a fragment is expanded from the single root $B$ that was inserted in the first step (denoted by the small node insertion index 1). The second insertion step inserts another node $B$. Step 3 simultaneously inserts the node $A$ and the edge from its predecessor $B$ (denoted by the node index 3 and the edge index 3.0). An edge to the previously inserted node (edge index 3.1) is inserted in the last insertion step without adding a new node simultaneously. The 0 in 3.0 indicates that the edge was inserted together with node 3. 3.1 means that this is the first edge targeting node 3 that has been inserted after the insertion of this node.

The canonical descriptions given below the fragments in Figure 2 consist of tuples of the form (node label index, predecessor index). For efficiency we sort and number labels according to their frequency. For simplicity, let us assume a lexicographic ordering of label indices ($A = 0$, $B = 1$, $C = 2$, ...). Hence, the first tuple $(1,0)$ tells us that node $B$ has been inserted first, and that it has no predecessor (0 as predecessor index). In general, the predecessor index refers to the insertion step of its parent node. For example, the tuple $(0,2)$ indicates that $A$ is inserted with node $B_2$ as its predecessor. If a single edge (e.g. 3.1) is inserted without adding a new node simultaneously, a special symbol $n$ that is bigger than all other label indices is used as the node label index. See tuple $(n,1)$ in the figure. This edge is connected to the last added node.

A canonical fragment is created by an insertion order of nodes and edges with the lexicographically biggest description. Thus, the fragment in Figure 2(b) that has a different edge insertion order is not canonical and can be pruned during the enumeration of the search lattice without any isomorphism test, because there is another insertion order for the fragment's nodes and edges with a bigger canonical form (see Figure 2(a) for the maximal possible one).

---

4 Edge label indices can be handled similar to node label indices and are omitted here to simplify explanation.
5 This canonical form requires and the creation order relies on the constraint that nodes are inserted topologically; therefore, there is no need to insert outgoing but only incoming edges.
Basic Structure of the DAG Mining Algorithm

The DAG miner (see Algorithm 1) computes an initial set of frequent single-node fragments (line 2) and starts to expand from them. Because of the canonical form, fragments are expanded according to the following rules that respect the topological sorting:

1. insert a new root node (line 9),
2. start a new topological level (i.e., insert a new node and a new edge, line 10),
3. stay at current topological level by inserting a new node (line 11),
4. insert a new single edge to the previously inserted node (whose predecessor has already been inserted, line 12).

These steps predominantly generate canonical fragments: only when rule 4 inserts single edges, duplicates may be generated. These are pruned afterwards (line 12). Finally, fragments that are inextensible or unwanted due to some application specific reason can be filtered out (line 14 and 16).

Algorithm 1  
DAG mining.

<table>
<thead>
<tr>
<th>Data:</th>
<th>database with DAGs db</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>frequent sub-DAGs res</td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>res ← Ø</td>
</tr>
<tr>
<td>2.</td>
<td>n ← frequentNodes(db)</td>
</tr>
<tr>
<td>3.</td>
<td>e ← frequentEdges(db)</td>
</tr>
<tr>
<td>4.</td>
<td>l ← createLabelFunction(n, e)</td>
</tr>
<tr>
<td>while n ≠ Ø do</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>res ← res ∪ n</td>
</tr>
<tr>
<td>7.</td>
<td>tmp ← Ø</td>
</tr>
<tr>
<td>8.</td>
<td>for f ∈ n do</td>
</tr>
<tr>
<td>9.</td>
<td>tmp ← tmp ∪ insertRoots(f, l)</td>
</tr>
<tr>
<td>10.</td>
<td>tmp ← tmp ∪ insertLevel(f, l)</td>
</tr>
<tr>
<td>11.</td>
<td>tmp ← tmp ∪ insertNode(f, l)</td>
</tr>
<tr>
<td>12.</td>
<td>tmp ← tmp ∪ pruneNonCanonical(insertEdge(f, l))</td>
</tr>
<tr>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>n ← filterInExtendibleFragments(tmp)</td>
</tr>
<tr>
<td>15.</td>
<td>end while</td>
</tr>
<tr>
<td>16.</td>
<td>res ← filterUnWantedFragments(res)</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

2.3 Example of Search Lattice Enumeration by Expansion Rules

Figure 3 shows a complete search lattice for a database that has only one graph (shaded gray) to keep the example simple. Different types of edges represent the expansion rule applied. The search lattice nicely demonstrates that none of the fragments (or embeddings) is visited twice, although without a canonical form most subgraphs could have been reached by several paths. I.e., in the example the enumeration and its restriction to consider only maximal canonical forms discard all duplicates. After the initialization, three single-node fragments are in the search lattice.

Rule 1. Because of the topological creation order, the first insertion rule new root can be applied to fragments on the first topological level without edges. The canonical form helps avoid duplicates by defining an order on the node labels and inserting no node with a label later in the order as the last one. Consider for example the initial fragment B in Figure 3. This is just extended with the additional root node A and not with the node C since the fragment (B, C) is already present as (C, B) which has a bigger canonical form.

---

6 This rule is similar to the candidate item set generation procedure described in (Agrawal, R. et al., 1993).
Rule 2. When a new topological level is started by the insertion of a node, the expansion of the current topological level is completed and no further node can be inserted at that level. It is easy to avoid duplicates in this phase of the algorithm by checking partitions that are the basis of graph isomorphism tests (McKay, B., 1981). Initial partitions of a subgraph are created by the indegree, outdegree, and node label index of every node. They are then iteratively refined based on their neighboring partitions until no more refinements are possible. The resulting partitions (automorphism groups) give evidence about the symmetries in a graph. This is possible in polynomial time. Regardless of the node selected as predecessor within a partition, the resulting graphs would be isomorphic. Therefore, a new level can only be started canonically by using the last inserted node of a partition as the predecessor (Werth, T., 2007).

Rule 3. The insertion of a new node at the current level is similar to the previous rule and does not generate duplicates, either. First of all, the partition check has to be applied, again. Since the node label index is the most significant element of each tuple in the canonical form, the next node inserted must have a smaller (or equal) index than its predecessor. In Figure 3 this rule is only used once to generate fragment $C \rightarrow (A,B)$. The canonical description of this fragment is $(2,0)(1,1)(0,1)$. The same fragment could have been reached from the fragment $C \rightarrow A$. But this would have produced the description $(2,0)(0,1)(1,1)$ which is lexicographically smaller and hence pruned. If the labels (and therefore the node label indices) are equal, the predecessor index has to be smaller or equal to the predecessor index of the previously inserted node to achieve the maximal canonical description.

Rule 4. Probably the most difficult expansion rule is the insertion of a single edge without adding a node simultaneously. These single edges can only target the last inserted node. As before, pruning is based on partitions and predecessor indices. Additionally, the set of predecessors of the current and the last inserted node are compared to exclude other insertion orders with bigger canonical descriptions (Werth, T., 2007). This approach avoids a good portion but not all of the remaining duplicates in the search space.\(^7\) One example of a duplicate that is not avoided by this rule and its insertion order of nodes is shown in Figure 4.

\(^7\) Complete avoidance may be possible, but has to be NP-complete due to the NP-complexity of sub-DAG mining.
The canonical form of the fragment in Figure 4(b) is bigger, so the other one needs to be pruned from the search lattice by application of an exponential is-Canonical test.

For each new fragment generated with one of the four rules, the support or frequency must be calculated. Support or frequency can be given based on the number of graphs a fragment appears in or on the number of embeddings. Embedding-based mining may face an additional complexity since, in general, embeddings of a fragment can overlap. If the application domain is interested only in non-overlapping embeddings (e.g., PA can only extract non-overlapping embeddings into procedures), our DAG miner calculates the maximal set of non-overlapping embeddings of a fragment. See (Dreweke, A. et al., 2007) for details.

3. EVALUATION

To evaluate our DAG miner we compare it to gSpan, the most general graph miner currently available. Both algorithms are implemented in Java (as part of a comprehensive suite of miners, see (Wörlein, M. et al., 2005)) and run on an AMD Opteron with 2 GHz and 11 GB of main memory; our execution environment is a Sun JVM version 1.5.0. For comparison we have used a synthetic DAG databases, a worst case database, and a database from our application domain, i.e., procedural abstraction to shrink assembly code size.

Synthetic DAGs are generated as follows: Every node and edge that is reachable from a randomly selected node in a big random master DAG is copied into the synthetic DAG database (Chen, Y. et al., 2004). This way, the graphs automatically contain similarities. For the benchmark, we generate the database out of a master DAG that contains 50 nodes, 200 edges, and 10 labels. We restrict sub-DAGs to 5 topological levels or 25 nodes. Every test on random data behaved similarly to the results shown.

Figure 5 and Figure 6 present the runtime comparison of the DAG miner and gSpan for both embedding-based as well as graph-based support. For both types of support, our DAG miner outperforms gSpan, except for connected graphs (Figure 5b and Figure 6b). This is because we search for unconnected fragments first and then filter the results afterwards. For decreasing minimal support resp. increasing number of embeddings, the differences are more prominent.

The worst case for DAG (and graph) miners is a DAG with the maximal number of edges in which all nodes have the same label. Such a maximal DAG with seven equally labeled nodes served as input for the results shown in Figure 7 and was searched for frequent fragments with embedding-based minimal support 1. In that case 2,895,493 embeddings can be found. Again, our DAG miner clearly outperforms gSpan with respect to both runtime and memory consumption. Its performance gets better from single-rooted mining over connected mining to unconnected mining. The main advantage and the reason for this behavior become apparent in Figure 7c: Due to its DAG-specific canonical form, our DAG miner has to handle far less duplicates in costly isomorphism tests than gSpan. The same holds for the artificial databases.
To evaluate our algorithm for procedural abstraction, we transformed several ARM assembly codes from the MiBench suite (Guthaus, M. et al., 2001) into DFGs and mined embedding-based. The program size after extracting the frequent fragments is compared to the original program size, see (Dreweke, A. et al., 2007) for more details. Figure 8 gives the savings in code size for different programs when mining with suffix tries in contrast to our DFG-based approach with DAGMA. Our search for unconnected fragments leads to significantly better results than the traditional approach.

4. CONCLUSION AND FUTURE WORK

We presented a new DAG miner that can search for induced, unconnected or connected, single- or multi-rooted fragments in DAG databases, both with embedding-based and graph-based calculation of support/frequency. Our novel canonical form and the basic lattice expansion steps rely on the fact that DAGs have topological levels. Since DAGMA faces fewer duplicates compared to the general graph miner gSpan, it runs faster and consumes less memory. When applied to Procedural Abstraction, DAGMA achieves more code size shrinking than traditional approaches.
We have repeatedly faced graphs that are too large to be mined at a low support. Procedural Abstraction traditional searches with a minimal support of 2 embeddings to get the best possible results. Hence, it seems necessary to study heuristics that can be applied during the mining process. This problem will become even more pressing and will probably require parallel DAG mining, when in future Procedural Abstraction considers not only DFGs of basic blocks but moves on to much larger DFGs of whole programs.

REFERENCES


Termier, A. et al., 2006. Mining Closed Frequent DAGs from Gene Network Data with Dryade. Proc. 20th Conf. of the Japanese Society for Artificial Intelligence, Tokyo, Japan.


Yan, X. and Han, J., 2002. gSpan: Graph-Based Substructure Pattern Mining. Proc. Int'l Conf. on Data Mining (ICDM '02), Maebashi City, Japan, IEEE Computer Society, pp. 721-724.