INVESTIGATING THE CANDIDATE PAIR GENERATION OF THE VF2 ALGORITHM

Péter Feher and László Lengyel
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
email: {feher.peter, lengyel}@aut.bme.hu

ABSTRACT
Recently, model transformation approaches are becoming increasingly popular; therefore their efficiency and usability are key factors and require further investigation. These approaches are regularly used in software engineering, most often as part of the model-driven development methodology. In the case of the graph rewriting-based model transformation approaches, the applied graph matching algorithm greatly determines the efficiency of the transformation approach. The matching algorithm is used to identify sub-graph isomorphism between the transformation rules and the host graph. A widely applied graph matching algorithm is the VF2 algorithm. In this paper we briefly introduce the original VF2 algorithm – specifically, its method – which manages the next possible node pairs. We present a new approach, which increases the efficiency of choosing these node pairs. We also introduce an ordering function, which increases the performance even further. Finally, the effects of these new approaches are presented based on our experimental results.

KEY WORDS
Graph matching, Graph rewriting, Graph transformation, Optimization, Pattern matching, Software algorithms, VF2 algorithm

1 Introduction
Currently, the model-driven software engineering is a popular approach in the field of software development. It is often based on model transformation, which is used throughout a wide spectrum of the development phases. A few representative objectives, including examples, are as follows: (i) to create a higher level of abstraction in the development phase through the mapping of platform-independent models to platform-specific models, e.g., to increase the reusability and maintenance of the systems and testing in smart home systems [1]. (ii) Transforming models between different domains, e.g., transforming a system into a Petri-net model to perform certain analysis. (iii) Refactoring purposes, i.e. improving model attributes and structure, while preserving the semantic meaning. Since model transformations appear in numerous scenarios, there is an obvious need for tools and techniques regarding their operational support. There are research groups concentrating on how to express model transformations and how they can be supported appropriately using different tools [9]. Other research results [5] [8] analyze the different development phases of a model transformation and deal with their specification and implementation [6]. This paper focuses on the efficiency of the model transformation process.

Graphs are commonly used to represent a variety of data structures. By attributing the nodes and edges, graphs are effectively used in modeling different states of a system, solving traffic problems, representing relationship between the different parts of the software and/or modeling the structure of chemical elements. Therefore, graphs have gained significant attention within the scientific community.

There are several algorithms to identify graph-subgraph isomorphisms, which methods are essential parts of the graph rewriting-based model transformation approaches. Since it is already proven that subgraph isomorphism is NP complete [11], the different approaches attempt to reduce the computational complexity. Therefore, some algorithms define restrictions towards the structure of the graphs, others attempt to reduce the searching space at each iteration.

Currently the VF2 algorithm [2] is a popular approach to locate complete or graph-subgraph isomorphisms. Experimental results have shown their efficiency, particularly regarding large graphs [7]. However, in the case of model transformation approaches, the performance of the pattern matching algorithms has a significant influence on the entire transformation process. Therefore, it remains essential to increase the performance of the matching.

The goal of the presented approach is to increase the efficiency of the development and quality of the software artifacts.

The rest of this paper is organized as follows: Section 2 includes the introduction of the VF2 algorithm. Section 3 provides a method to increase the performance of the VF2 algorithm. Section 4 presents a supportive ordering function. Section 5 introduces our experimental results. Finally, concluding remarks are elaborated.
2 The VF2 Algorithm

The VF2 algorithm presented in [2] and [4] is an updated version of the VF algorithm [3] from the same authors. The main difference between the two versions is the applied data structure, but the generation of the candidate node pairs—which is the scope of this paper—remains the same.

Given two graphs as $G_{Host} = (N_{Host}, E_{Host})$ and $G_{LHS} = (N_{LHS}, E_{LHS})$ as is depicted in Figure 1. During the matching process, the algorithm determines the mapping $M$, which associates nodes from $G_{Host}$ to nodes of $G_{LHS}$. Basically, $M$ is a set of $(n, m)$ pairs, where $n \in N_{Host}$ and $m \in N_{LHS}$ and each pair represents a mapping of a node $n$ from $G_{Host}$ with a node $m$ from $G_{LHS}$.

The VF2 algorithm uses the State Space Representation (SSR) [10] to describe a given state of the matching process. Each state $s$ is a partial mapping solution, which is a subset of the $M$. This partial mapping solution ($M(s)$) contains only some elements of $M$. In the SSR, the transition between two states is the addition of a new $(n, m)$ node pair.

In order to reduce the searching space, the algorithm applies some coherence conditions at each state — $s_t$, $(s_t \subset M)$ — from $s_0$ to the final, isomorphic state. For example, it is required that the partial mappings are isomorphisms between the subgraphs. If the addition of an $(n, m)$ pair to the $M(s)$ does not meet the coherence conditions, then the examination of that path is unnecessary because the path will not arrive at the final state.

The effectiveness of the VF2 algorithm is the ability to determine if the state $s$ has no successors, i.e. further states, following a number of steps that meet the coherence conditions. This examination of $s$ is done by the so-called feasibility rules. Naturally, these rules should detect if the $s$ has no coherence successors as early as possible. In doing so, the rules assist in reducing as much of the searching space as is possible. If the rule determines whether the given state $s$, with the addition of the pair $(n, m)$, can establish coherence successors in the next $k$ step then the rule implements a $k$-look-ahead.

The VF2 algorithm specifies five feasibility rules to check isomorphism, each of which regards the syntax of the graphs. Moreover, the two $0$-look-ahead rules ensure necessary and sufficient conditions for coherence. The other rules’ main function are to reduce the searching space. The feasibility rules are the following [2]:

- The $R_{Pred}$ $0$-look-ahead rule states that each predecessor $nt$ of $n$ in the partial mapping belongs to the node $mt$, which is a predecessor of $m$. The inverse of this rule is also true.
- The $R_{Succ}$ $0$-look-ahead rule states that each successor $nt$ of $n$ in the partial mapping belongs to the node $mt$, which is a successor of $m$. The inverse of this rule is also true.
- The $R_{In}$ $1$-look-ahead rule states that the number of predecessors (successors) of $n$ that are in $T^*_{Host}(s)$ is equal to the number of predecessors (successors) of $m$ that are present in $T^*_{LHS}(s)$.
- The $R_{Out}$ $1$-look-ahead rule states that the number of predecessors (successors) of $n$ that are in $T^*_{Host}(s)$ is equal to the number of predecessors (successors) of $m$ that are present in $T^*_{LHS}(s)$.
- The $R_{New}$ $2$-look-ahead rule states that the number of predecessors (successors) of $n$ that are neither in $M_{Host}(s)$ nor in $T^*_{Host}(s)$ is equal to the number of predecessors (successors) of $m$ that are neither present in $M_{LHS}(s)$ nor in $T^*_{LHS}(s)$.

During the presentation of the feasibility rules, some new sets were introduced. The $T^*_{Host}(s)$ and the $T^*_{LHS}(s)$ sets stand for the nodes that have ongoing edges to the two subgraphs in state $s$ and are not in the $M_{Host}(s)$, $M_{LHS}(s)$ respectively. Similarly, $T^*_{Host}(s)$ and the $T^*_{LHS}(s)$ sets contain nodes that have incoming edges from the subgraphs in the given state and are not a part of the partial mappings. Moreover, there are $T_{Host}(s) = T^*_{Host}(s) \cup T^{in}_{Host}(s)$ and $T_{LHS}(s) = T^*_{LHS}(s) \cup T^{in}_{LHS}(s)$. These sets are depicted in Figure 1 as well.

In the case of graph-subgraph isomorphism, these original feasibility rules need to be slightly altered. Since the algorithm is looking for subgraph isomorphism, the $R_{Pred}$ and $R_{Succ}$ rules cannot check whether all predecessor/successor $nt$ of $n \in G_{Host}$, in the partial mapping, belongs to a node from the LHS graph. This is due to the lack of certainty that the edge between the $n$ and $nt$ is part of the subgraph as well. However, the other part of the rule has to be checked, i.e. all predecessor/successor $nt$ of $m \in G_{LHS}$ in the partial mapping has to belong to the node $nt$, which is a predecessor/successor of $n$. This is upheld because every edge in the LHS graph has to appear in the host graph.

The only difference in the $R_{In}$ and $R_{Out}$ rules, regarding subgraph matching, is the change of the equals (=) sign to a greater than (>) sign. The reason for this change is because a node in the host graph can have more predecessors/successors than its matched pair in the LHS graph.
The final feasibility rule is not applicable in case of graph-subgraph isomorphism and necessitates a greater change; therefore it is not applied in the algorithms discussed in this paper.

At each state the VF2 algorithm generates a set of candidate node pairs, denoted as $P(s)$. These node pairs are checking if the feasibility rules are met and can be added to the partial mapping ($M(s)$). Following each addition, the VF2 algorithm generates the candidate node pairs again and checks them against the rules. If there is no candidate pair that meets the feasibility rules then the algorithm simply backtracks and drops that candidate node pair.

The algorithm that constructs the $P(s)$ is shown in Algorithm 1.

**Algorithm 1 Generation of candidate pairs in the VF2 algorithm**

1. `procedure GENERATECANDIDATEPAIRS()`
2. if $T_{Host}^{out}(s) \neq \emptyset$ and $T_{LHS}^{out}(s) \neq \emptyset$ then
3. $P(s) \leftarrow T_{Host}^{out}(s) \times \{\min(T_{Host}^{out}(s))\}$
4. `else if $T_{Host}^{in}(s) \neq \emptyset$ and $T_{LHS}^{in}(s) \neq \emptyset$ then`
5. $P(s) \leftarrow T_{Host}^{in}(s) \times \{\min(T_{LHS}^{in}(s))\}$
6. `else`
7. $P(s) \leftarrow \{N_{Host}^{in} - M_{Host}^{in}(s)\} \times \{\min(N_{LHS}^{in} - M_{LHS}^{in}(s))\}$
8. `return $P(s)$`

In this context, $\min$ refers to the node in $T_{LHS}^{out}(s)$, which contains the smallest label (actually, any other total ordering criterion could be applied) [4].

It is important to note, in order to ascertain there is no match between the host and the LHS graph, the VF2 has to check all candidate node pairs. This means that the efficiency of the algorithm can be increased if the cardinality of the checked candidate node pairs is reduced.

We would also like to mention that the rest of the paper uses undirected graphs. However, this has no effect on the operation of VF2 algorithm.

### 3 Candidate Pair Number Reduction

The original construction of the candidate node pairs does not consider the success of the previous candidates, nor the last added nodes. It simply creates the node pairs and selects one to check against the feasibility rules. Assume the following partial mapping: $M(s) = \{(A, X), (B, Y), (C, Z)\}$ as it is depicted in Figure 2. Supposing the last added pair was $(C, Z)$. At this point the $T_{Host}$, $T_{LHS}$ sets and the candidate node pairs are the following:

- $T_{Host} = \{D, E, F, G, H\}$
- $T_{LHS} = \{Q, T\}$
- $P(s) = \{(D, Q), (D, T), (E, Q), (E, T), (F, Q), (F, T), (G, Q), (G, T), (H, Q), (H, T)\}$

Note that the $Z$ node has only one successor which is not part of the partial mapping, the $Q$ node. This means that, if $Q$ cannot be mapped, this path cannot lead to a complete match, regardless of whether the $T$ node can be mapped or not. Moreover, there is no need to check the candidate node pairs that contain $T$ if $Q$ cannot be matched. This way the number of checks can be significantly reduced, depending on the graph structures.

Based on the example, it may be worthwhile to check the node pairs that contain the $Q$ node. In this specific example, the $(E, Q), (F, Q)$ and $(G, Q)$ candidate pairs do not satisfy the feasibility rules, whereas both the $(D, Q)$ and $(H, Q)$ pairs meet the feasibility rules and can be added to the mappings. However, the algorithm then backtracks in both cases, since the last node $(T)$ cannot be added without breaking the first two feasibility rules. This means that the algorithm returns to the mapping defined above but the $P(s)$ does not contain any node pair containing the $Q$ node.
Thus, the \( P(s) \) does not contain any node from the LHS graph that is a successor of the \( Z \) node. This means that the algorithm cannot match the \( Q \) node to any node in the host graph in this partial mapping, therefore further examination of the other candidate pairs is unnecessary and the algorithm can backtrack. There is no need to check the other five candidate node pairs because they cannot lead to a final state. This way, the performance of the VF2 algorithm can be increased since fewer number of node pair checks are required.

**Algorithm 2** Extension for the procedure that generates candidate node pairs

**Require:** \( m \in M^{LHS} \)

1. **procedure** \( \text{GENERATECANDIDATEPAIRSEXTEDENDED} \) (Node \( m \))
2. \( P(s) \leftarrow \text{GENERATECANDIDATEPAIRS}() \)
3. SuccList \( \leftarrow \emptyset \)
4. PredList \( \leftarrow \emptyset \)
5. SuccList \( \leftarrow \forall n \in \text{succ}(G^{LHS}, m), n \notin M^{LHS} \)
6. if \( \# \{\text{SuccList}\} > 0 \) then
7. return \( \text{GETPAIRSFROMLISTS}(\text{SuccList}, P(s)) \)
9. PredList \( \leftarrow \forall n \in \text{pred}(G^{LHS}, m), n \notin M^{LHS} \)
10. if \( \# \{\text{PredList}\} > 0 \) then
12. return \( \text{GETPAIRSFROMLISTS}(\text{PredList}, P(s)) \)
13: return \( P(s) \)

The algorithm described above is shown in Algorithm 2. It first generates possible candidate node pairs as it is created in the original VF2 algorithm; then creates the SuccList list. This contains all nodes from the LHS graph that are successors of the given \( m \) node and not yet part of the partial mapping. The procedure gets the \( m \) node as a parameter, which is the last added LHS node. If this SuccList variable is not an empty set, then GETPAIRSFROMLISTS is used with the SuccList as well as the list containing the original candidate node pairs as parameters. Finally, it returns with the result of the method.

If the SuccList is empty, then the PredList is created. This list contains the predecessors of \( m \) that are not part of the mapping. If PredList is not empty, then the GETPAIRSFROMLISTS is used but this time the PredList is passed and its result is returned. If both lists are empty, then the algorithm returns with the original list of candidate pairs and it cannot improve the efficiency. The GETPAIRSFROMLISTS is shown in Algorithm 3.

The GETPAIRSFROMLISTS retrieves two parameters. The first is a list of nodes that contains either the successors or the predecessors of a LHS node that are not part of the current mapping. The second parameter contains the candidate node pairs. The algorithm simply checks if there exists a node in the successors/predecessors which is not part of any pairs of the candidate list. If there is such a node then it is not worth examining this partial mapping further because there will exist a node in the LHS graph that cannot be mapped. Thus, the procedure returns with an empty set. Otherwise, the algorithm creates a list that contains only the node pairs which have the LHS part from the given successors or predecessors. When this list is created, the procedure returns with it.

The modified generation algorithm, as it is presented in Section 5, improves the performance of the VF2 algorithm. Table 1 demonstrates that in this way the number of required feasible checks is reduced significantly.

### 4 Ordering the Candidate Pairs

In Section 2 it was mentioned that the VF2 algorithm uses an ordering function during the generation of the candidate node pairs. Therefore, this ordering function also affects the efficiency of the algorithm.

In the previous Section, a modified algorithm was proposed in order to increase the performance of the VF2 algorithm. The main objective of this approach is to ensure that the algorithm backtracks immediately when there is a successor/predecessor of the last added LHS node, which is not a part of any candidate node pair. This means that the algorithm should systematically select the next candidate pair to achieve this state as soon as possible.

**Algorithm 3** Generate the candidate node pairs

1: **procedure** \( \text{GETPAIRSFROMLISTS} \) (NodeList Neighbors, NodePairList Candidates)
2: \( Q \leftarrow \emptyset \)
3: if \( \exists n \in \text{Neighbors}, (l, n) \notin \text{Candidates} \) then
4: return \( \emptyset \)
5: else
6: for all \( (l, n) \in \text{Candidates} \) do
7: if \( n \in \text{Neighbors} \) then
8: add \( (l, n) \) to \( Q \)
9: return \( Q \)

When a new node pair is added to the partial map-
ping, the cardinality of each LHS node in the candidate node pairs is equal. Figure 2 shows that two LHS nodes are part of the candidate node pairs: the $Q$ node and the $T$ node. Both nodes contain five candidate pairs. At this moment, any of the pairs could be selected. For example, assume that the $(F, Q)$ node pair is selected. As was discussed earlier, this pair fails the first feasibility rule and, therefore, the algorithm drops it. In the next step there are only nine different candidate node pairs. Five of them contains the $T$ node, but only four the $Q$ node. Since the algorithm backtracks at the moment an LHS node is not contained by any candidate node pair, it should attempt to reach such a state. In this particular example, this signifies that the algorithm will select from the node pairs containing the $Q$ node. Therefore, it is guaranteed to find, in four steps, a new feasible pair or it will backtrack. Considering this, the GETPAIRSFROMLISTS is modified as it is shown in Algorithm 4.

The second half of this algorithm differs from the previous one in the following ways. If all LHS nodes appear in the candidate pairs, then the algorithm select the node which has the fewest occurrences in the set of candidate pairs. This is done by the GETMINOCCURRENCE method. Finally, it returns all pairs, which contain this particular node.

5 Performance Analysis

In the previous two sections, we presented a modified algorithm to produce candidate node pairs and an ordering function, which is well adjusted to this modified algorithm. To examine the effects of the new approaches, experimental results were measured.

In the examples below, the VF2 attempts to identify a match between three host graphs and five LHS graphs. In each case, the algorithm uses three different approaches:

- The first approach uses the default algorithm to generate candidate pairs without any ordering function.

- The second approach uses the modified algorithm, described in Section 3, and backtracks when a node from the LHS graph does not appear in the candidate pairs. It does not use any specific ordering function.

- The third approach combines the modified approach with the ordering function presented in Section 4.

Some highly symmetric graphs have been chosen as host graphs to ensure the possibility of a frequent occurrence of backtracking. The host graphs are the followings:

- The Petersen graph: Figure 3a.

- A vertex-transitive graph: Figure 3b.

- A t-transitive graph: Figure 3c.

Figure 3: The example host graphs
Table 1: Comparison of the algorithms

<table>
<thead>
<tr>
<th></th>
<th>Petersen</th>
<th>Vertex-transitive</th>
<th>T-transitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Modified</td>
<td>Ordered</td>
</tr>
<tr>
<td>1st LHS</td>
<td>12830</td>
<td>7465</td>
<td>6320</td>
</tr>
<tr>
<td>2nd LHS</td>
<td>210</td>
<td>180</td>
<td>120</td>
</tr>
<tr>
<td>3rd LHS</td>
<td>39360</td>
<td>4970</td>
<td>6180</td>
</tr>
<tr>
<td>4th LHS</td>
<td>34850</td>
<td>27620</td>
<td>10280</td>
</tr>
<tr>
<td>5th LHS</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

The LHS graphs are shown in Figure 4. Only the fifth LHS graph can be found in each host graph, and the second one is a subgraph of the vertex-transitive host graph.

Table 1 presents how often the feasibility rules are checked in the different cases: The Original columns regard the original VF2 approach, the Modified columns to the algorithm including the modified candidate pair generation and the Ordered columns refer to the modified algorithm with the presented ordering function.

The results indicate that the modified algorithm performs more effectively, in every case, when the LHS graph cannot be found in the host graph. Provided the ordering function, even better results are possible. Although, as is the case in the third LHS graph – the Petersen graph – it does not necessarily perform better. However, the Ordered version can perform as well as 6 times more effective than the Original.

Based on the results, the efficiency of the algorithm depends upon the graphs. The following aspects have an impact on its improvement:

- The number of neighbors the lastly added LHS node has that are not part of the partial mapping. (The more nodes can result in more significant decrease.)
- The cardinality of the $T_{Host}$ set. (More nodes result in more significant decrease.)
- The selection of the next LHS node to map.

**Proposition 1.** The `GenerateCandidatePairsExtended` algorithm cannot conduct more checks than the default approach (`GenerateCandidatePairs`).

**Proof.** The `GenerateCandidatePairsExtended` reduces the number of checks through the implementation of backtracking, when an LHS node is absent from every candidate pairs. In these cases there is no need to check the remaining nodes, because that particular LHS node cannot be mapped. However, in the worst case scenario, the algorithm cannot reduce the number of checks and needs to examine all candidate pairs in the exact same way, as is the case in the `GenerateCandidatePairs` approach. Since the original approach checks all pairs, the `GenerateCandidatePairsExtended` cannot exceed this value. 

\[\square\]
Proposition 2. If the primarily selected LHS node, in the algorithm $\text{GENERATECANDIDATEPAIRSEXTENDED}$, cannot be matched, the VF2 with the $\text{GETPAIRSFROMLISTS}$ ordering function backtracks as soon as possible.

Proof. The $\text{GETPAIRSFROMLISTS}$ orders the candidate pairs based on the number of their LHS nodes occurrences. Following the first check with this ordering function, the candidate pairs with the given LHS node are examined, since the pairs containing this node are atop the ordered list. Therefore, the algorithm initially checks only the pairs which contain the LHS node. Since this LHS node cannot be mapped, the algorithm backtracks after all pair containing this node is observed. This cannot be achieved earlier, since the algorithm must first check these pairs. Better result cannot be achieved using another LHS node because, as a result of the candidate pair generation, all LHS nodes are contained by the same number of candidate pairs.

6 Conclusion

The focus of this paper began with a popular graph matching approach, the VF2 algorithm. The method, which is used to generate the set of candidate node pairs, has been discussed in details.

In Section 3, we presented optimization criteria to generate the candidate node pairs. This algorithm backtracks at the moment an LHS node does not appear in any candidate pairs and, in doing so, increase the efficiency of the algorithm. The method considers the last added LHS node and selects a new node from its neighbors if possible.

Next, an ordering function has been presented to support this algorithm. Using this method, the algorithm selects the next candidate pair with the LHS node, which occurs in the fewest pairs. In this way it assists the algorithm in reaching a state from it can backtrack.

Finally, some experimental results have been presented. It has been shown, that the application of these methods can greatly increase the performance of the VF2 algorithm. Although the host graphs contained only 24, maximum, nodes and the LHS graphs 6, the difference between the original and the modified algorithm is strikingly evident.

Currently, our focus is to present approaches to maximize their efficiency regarding every graph family.

Acknowledgments

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TÁMOP-4.2.2.C-11/1/KONV-2012-0013).

References


