A Unified Framework for Strengthening Topological Node Features and Its Application to Subgraph Isomorphism Detection

Nicholas Dahm[1,3], Horst Bunke[4], Terry Caelli[2,5], and Yongsheng Gao[3]

[1] Queensland Research Laboratory, National ICT Australia
[2] Victoria Research Laboratory, National ICT Australia
terry.caelli@nicta.com.au
[3] School of Engineering, Griffith University, Brisbane, Australia
{n.dahm,yongsheng.gao}@griffith.edu.au
[4] Institute of Computer Science and Applied Mathematics, University of Bern, Switzerland
bunke@iam.unibe.ch
[5] Electrical and Electronic Engineering, University of Melbourne, Australia
tcaelli@unimelb.edu.au

Abstract. This paper presents techniques to address the complexity problem of subgraph isomorphism detection on large graphs. To overcome the inherently high computational complexity, the problem is simplified through the calculation and strengthening of topological node features. These features can be utilised, in principle, by any subgraph isomorphism algorithm. The design and capabilities of the proposed unified strengthening framework are discussed in detail. Additionally, the concept of an n-neighbourhood is introduced, which facilitates the development of novel features and provides an additional platform for feature strengthening. Through experiments performed with state-of-the-art subgraph isomorphism algorithms, the theoretical and practical advantages of using these techniques become evident.

Keywords: Graph Matching, Subgraph Isomorphism, Topological Node Features.

1 Introduction

Identifying subgraph isomorphisms between a pair of graphs is a key problem in structural pattern recognition with an exponential worst case complexity. Subgraph isomorphism algorithms can be either exact, or inexact, with the latter dealing with incomplete or noisy graphs. In this paper we focus solely on exact subgraph isomorphism, where the graphs are complete and error-free. The concepts presented in this paper however, may apply equally to inexact algorithms. Applications for exact subgraph isomorphism include chemical substructure and protein-protein interaction network matching, social network analysis and VLSI design [1]. The most widely used algorithms for subgraph isomorphism are Ullmann's algorithm [11] and the VF2 algorithm [2]. These tree-search algorithms are able to obtain practical runtime speeds by pruning branches from the search tree that contain incompatible node matchings. However due to the exponentiality of subgraph isomorphism, as the number of nodes in the graphs increases, the matching times of both algorithms can quickly become infeasible [4].

To identify more node incompatibilities, and hence further reduce matching times, local topological information about a node can be encoded into a *topological node feature* (TNF). VF2 for example, uses the simplest TNF, namely the vertex degree (number of adjacent nodes) to identify incompatible node matchings. Topological node features like this are also known as *subgraph isomorphism consistents* or, in the case of graph isomorphism, *invariants*. On the simpler problem of graph isomorphism, a number of algorithms exist that utilise complex structural information. An early example of this is the Nauty algorithm by McKay [6], which uses TNFs and a strengthening procedure similar to the tree-index method discussed in this paper. Using this structural information, Nauty is able to identify nodes which have identical topological structure, so becoming acceptable isomorphic mappings. Recently, this idea was extended by Sorlin & Solnon to create the IDL algorithm [10]. In another recent paper Riesen et al. [5] ignores the search tree entirely and uses only TNFs to determine isomorphisms. Their method has a polynomial runtime, but in some cases cannot resolve the matching due to outstanding permutations.

When extending these concepts to subgraph isomorphism, any pair of matched nodes is less likely to have identical topological structure. Despite this, it is still possible to exploit the fact that a node in the full graph will contain the same topological structure as a node from the subgraph, however with some extra structure possibly added. From this observation, a rule can be defined for each TNF as to when a node mapping is considered invalid. For example, if a subgraph node has a degree of 5, a mapping to any full graph node with a degree less than 5 is invalid. A recent algorithm utilising TNFs is the ILF algorithm by Zampelli et al. [12]. This algorithm uses TNF values strengthened through a similar procedure to Nauty to eliminate incompatibilities. In the paper, the authors show that ILF can outperform VF2 in many cases even while only using a simple TNF like degree. The recent LAD algorithm by Solnon [9] uses a local *all different* constraint which ensures that for each mapping, the nodes adjacent to the subgraph node can be uniquely mapped to nodes adjacent to the full graph node. When combined with the generalised arc consistency (GAC) *all different* constraint that is commonly used in constraint programming, LAD has been shown to be even faster than ILF, on most cases.

In this paper we present a number of techniques that can be used to simplify the subgraph isomorphism process. Similar to [4], the techniques described here are not designed to challenge existing methods. On the contrary, they are designed so that they can be utilised as an enhancement to any subgraph isomorphism algorithm. In Section 2, we describe the concept of a node's n-neighbourhood and propose some novel topological node features which utilise it. Section 3 presents a unified framework consisting of three strengthening techniques. These strengthening techniques, introduced in Sections 3.1 to 3.3, can be applied to both topological node features as well as application-specific node labels. Section 3.4 then shows how these concepts can be combined to create strengthened features that are resistant to noise. All of these techniques can be calculated independently on each graph, allowing them to be computed ahead of time and stored. This makes matching against a large database of graphs particularly effective. Finally in Section 4, we empirically show the performance of these techniques to determine when they are best applied.

2 Topological n-Neighbourhood Features

The graphs dealt with in this paper are simple (no self loops, no duplicate edges) unlabelled graphs, both directed and undirected. A graph is defined as an ordered pair $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is a set of vertices and $E = \{\{v_x, v_y\}, \ldots\}$ is a set of edges.

A topological node feature is defined as any feature that is calculated solely on topological information, as viewed from a particular node. Some traditional TNFs used in graph matching are:

- degree. The number of adjacent nodes.
- clusterc (clustering coefficient). The number of edges between adjacent nodes (not including edges to the node being evaluated).
- ncliques$_k$. The number of cliques of size k that include a particular node.
- nwalksp$_k$. The number of walks of length k that pass through a particular node.

Both ncliques$_k$ and nwalksp$_k$ are vectors, holding values for each different k.

An n-neighbourhood (nN) of a node v is an induced subgraph formed from all the nodes that can be reached within n steps from v. This induced subgraph is centered around node v and contains all nodes up to n steps away, and all edges between those nodes. It is denoted as $n\text{N}(v, n)$. For any single node v, a unique nN may be created for each value $n = 1, 2, \ldots, m$, where $n\text{N}(v, m) = G$ (the entire graph can be reached in m steps).

There are a number of TNFs that can be calculated from each nN of a node. Firstly we have the node count, or number of nodes in the nN, denoted by nN-ncount. Likewise we have the edge count, denoted by nN-ecount. Next we have the number of walks of length k in the nN, denoted nN-nwalks$_k$. Lastly we have the number of walks of length k in the nN, that pass through the main node, denoted nN-nwalksp$_k$. Each of these TNFs will give a different result for each nN of a node, giving n values, or $n * p$ values for nN-nwalks$_k$ and nN-nwalksp$_k$ where $k = 1, 2, \ldots, p$. The primary benefit of calculating TNFs on nNs is the reduced likelihood of *noise* (topological structure not present in the subgraph) from distant nodes being encoded in the feature. For small values of n, the features contain less information but also less noise. On larger values of n, the amount of information encoded is higher, but so is the likelihood that noise will prevent the feature from detecting mismatches.

3 Node Label Strengthening Framework

Our *strengthening framework* consists of the summation, listing, and tree indices, SI, LI and TI, respectively. In this order, there is a natural progression from one index to the next as they provide more resolution but take longer to compare. Each of these indices can be applied iteratively and works by incorporating the indices of neighbouring nodes. One or all of these indices may be applied for each different value created by a TNF or even an application-specific label (listing and tree indices only). A detailed description of these indices is given in the following subsections. Table 1 in Section 3.3 compares the strengths and weaknesses of each index.

3.1 Summation Index

The *Morgan index* [7] is an effective TNF, originally used to characterise chemical structures, and more recently to assist in graph isomorphism detection [5]. Despite its success in graph isomorphism, it has limited effectiveness on subgraph isomorphism.

Derived from the calculation procedure of the Morgan index, we propose the *summation index* (SI). The summation index propagates TNFs through the graph, allowing nodes to encode neighbouring structural information into their own *strengthened* TNFs. An example is shown in Figure 1. In this example, we show how SI can strengthen the node degree. Initially (iteration 0), the SI values of nodes $A - E$ are their TNF (degree in this case) values: 1, 2, 3, 2, and 2. For all subsequent iterations, the SI values are the sum of the neighbouring SI values from the last iteration. After iteration 1, these are 2, 4, 6, 5, and 5. After iteration 2, these are 4, 8, 14, 11, and 11. This process continues for a user-defined number of iterations, or as required by a particular matching algorithm. Note that on iteration 0, nodes B, D, and E all had an equal degree, and hence SI, but after iteration 1, B could be separated from the others.

Definition 1 (Summation Index (SI)).

$$SI_i(v) = \begin{cases} \mathit{feature}(v) & \mathit{if}\ i = 0, \\ \sum_u SI_{i-1}(u) & \mathit{otherwise.} \end{cases}$$

where u is a vertex adjacent to v.

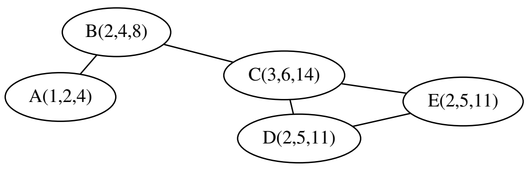

Fig. 1. Calculating the summation indices. Here we see a simple graph with three iterations of (degree) SI values shown for each node (iterations 0, 1, and 2).

As summation requires features to be added (+) and ranked ($\leq$), it cannot be used for many application-specific labels. To strengthen such labels, the listing or tree indices below can be used.

3.2 Listing Index

The second feature strengthening technique in our framework is the *listing index* (LI). The listing index is a natural progression from summation, containing more information but also requiring more complex comparisons. Fankhauser et al. [5] presented this technique for graph isomorphism under the name *neighbourhood information*. A node's neighbourhood information is essentially a list (formally a multiset) of all feature values of the neighbouring nodes.

The key difference with LI is that features are evaluated separately. This provides more resolution, at the cost of increased comparison time. As with summation, this process can be repeated to include more information. The listing index of a node, at iteration i, is equal to the union of the listing indices of all neighbouring nodes at $i-1$. For Figure 1, iteration 0 would be $\{1\}$, $\{2\}$, $\{3\}$, $\{2\}$, and $\{2\}$. This would then become $\{2\}$, $\{1,3\}$, $\{2,2,2\}$, $\{2,3\}$, and $\{2,3\}$ for iteration 1, and $\{1,3\}$, $\{2,2,2,2\}$, $\{1,2,2,3,3,3\}$, $\{2,2,2,2,3\}$, and $\{2,2,2,2,3\}$ for iteration 2.

The listing index also follows the same convention as summation, in that on each iteration, only the previous values of the neighbours are considered, with no regard to the node's own previous value.

Definition 2 (Listing Index (LI)).

$$LI_i(v) = \begin{cases} \{\mathit{feature}(v)\} & \text{if } i = 0, \\ \cup_u LI_{i-1}(u) & \text{otherwise.} \end{cases}$$

One advantage of listing over summation is that there is no requirement for the feature values to be numerical. The only requirement is that they can be compared for equality (=), unless they are TNFs being used on subgraph isomorphism, in which case they must be able to be ranked ($\leq$).

3.3 Tree Index

The final feature strengthening technique we present is the *tree index* (TI). We show this technique in its pure form as a natural progression from the other indices, and discuss some alternative versions. This technique can be thought of as a second interpretation of the listing technique. The initial step is identical to listing, however the iterations are performed differently. In our listing technique, each iteration takes the union of neighbouring lists from the last iteration. Instead of taking the union of neighbouring lists from the last iteration, we simply create a list with those lists as elements. This creates an iteratively deeper list which can be thought of as a tree, beginning from the node and branching i layers, where i is the iteration number. Using the tree index, the second iteration indices for nodes $A-E$ would be $\{\{1,3\}\}$, $\{\{2\},\{2,2,2\}\}$, $\{\{1,3\},\{2,3\},\{2,3\}\}$, $\{\{2,2,2\},\{2,3\}\}$, and $\{\{2,2,2\},\{2,3\}\}$ respectively.

Definition 3 (Tree Index (TI)).

$$TI_i(v) = \begin{cases} \mathit{feature}(v) & \mathit{if}\ i = 0, \\ \cup_u \{TI_{i-1}(u)\} & \mathit{otherwise.} \end{cases}$$

This provides us with a rich description of the node's local structure, resulting in more complex comparison challenges. In the worst case, where feature values are not ranked, this leaves us with a tree of linear assignments for each node comparison. Since the tree index effectively creates a tree of the graph starting from a node, there is no need to store the resulting values. Instead, we can simply traverse the graph during the matching.

Alternative versions of this have been presented for graph isomorphism [6,10] and subgraph isomorphism [12]. These alternative versions precompute the values and use a renaming step in an attempt to limit the size that must be stored. The effectiveness of this renaming step depends on the type of graph and can vary greatly.

Table 1. A naive comparison of the strengths and weaknesses of each index

	SI	LI	TI
Preprocessing Time	Very Low	Moderate	Zero
Preprocessing Space	Very Low	High	Zero
Matching Time	Very Low	Low	Very High
Pruning Effectiveness	Moderate	High	Very High

3.4 Strengthening in n-Neighbourhood

As mentioned in Section 2, TNFs calculated on an nN can be thought of as less noisy than their counterparts obtained on the main graph. This same concept applies equally (if not more) to the indices introduced in Sections 3.1 to 3.3. Instead of propagating the indices through the original graph, we can propagate them through the nN of each node. Although this means that each node's strengthened TNF values are calculated on a unique nN graph, these values are still valid to compare in subgraph isomorphism. The benefit of propagating through nNs is that it allows us to construct a very distinct picture of the local structure without being distorted by structural information many steps from the node. The downside to this is that structure many steps away is ignored completely, regardless of how useful such information could have been. Since nN propagation requires propagating information through each nN separately, the computation required is far more than propagation on the main graph, however the storage cost is not significantly higher. This makes nN propagation ideal for databases where graphs can be preprocessed once and matched many times.

4 Experimental Results

To evaluate the effectiveness of the techniques discussed in this paper, we first perform some analytical tests, followed by practical tests using a state-of-the-art subgraph isomorphism algorithm. For our testing data, we use positive subgraph isomorphism instances created using the geometric random graph generator from the igraph library [3]. This generator was chosen as it generates edges using a geometric method, resulting in graphs likely to be found in computer vision applications. Each test instance contains a 100 node full graph and a 90 node subgraph.

4.1 Evaluating Pruning Techniques by ABF

Subgraph isomorphism detection is most commonly performed using a search tree and some pruning techniques. Given a full graph with N nodes and a subgraph with M nodes, we have $\frac{N!}{(N-M)!}$ permutations in the search tree. The depth of the search tree is determined by the number of nodes in the subgraph. This value is static as solutions are found only at the leaves. The number of branches at each search tree node is the number of valid mappings possible for a particular subgraph node. This value, called the node's branching factor, starts as N, but may be pruned to be significantly lower.

To compare the pruning effectiveness of the techniques discussed in this paper, we use the average branching factor (ABF). We define the ABF as the average of the branching factors of subgraph nodes. To ensure the order of nodes does not skew the results, we calculate each node's branching factor as if it were at the root of the search tree. More information regarding ABF can be found in Section 3.5 of [8].

4.2 Analytical Experiments

We evaluate each of the techniques shown in this paper using the average branching factor (ABF) defined in Section 4.1, averaged over 10 test instances. For comparison with our techniques in the following figures, we provide the ABFs of the traditional TNFs that were described in Section 2. These ABF values are: 39.5 (degree), 39.7 (clusterc), 57.0 ($\text{ncliques}_k, k = 3, 4, \ldots, 8$), 48.1 ($\text{nwalksp}_k, k = 1, 2, \ldots, 8$), and 29.5 (the combination of all four). These ABF values are based solely on the traditional TNFs, without the use of any nN or index-related strengthening. As such, they can be compared *as is* with every value reported in the following figures.

In Figure 2 we compare the results from the different n-neighbourhood (nN) features defined in Section 2. For each feature, we show the ABF and comparison time for nNs up to a depth of 10 steps. Note that for a maximum nN depth of x in the figure, nNs are calculated and compared for $n = 1, 2, \ldots, x$. The comparison time here is the average time required to determine all compatibilities between the subgraph and full graph nodes on each test instance.

Comparing these figures to the traditional TNFs listed above, we can clearly see the pruning effectiveness of these nN features. At a maximum nN depth of just two steps, each nN feature has achieved more pruning than all four of the traditional TNFs. While not performing as well as some others, the nN-nwalksp feature here is particularly noteworthy, as it is the only feature that has a non-nN

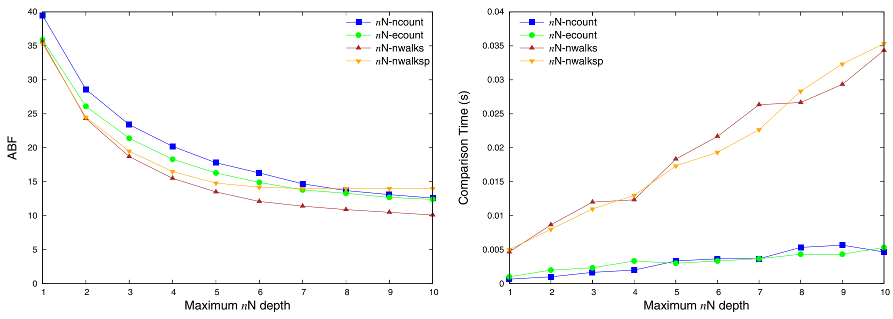

Fig. 2. ABF and comparison times for n-neighbourhood features

counterpart. A comparison between the non-nN nwalksp (ABF: 48.1) and nN-nwalksp (ABF: 35.3 $\rightarrow$ 14.0) shows the effectiveness of nNs. This effectiveness comes from the noise-reduction inherent in nNs, as discussed in Section 3.4.

Figure 3 compares the results for each of the strengthening indices detailed in Sections 3.1 to 3.3, and their nN-strengthened counterparts from Section 3.4. These figures show how the ABF and comparison times change as the maximum iteration number of the indices increases. The TNF strengthened here is degree, as this simple TNF will best show the capabilities of the strengthening indices. Note that due to the significant time required for tree-index comparisons, all times are shown on a logarithmic scale.

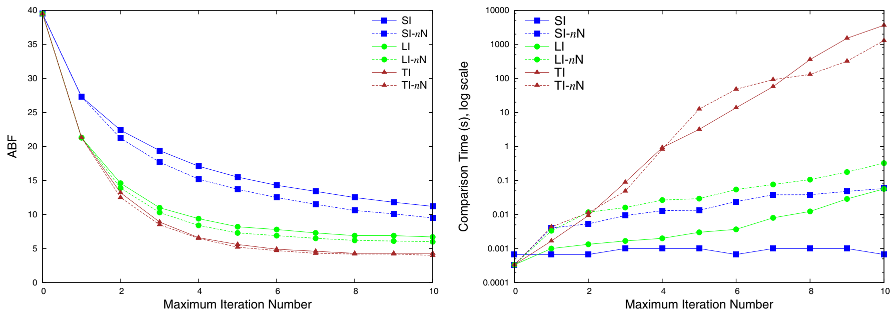

Fig. 3. ABF and comparison times for SI/LI/TI and their nN-strengthened counterparts, using only the degree TNF

Starting from the standard degree ABF of 39.5, it takes only a single iteration before all indices are below the 29.5 ABF of the combined traditional TNFs. After four iterations, both LI and TI have achieved greater pruning with just degree than any single TNF, including nN TNFs. Given a more complex feature, or combination of features, these strengthening indices can achieve even greater pruning. Of course as we include more features and strengthening techniques, the comparison time may also increase.

4.3 Practical Experiments

The ABF reductions in the previous section show that our techniques can simplify the matching problem, allowing matching algorithms to perform faster. In this section, we determine whether this reduction in matching time is worth the increase in feature comparison time. To achieve this, we perform subgraph isomorphism detection (searching for all solutions) using the VF2 algorithm, as this is the most commonly used benchmark.

Each full graph in our test set contains exactly 100 nodes. However in order to show how the number of edges affects the performance of certain techniques, we run our tests five times, each time with an increasing number of edges. For each algorithm and number of edges, we report the average matching time for 100 test instances. Figure 4 shows subgraph isomorphism detection times for VF2 when combined with the techniques from this paper.

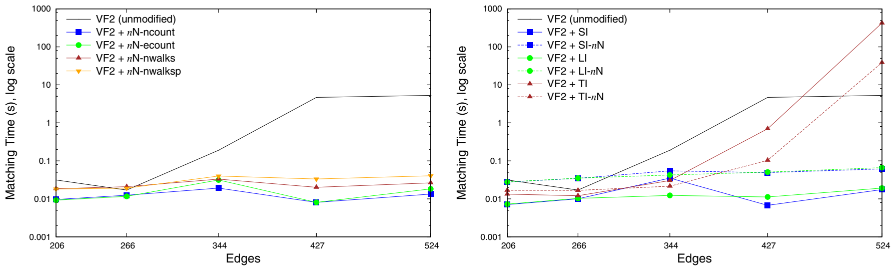

Fig. 4. Subgraph Isomorphism times for VF2 using the nN TNFs and strengthening techniques (again only strengthening degree) from this paper. Regular VF2 is included for comparison.

Due to the exponentiality of the graph matching problem, VF2 often takes significantly longer on certain instances, giving its results a high standard deviation. In addition to reducing the average matching times as shown in Figure 4, our techniques also help VF2 overcome these hard instances. This results in a more consistent matching time, which is advantageous in real-time applications.

5 Conclusions

In this paper we proposed a number of strengthening techniques that can greatly increase the pruning power of both TNFs and application-specific labels. By iteratively encoding the neighbouring topological information, these techniques were able to significantly reduce matching times while only using the simple TNF of degree. Both the summation and listing indices were able to perform well in all cases, whereas the tree index only proved useful on graphs with fewer edges. With index type, TNFs used, and iteration depth configurable, this framework can be tailored to suit particular problems or classes of graphs as required.

Additionally we presented some new TNFs based on the n-neighbourhoods of nodes. These TNFs use the reduced noise inherent in nNs to identify otherwise-hidden topological incompatibilities between mismatched nodes.

Through analytical and practical experiments, the effectiveness of these techniques has been shown to achieve significant gains over the standard VF2 algorithm. Subsequent testing has also shown that by adding these strengthening techniques to the *iterative node elimination* technique from [4], matching times can be over twice as fast as those reported in that paper, on the same data.

It should be noted that certain constraint programming algorithms, such as LAD, achieve significantly lower gains from these techniques. This is due to additional requirements to integrate TNFs and how such algorithms utilise this information in the matching process. The creation of alternative integration techniques to circumvent this issue remains an open research problem. Another interesting research problem is the creation and matching of topological edge features.

Acknowledgements. The authors would like to thank Gábor Csárdi and Tamás Nepusz for providing the igraph library, including VF2 implementation.

References

1. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. International Journal of Pattern Recognition and Artificial Intelligence 18(3), 265–298 (2004)
2. Cordella, L., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 26(10), 1367–1372 (2004)
3. Csardi, G., Nepusz, T.: The igraph software package for complex network research. Inter. Journal Complex Systems 1695, 1–9 (2006), `http://igraph.sourceforge.net`
4. Dahm, N., Bunke, H., Caelli, T., Gao, Y.: Topological features and iterative node elimination for speeding up subgraph isomorphism detection. In: Proceedings of the 21st International Conference on Pattern Recognition (2012)
5. Fankhauser, S., Riesen, K., Bunke, H., Dickinson, P.: Suboptimal graph isomorphism using bipartite matching. International Journal of Pattern Recognition and Artificial Intelligence (accepted for publication)
6. McKay, B.B.: Practical graph isomorphism. Congressus Numerantium 30, 45–87 (1981)
7. Morgan, H.L.: The generation of a unique machine description for chemical structures - a technique developed at chemical abstracts service. Journal of Chemical Documentation 5(2), 107–113 (1965)
8. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 1st edn. Prentice Hall Press, Upper Saddle River (1995)
9. Solnon, C.: AllDifferent-based filtering for subgraph isomorphism. Artificial Intelligence 174(12–13), 850–864 (2010)
10. Sorlin, S., Solnon, C.: A parametric filtering algorithm for the graph isomorphism problem. Constraints 13, 518–537 (2008)
11. Ullmann, J.R.: An algorithm for subgraph isomorphism. Journal of the ACM 23(1), 31–42 (1976)
12. Zampelli, S., Deville, Y., Solnon, C.: Solving subgraph isomorphism problems with constraint programming. Constraints 15(3), 327–353 (2010)