Chapter 1
DAG Mining for Code Compaction

T. Werth[1], M. Wörlein[1], A. Dreweke[1], I. Fischer[2], and M. Philippsen[1]

Abstract In order to reduce cost and energy consumption, code–size optimization is an important issue for embedded systems. Traditional instruction saving techniques recognize code duplications only in exactly the same order within the program. As instructions can be reordered with respect to their data dependencies, Procedural Abstraction achieves better results on data flow graphs that reflect these dependencies. Since these graphs are always directed acyclic graphs (DAGs), a special mining algorithm for DAGs is presented in this chapter. Using a new canonical representation that is based on the topological order of the nodes in a DAG, the proposed algorithm is faster and uses less memory than the general graph mining algorithm gSpan. Due to its search lattice expansion strategy, an efficient pruning strategy is applied to the algorithm while using it for Procedural Abstraction. Its search for unconnected graph fragments outperforms traditional approaches for code–size reduction.

1.1 Introduction

We present DAGMA, a new graph mining algorithm for Directed Acyclic Graphs (DAGs). DAG mining is important in general, but our work is inspired by DAGs that appear 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. Moreover, the smaller the code is, the more functionality fits into the memory.

Procedural Abstraction (PA) reduces assembly code size by detecting frequent code fragments, extracting them into new procedures and substituting them with

Programming Systems Group, Computer Science Department, University of Erlangen–Nuremberg, Germany, phone: +49 9131 85-28865, e-mail: {werth, woerlein, dreweke, philippsen}@cs.fau.de · Nycomed Chair for Bioinformatics and Information Mining, University of Konstanz, Germany, phone: +49 7531 88-5016, e-mail: Ingrid.Fischer@inf.uni-konstanz.de

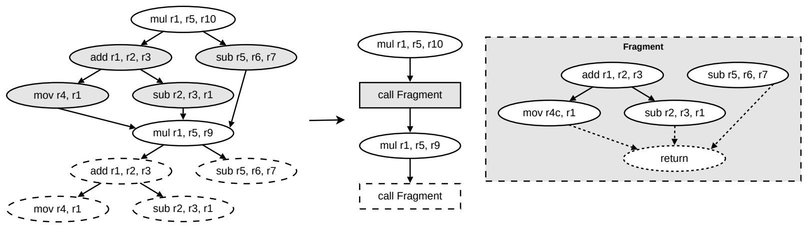


Fig. 1.1 PA example: data flow graph with frequent unconnected fragment (gray and dashed).

call/jump instructions. Traditionally, text–matching, e.g. suffix trees [6], is used for candidate detection. The obvious disadvantage is that to detect repeated instructions they must occur in the exact same order in the program. Dreweke et al. have shown in [7] that a graph–based PA outperforms the code shrinking results of a purely textual approach. Graph–based PA transforms the instruction sequences of basic blocks into data flow graphs (DFGs). A basic block is a code sequence that has exactly one entry point (i.e. only the first instruction may be the target of a jump instruction) and one exit point (i.e. only the last instruction may be a jump instruction). A DFG is a DAG that represents the instructions as nodes and the data dependencies between these instructions as directed edges. In general, from a single DFG several differently ordered instruction sequences can be generated that have the same semantics but cannot be detected by textual approaches. A graph–based approach, i.e. mining for frequent DFG fragments, can therefore find more opportunities for PA.

This chapter will show that for our domain DAG mining is better than regular graph mining. In addition, DFG–based code compaction has the following domain-specific requirements. While it is hard to extend general purpose miners accordingly, our DAG Mining Algorithm DAGMA addresses them up front. First, DAGMA can find *unconnected* fragments. This is crucial for PA as shown in Fig. 1.1. In the example DFG an unconnected fragment appears twice and is extracted into a procedure. Existing miners for connected graphs cannot find such unconnected fragments without applying tricks. For example, they use multiple starting points [2] to grow fragments or they add a helper pseudo–root that is connected to all other nodes [16]. Of course DAGMA can also search for connected fragments with one or more roots. Second, in addition to the traditional *graph–based* way to compute support/frequency of a fragment, DAGMA can also calculate it in an *embedding–based* way. Whereas graph–based counting detects that the frequent fragment of Fig. 1.1 appears in one graph (i.e. support = 1), an embedding–based counting distinguishes the two (non-overlapping) embeddings (i.e. support = 2). Since PA can extract both embeddings, PA requires an embedding–based support calculation. To be more exact, PA requires a search for *induced* fragments, because not every *embedded* fragment can be extracted. More details are given in Section 1.3.2.

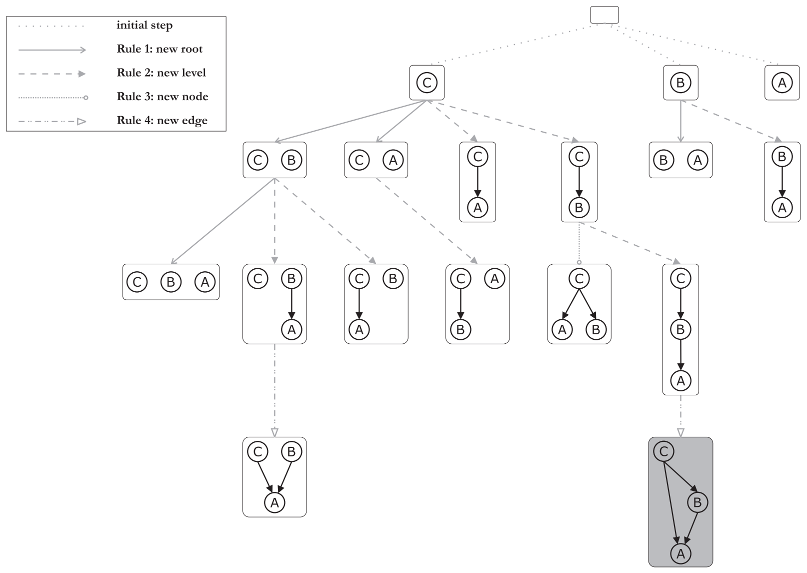


Fig. 1.2 Example search lattice.

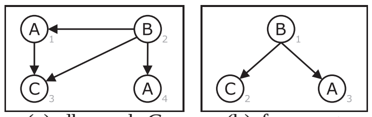
1.2 Related Work

Whereas there is a bunch of algorithms for (undirected) mining in trees [4] and graphs [16, 12, 2], the situation is different for DAGs. The only three other DAG miners known to us [14, 3, 17] are not applicable to PA. The first is a miner for gene network data that does not handle induced but just embedded fragments. The others only address single–rooted and connected sub–DAGs and therefore detect fewer extractable fragments. (Note that in research on code compaction, ‘template generation’, ‘clone detection’, or ‘regularity extraction’ are the key words used to denote related forms of DAG mining.) DAGMA is more general and can be used for more than just PA. DAGMA can mine both with embedding–based and with traditional graph–based support and it can also mine for connected fragments (by filtering unconnected ones) or for single–rooted fragments (by using only one root).

After covering some preliminaries, we present DAGMA in detail in Section 1.4. To ease applicability of DAGMA to other DAG problems we focus on its general purpose aspects and only address PA specifics when needed. Section 1.5 gives performance results both on synthetic DAGs and on DFGs of ARM assembly programs.

1.3 Graph and DAG Mining Basics

Most graph miners build their *search lattice* by starting from single–node or single–edge graph *fragments* (i.e. common subgraphs of the database graphs) and grow



(a) db graph *G* (b) fragment

Fig. 1.3 Example of one fragment in two database graphs.

them by adding nodes and/or edges in a stepwise fashion based on a set of expansion rules. Fig. 1.2 holds an example search lattice and shows the relationship between the fragments (nodes) and their expansions (the directed edges) that grow a fragment from another one. The concrete instances of a fragment, i.e. the appearances of its isomorphic subgraphs in the database graphs, are called *embeddings*. A fragment is *frequent* and therefore interesting, if its number of embeddings, counted after each expansion step, exceeds some threshold. This can be a concrete number, denoted by *minimum support*, or a percentage value of the database graphs, denoted by *minimum frequency*. The search process prunes the search lattice at infrequent fragments, since extending such a fragment always leads to other infrequent fragments, so further extensions are worthless (known as *anti–monotone principle*). The main difficulty is to avoid traversing multiple paths to already created fragments, i.e. a fragment should not be generated again if it has already been reached by another sequence of expansion steps. Otherwise such a fragment is processed again, including the detection of embeddings in the database. Since this is costly with respect to space and time, it is essential to check for duplicates efficiently.

1.3.1 Graph–based versus embedding–based mining

There are two interpretations of minimum support. Support of a fragment in *graph–based* mining specifies the minimum number of database graphs with one or more embeddings of this fragment. *Embedding–based* mining defines support as the minimum number of non–overlapping embeddings regardless of the database graphs. The number of non–overlapping embeddings is computed by means of a maximum independent set algorithm [13], for PA this process is described in [7].

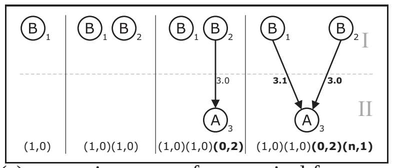
As mentioned above, the fragment shown in Fig. 1.1 has a graph–based support of 1 but an embedding–based support of 2. In contrast, the example in Fig. 1.3 shows a graph *G* and a fragment. Although there are two ways to embed the fragment into *G*, the embedded–based support is just 1, since the two ways of embedding overlap. There are two main reasons for only taking non–overlapping reasons into account. First, PA requires an embedding–based mining because only non-overlapping fragments can be used to shrink the code. An extraction of an embedding replaces all its nodes with a single instruction (call or jump) and therefore afterwards the extraction of an overlapping second fragment is no longer possible. Second, only for edge–disjoint (and therefore disjoint) embeddings the anti–monotone principle can be used to prune the search lattice. If we would also count overlapping embeddings, it is no longer true that the minimum support monotonously decreases with growing fragment sizes.

1.3.2 Embedded versus induced fragments

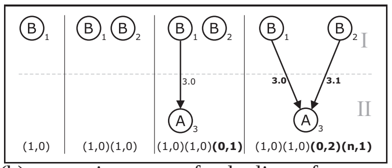
Induced fragments are a subset of *embedded fragments*, because of the more strict parent–child relationship in contrast to the more general ancestor–descendant relationship. Induced fragments are real subgraphs of the database graphs, because directly connected nodes also have to be directly connected in the corresponding database graphs. Nodes that are directly connected in an embedded fragment have to be connected in the original graph but the connection may be a path over several nodes and edges. Therefore, a parent node in the embedded fragment has to be an arbitrary ancestor and a child node must be a descendant in the database graph. Only induced fragments are useful for PA, since embedded fragments can skip nodes. For example, if the chain $A \longrightarrow B \longrightarrow C$ is used as input, an possible embedded but not induced fragment is $A \longrightarrow C$ that ignores the dependency of the node B.

1.3.3 DAG mining is NP–complete

Whereas the 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 [8]. Unfortunately, sub-DAG isomorphism is in the same complexity class. As a proof, consider the following transformation of a general graph into a DAG: Replace each original edge with a new node (carrying the edge label, if existent) plus two directed edges from the old nodes to the new node. If the source graph is a directed graph, edge labels represent the direction. Obviously, the transformed graph is a DAG since every old node has only outgoing and every new node has only incoming edges. Since this transformation (and the inverse one) can be done in polynomial time and the increase of nodes and edges is polynomial, the transformation is a valid reduction. If two original graphs are isomorphic to each other, they also are isomorphic after the transformation. If a graph contains a subgraph, its transformed graph contains the transformed subgraph. The inverse reduction is obvious, since the DAG can be treated as a general graph. Hence, each (sub-)graph isomorphism problem can be solved by solving the corresponding (sub-)DAG isomorphism problem and vice versa. As a result, DAG isomorphism is in the same complexity class as graph isomorphism and sub-DAG isomorphism is in the same complexity class as subgraph isomorphism and therefore DAG mining is NP–complete.



(a) expansion steps of a canonical fragment



(b) expansion steps of a duplicate fragment

Fig. 1.4 Construction of two isomorphic subgraphs with different canonical forms.

1.4 Algorithmic Details of DAGMA

Because of the *NP*–completeness, one of the challenging problems for DAG mining is to avoid as many costly (sub-)DAG isomorphism tests as possible. The enumeration of the fragments has to quickly detect duplicates, i.e. fragments that are reached through several paths. As other mining algorithms do, DAGMA solves this by encoding fragments in a canonical form that is both simple to construct and more amenable to comparison than costly subgraph isomorphism tests.

1.4.1 A Canonical Form for DAG enumeration

The fundamental idea of DAGMA is a novel canonical form that exploits that DAGs can be sorted topologically (in linear time with respect to the number of nodes and edges [5]). This way each node has a *topological level* based on the length of the longest path from a root node to the node itself. See the two levels in Fig. 1.4, indicated with roman numbers I and II. The main idea is the step–by–step construction of fragments by inserting nodes and edges in topological order. Our canonical form of a DAG contains information about the graph structure, the edge directions, the labels (if any), and the insertion order (when enumerating the search lattice and constructing growing fragments).

In Fig. 1.4(a) fragment expansion starts from a single root B (denoted by index 1). Another node B (index 2) is added in the second step. Step 3 simultaneously inserts the node A (index 3) and the edge from its predecessor B. The edge index 3.0 indicates that the edge was inserted at the same time as node 3. The last step adds an edge to the previously inserted node without adding a new node. This *first* edge targeting node 3 after its insertion is labeled 3.1. The canonical descriptions given below the fragments in Fig. 1.4 consist of tuples of the form $(\textit{node label index}, \textit{predecessor index})$. Since edge label indices are irrelevant for PA we omit them to simplify explanation. For efficiency, node labels are sorted according to their frequency (let us assume: $A = 0, B = 1, C = 2, \ldots$). Hence, the first tuple $(1,0)$ states that node B $(= 1)$ has been inserted first with no predecessor (0 as predecessor index). In general, the predecessor index refers to the insertion index of its parent node. For example, the tuple $(0,2)$ indicates that node A $(= 0)$ is inserted with node

Algorithm 1: DAG mining.

```
   Data: database with DAGs $db$, mining parameter $s$
   Result: frequent sub–DAGs $res$
1  begin
2      $res \longleftarrow \emptyset$
3      $n \longleftarrow frequentNodes(db, s)$
4      $l \longleftarrow createLabelFunction(n)$
5      while $n \neq \emptyset$ do
6          $res \longleftarrow res \cup n$
7          $tmp \longleftarrow \emptyset$
8          for $f \in n$ do
9              $tmp \longleftarrow tmp \cup \text{insertRoots}(f, l)$
10             $tmp \longleftarrow tmp \cup \text{insertLevel}(f, l)$
11             $tmp \longleftarrow tmp \cup \text{insertNode}(f, l)$
12             $tmp \longleftarrow tmp \cup \text{pruneNonCanonical}(\text{insertEdge}(f, l))$
13         $n \longleftarrow \text{filterInfrequentOrUnExtendibleFragments}(tmp, s)$
14     $res \longleftarrow \text{filterUnWantedFragments}(res, s)$
15 end
```

B_2 as its predecessor. If an edge (e.g. 3.1) is inserted without adding a new node at the same time, a special *node label index n* that is bigger than all other label indices is used. Tuple $(n, 1)$ expresses that the edge is connected to the last added node.

A *canonical fragment* is created by the insertion order of nodes and edges with the biggest canonical description. Two canonical descriptions are compared numerically, tuple by tuple and tuple-element by tuple-element. Thus, the fragment in Fig. 1.4(b) with its different edge insertion order is not canonical since $(0, 1)$ is smaller than $(0, 2)$. Hence, it can be pruned during the enumeration of the lattice. The structure of the canonical form can be used to restrict the expansion of fragments and to avoid without explicitly checking the canonical for duplicates in many cases. This will be explained in the next section.

1.4.2 Basic Structure of the DAG Mining Algorithm

The DAG mining algorithm (Algo. 1) computes an initial set of frequent single–node fragments (line 3) that are then expanded in a stepwise fashion. As a consequence of the canonical form, fragments are expanded according to the following rules:

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

Algorithm 2: Expanding Fragments with a *new Root*.

```
  Data: fragment $f$, labelIndexFunction $l$
  Result: frequent fragments $res$
1 begin
2     $res \longleftarrow \emptyset$
3     if $containsEdges(f)$ then
4         return
5     for $embedding\ x \in f$ do
6         for $unused\ node\ y \in database\ graph\ of\ x\ with\ l(y) <= l(last\ ins.\ node)$ do
7             $tmp \longleftarrow$ expand $x$ with $y$
8             add new embedding $tmp$ to $res$
9 end
```

The first three rules generate canonical fragments, so just the remaining few duplicates generated by rule 4 must be pruned (line 12). Frequency and other requirements allow more pruning (line 13) before the main loop further expends the lattice.

1.4.3 Expansion Rules

Fig. 1.2 shows a complete search lattice for a database of only one graph (shaded gray) to keep the example simple. Typically, a database contains more than one graph. Different types of edges represent the expansion rules applied. The search lattice nicely demonstrates that in this example none of the fragments (or embeddings) is visited twice, although without a pruning based on the canonical form most fragments could have been reached along several paths. After the initialization, the search lattice holds three single–node fragments.

Rule 1: Because of the topological creation, the first expansion rule (*new root*) can only be applied to fragments with one topological level and without edges (Algo. 2, lines 3–4). This rule completely avoids duplicates because no node with a label index greater than the last one will be inserted (line 7). That is similar to the *candidate item set generation* described in [1]. Consider, for example, the initial fragment *B* in Fig. 1.2. It is just extended with the root node *A* and not with *C* since the fragment (*B*,*C*) is already present as (*C*,*B*) which has a bigger canonical form $(2,0)(1,0)$ (instead of $(1,0)(2,0)$). More formally: the insertion order of roots is valid and therefore the canonical description $(x_1,0)(x_2,0)\ldots(x_i,0)$ is maximal, if the condition $x_a >= x_b$ holds for every $a < b$.

Rule 2: When a *new topological level* is started by the insertion of a node, the expansion of the current topological level is completed. All duplicates can easily be avoided during this phase by checking *partitions* that reflect the symmetries in a graph. Partitions are the basis of graph isomorphism tests [10] and can be constructed in polynomial time. Partitions are created by the indegree, outdegree,

Algorithm 3: Expanding Fragments with a *new Level* or a *new Node*.

```
Data: fragment $f$, labelIndexFunction $l$
Result: frequent fragments $res$
1 begin
2     $res \longleftarrow \emptyset$
3     for node $x$ of $f$ at the current topological level or before the current level do
4         $step \longleftarrow$ ins. step of $x$
5         if $\neg$ samePartition$(step, step+1)$ then
6             for embedding $e \in f$ do
7                 $res \longleftarrow res \cup$ expandNewLevel$(l,e,x)$ or expandNewNode$(l,e,x)$
8 end
```

Algorithm 4: Subroutine *expandNewLevel*.

```
Data: labelIndexFunction $l$, embedding $e$, node $x$
Result: frequent fragments $res$
1 begin
2     $res \longleftarrow \emptyset$
3     $superX \longleftarrow$ corresponding node to $x$ in supergraph of $e$
4     for unused edge $y$ to unused node $z \in$ supergraph of $e$ do
5         $tmp \longleftarrow$ expand $e$ with edge $y$ ($superX \rightarrow z$)
6         add new embedding $tmp$ to $res$
7 end
```

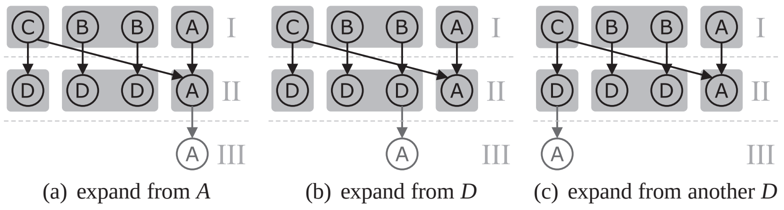


Fig. 1.5 Starting a new level III by insertion of a node and an edge.

and node label index of every node and are afterwards iteratively refined based on their neighboring partitions. Regardless of which node of a partition is selected as the predecessor, the resulting graphs are isomorphic. Therefore, a new level can only be started canonically when the last inserted node of a partition (with the highest insertion index) is used as the predecessor (Algo. 3, line 5). Since only neighboring nodes can be in the same partition, the check in line 5 is simple. The subroutine in Algo. 4 finds all unused edges of the supergraph of the current embedding with a used node as startingnode leading to an unused node. Fig. 1.5 shows all possible ways to extend a two–level graph with a new node A, since the predecessor has to be in the last topological level and the last in its partition (the gray boxes).

Algorithm 5: Subroutine *expandNewNode*.

```
Data: labelIndexFunction $l$, embedding $e$, node $x$
Result: frequent fragments $res$
begin
    $res \longleftarrow \emptyset$
    $superX \longleftarrow$ corresponding node to $x$ in supergraph of $e$
    for unused edge $y$ to unused node $z \in$ supergraph of $e$ with $l(z) \leq l(last\ ins.\ node)$ do
        $stepX \longleftarrow$ ins. step of node $x$
        $stepLast \longleftarrow$ ins. step of last predecessor
        if $\neg(l(z) = l(last\ ins.\ node) \wedge stepLast < stepX)$ then
            $tmp \longleftarrow$ expand $e$ with edge $y$ ($superX \rightarrow z$)
            add new embedding $tmp$ to $res$
end
```

Algorithm 6: Expanding Fragments with a *new Single Edge*.

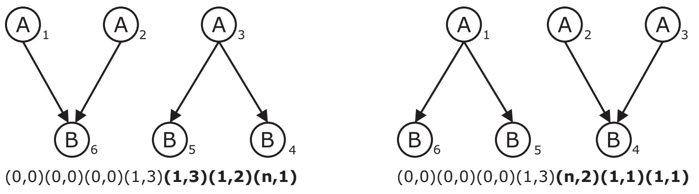
```
Data: fragment $f$, labelIndexFunction $l$
Result: frequent fragments $res$
begin
    $res \longleftarrow \emptyset$
    if samePartition(last ins. node, next to last ins. node) then
        return
    for embedding $e \in f$ do
        $superX \longleftarrow$ corresponding node to the last ins. node in supergraph of $e$
        for unused edge $y$ from used node $z$ to last ins. node $\in$ supergraph of $e$ do
            $stepZ \longleftarrow$ ins. step of node $z$
            $stepLast \longleftarrow$ ins. step of last edge–adding–node
            if $stepZ < stepLast$ then
                $tmp \longleftarrow$ expandSingleEdge$(l, stepZ, e, y, z)$
                add new embedding $tmp$ to $res$
end
```

Rule 3: The insertion of a *new node* at the current level is similar to the previous rule and does not generate duplicates, either. The partition check has to be applied again (Algo. 3, line 5). Since the node label index is the most significant element of each tuple in the canonical form, the next inserted node must have a smaller (or equal) index than its predecessor (Algo. 5, line 4). For equal labels, the new predecessor index also has to be smaller or equal to the previous predecessor index to achieve the maximal canonical description (line 7). In Fig. 1.2 this rule is used once to generate fragment $C \rightarrow (A,B)$ with its canonical description $(2,0)(1,1)(0,1)$. The same fragment could have been reached from the fragment $C \rightarrow A$ with the numerically smaller description $(2,0)(0,1)(1,1)$. Hence it is pruned.

Rule 4: Probably the most difficult expansion rule is the insertion of a *new single edge* targeting the last inserted node. As before, pruning is based on partitions

Algorithm 7: Subroutine *expandSingleEdge*.

```
Data: labelIndexFunction $l$, ins. step $stepZ$, embedding $e$, edge $y$, node $z$
Result: frequent fragment $res$
1 begin
2     $res \longleftarrow \emptyset$
3     if $\neg$ samePartition($stepZ$, $stepZ+1$)$\wedge$
4             $\neg$sameLabelAndPredecessors($l$, last ins. node, next to last ins. node) then
5         $res \longleftarrow$ expand $e$ with edge $y$ ($z \rightarrow$ last ins. node)
6 end
```


(0,0)(0,0)(0,0)(1,3)(1,3)(1,2)(n,1)
(0,0)(0,0)(0,0)(1,3)(n,2)(1,1)(1,1)

(a) duplicate fragment (b) canonical fragment

Fig. 1.6 Duplicate fragment that is not avoided by the enumeration process.

and predecessor indices (see Algo. 6, line 3 and Algo. 7, line 3). In addition, the set of predecessors of the current and the last inserted node are compared to exclude non-canonical insertion orders (Algo. 7, line 4). This approach can avoid a good portion of potential duplicates but not all of them. Complete avoidance may be possible, but has to be *NP*–complete due to the *NP*–complexity of sub–DAG mining. Hence, there is the usual trade–off: a more complex test is slower but speeds up the search process by more pruning.

Fig. 1.6(a) shows a duplicate of the canonical fragment in Fig. 1.6(b) that is not avoided by the enumeration process and needs to be pruned by an exponential test. We accelerate this test by reusing the partition information computed during expansion. Permuting the insertion order of nodes in the same partition leaves the canonical form unchanged, so only the permutations of partitions at each topological level must be checked. This does not decrease the theoretical complexity compared to permuting all nodes, but speeds up the process considerably.

1.4.4 Application to Procedural Abstraction

In general, compilers do not reach minimal code size. PA can reduce code size by extracting duplicate code segments from a program (i.e. the binary code). The instructions of an assembly program do not depend on each other line by line but they can be reordered as long as the data flow dependencies between the instructions are respected. These can be modeled as directed edges in an acyclic graph [11], called *data flow graph*. For PA we minimize the DFGs by removing edges between two nodes if there are alternative chains of dependencies between them. The resulting sparse graphs are faster to mine, but yield the same relevant fragments. Embedding–based DAGMA finds a maximal non-overlapping subset of embeddings of each basic block in the minimized DFGs. Searching for the best maximal non-overlapping subset of *all* embeddings would probably lead to even better results, but its *NP*–complete complexity is too costly for our experiments.

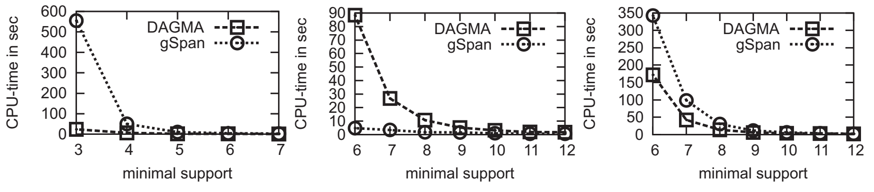
After the mining, we judge the resulting fragments and embeddings with respect to their size, number of occurrences, and type in order to get the maximal code size reduction. Depending on the fragment and extraction type, an extraction by means of a jump my be cheaper than the call–instruction shown in Fig. 1.1. With respect to the compaction profit (if positive), we extract the best fragment by clustering the nodes and edges of the embedding to a new single node and we add new instructions to the graph according to the extraction type (like *return*). Afterwards, DAGMA is applied again and searches for further frequent fragments until no more frequent subgraphs are found or the best compaction profit is below some threshold.

Unfortunately, there are embeddings of induced frequent fragments that cannot be correctly extracted by PA, because they do not respect all original dependencies after such an embedding is extracted. There is a simple way to check if an embedding cannot be extracted: replace all nodes of the embedding with a single new node and redirect edges between in– and outside of the embedding so that they are connected to the new node. If the resulting graph is cyclic, embedding extraction would break dependencies.

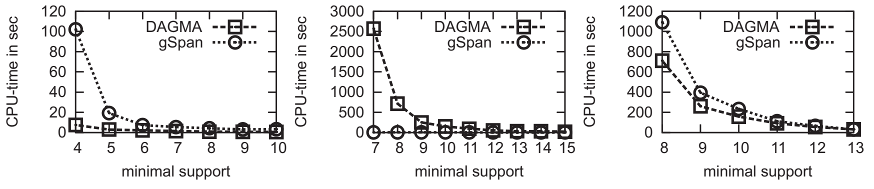
As DAGMA expands fragments level by level and node by node some additional pruning is possible. Unregarded dependencies are reflected by cycles in the clustered graph and are the result of missing edges in the embedding. Due to DAGMA's topological expansion, only single–edges towards the last inserted node (or the corresponding instruction) can be added and only cycles that contain this last node can be eliminated by further expansion steps. The expansion of the other nodes is finished at this time and cycles that contain only those finished nodes cannot be included into the embedding. Therefore, those fragments and their expansions can be pruned from the search lattice without affecting the number of instructions PA saves. In our PA experiments, we can prune over 90% of the embeddings that otherwise would be generated.

1.5 Evaluation

To evaluate DAGMA we compared it to gSpan, the most general and flexible graph miner currently available [15]. Since gSpan only addresses connected mining, we extended it with a pseudo–root node that is connected to every other node. This helper node is later removed from the resulting fragments [16]. Both algorithms are implemented in the same Java framework (using Sun JVM version 1.5.0). An AMD Opteron with 2 GHz and 11 GB of main memory has executed our comparisons



(a) graph–based mining



(b) embedding–based mining

Fig. 1.7 Runtimes for mining single–rooted (left), connected (middle) and unconnected (right) fragments(buttom) in a synthetic DAG database

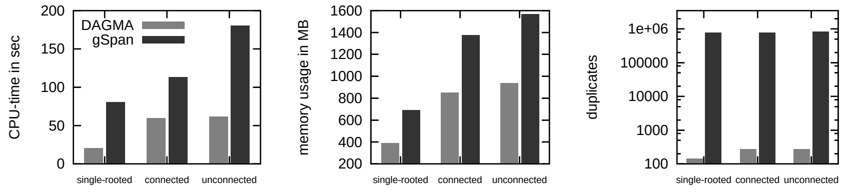


Fig. 1.8 Runtime, memory, and number of duplicates for a fully connected DAG with 7 nodes

on synthetic DAG databases, on a worst case database, and on a database from our application domain (Procedural Abstraction).

Synthetic DAGs were generated as follows to contain similarities: Every node and edge reachable from a randomly selected node in a big random master DAG is copied into the DAG database [3]. Our master DAGs contain 50 nodes, 200 edges, and 10 labels each. We restrict sub–DAGs to 5 topological levels or 25 nodes. Regardless of the random database, we always got almost the same results. Fig. 1.7 compares the runtime for graph– and embedding–based support. For both types, our approach clearly outperforms gSpan, except for connected fragments because of our preprocessing that filters out unconnected fragments. The number of fragments and embeddings is significantly higher when mining unconnected. Since an unconnected fragment can become connected during expansion, no pruning is possible and our approach has to do much unnecessary work. For a decreasing minimal support resp. an increasing number of embeddings the differences between the approaches get more prominent regardless of the fragments' shapes. A simple extension restriction leads to single–rooted mining in DAGMA: After computing the initial set of frequent nodes, no other root is added to the fragments.

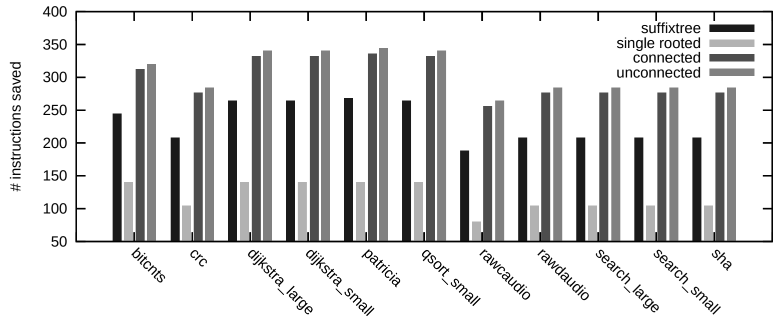


Fig. 1.9 Instruction savings for programs from MiBench

The *worst case* for DAG (and graph) miners is an equally labeled and fully connected DAG that can be created stepwise by inserting a node and connecting it to all previous nodes until the desired number of nodes is reached. Fig. 1.8 shows the results for an embedding–based search with minimal support 1 on such a maximal DAG with seven nodes. In that case 2,895,493 embeddings can be found. Again, DAGMA clearly outperforms gSpan with respect to both runtime and memory consumption in all three mining types. The main advantage and the reason for this behavior become apparent on the right of Fig. 1.8: Due to its DAG–specific canonical form, DAGMA has to handle far less duplicates in costly isomorphism tests than gSpan. The same holds for the synthetic databases.

To evaluate our algorithm for *PA*, we transformed several ARM assembly codes from the MiBench suite [9] into DFGs and mined embedding–based. Fig. 1.9 gives the savings in code size compared to the original code size when mining with suffix trees, mining for single–rooted, connected, and unconnected fragments. Mining for single–rooted DAGs is not as successful as mining with suffix trees. But when searching for connected fragments a lot more instructions can be saved. The search for unconnected fragments leads to the best results, yielding smaller assembly code and therefore higher efficacy.

1.6 Conclusion and Future Work

With DAGMA, we presented a flexible new DAG mining algorithm that is able to search for induced, unconnected or connected, multi– or single–rooted fragments in DAG databases. Since both graph– and (induced) embedding–based mining is possible (the latter is necessary for PA) and since DAGMA can mine both connected and unconnected (necessary for PA), DAGMA can be used for several application scenarios. The novel canonical form and the basic operations of the miner are based on the fact that DAGs have topological levels. The new algorithm faces significantly fewer duplicates in the search space enumeration compared to the general graph miner gSpan. This leads to faster runtime and reduced memory consumption. When applied to Procedural Abstraction, DAGMA achieves more code size reduction than traditional approaches. Procedural Abstraction traditionally searches with a minimal support of 2 embeddings to get the best possible results. For big binaries, the resulting graphs sometimes have been too large to be mined at such a small support. Hence, it seems necessary to study heuristics that guide the mining process. It will probably also require parallel DAG mining.

References

1. Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Record*, 22(2):207–216, May 1993.
2. Christian Borgelt and Michael R. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *Proc. IEEE Int'l Conf. on Data Mining (ICDM'02)*, pages 51–58, Maebashi City, Japan, December 2002.
3. Y.-L. Chen, H.-P. Kao, and M.-T. Ko. Mining DAG Patterns from DAG Databases. In *Proc. 5th Int'l Conf. on Advances in Web-Age Information Management (WAIM '04)*, volume 3129 of *LNCS*, pages 579–588, Dalian, China, July 2004. Springer.
4. Y. Chi, R. Muntz, S. Nijssen, and J. Kok. Frequent subtree mining – an overview. *Fundamenta Informaticae*, 66(1-2):161–198, 2005.
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
6. S.K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler Techniques for Code Compaction. *ACM Trans. on Programming Languages and Systems*, 22(2):378–415, March 2000.
7. A. Dreweke, M. Wörlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen. Graph-Based Procedural Abstraction. In *Proc. of the 5th Int'l Symp. on Code Generation and Optimization*, pages 259–270, San Jose, CA, USA, 2007. IEEE.
8. Scott Fortin. The Graph Isomorphism Problem. Technical Report 20, University of Alberta, Edmonton, Canada, July 1996.
9. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. Int'l Workshop on Workload Characterization (WWC '01)*, pages 3–14, Austin, TX, Dec. 2001.
10. Brendan McKay. Practical Graph Isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
11. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
12. Siegfried Nijssen and Joost N. Kok. A Quickstart in Frequent Structure Mining can make a Difference. In *Proc. Tenth ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD '04)*, pages 647–652, Seattle, WA, USA, August 2004. ACM Press.
13. Robert Endre Tarjan and Anthony E. Trojanowski. Finding a Maximum Independent Set. *SIAM Journal on Computing (SICOMP)*, 6(3):537–546, 1977.
14. A. Termier, T. Washio, T. Higuchi, Y. Tamada, S. Imoto, K. Ohara, and H. Motoda. Mining Closed Frequent DAGs from Gene Network Data with Dryade. In *20th Annual Conf. of the Japanese Society for Artificial Intelligence*, pages 1A2–3, Tokyo, Japan, June 2006.
15. M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In *Proc. Conf. on Knowledge Discovery in Database (PKDD'05)*, volume 3721 of *LNCS*, pages 392–403, Porto, Portugal, October 2005.
16. Xifeng Yan and Jiawei Han. gSpan: Graph-Based Substructure Pattern Mining. In *Proc. IEEE Int'l Conf. on Data Mining (ICDM'02)*, pages 721–724, Maebashi City, Japan, Dec. 2002.
17. David Zaretsky, Gaurav Mittal, Robert P. Dick, and Prith Banerjee. Dynamic Template Generation for Resource Sharing in Control and Data Flow Graphs. In *Proc. 19th Int'l Conf. on VLSI Design*, pages 465–468, Hyderabad, India, January 2006.