Topological Features and Iterative Node Elimination for Speeding up Subgraph Isomorphism Detection

Nicholas Dahm[†§], Horst Bunke[¶], Terry Caelli[‡‖], and Yongsheng Gao[§†]

[†]Queensland Research Laboratory / [‡]Victoria Research Laboratory, National ICT Australia
{*nicholas.dahm, terry.caelli, yongsheng.gao*}*@nicta.com.au*

[§]School of Engineering, Griffith University, Australia
{*n.dahm, yongsheng.gao*}*@griffith.edu.au*

[¶]Institute of Computer Science and Applied Mathematics, University of Bern, Switzerland
bunke@iam.unibe.ch

[‖]Electrical and Electronic Engineering, University of Melbourne, Australia
tcaelli@unimelb.edu.au

Abstract

In this paper we tackle the problem of subgraph isomorphism detection on large graphs, which may commonly be intractable, even with state of the art algorithms. Rather than competing with other matching algorithms, we define enhancements that can be used by (almost) any subgraph isomorphism algorithm, both current and future. These enhancements consist of a number of topological features to be added to the nodes, and a technique which we term "iterative node elimination". The fusion of these enhancements is shown to be able to reduce subgraph isomorphism matching times by a factor of over 4,500.

1 Introduction

In structural pattern recognition, graphs are a powerful and flexible data structure. However despite their power, their use is currently limited due to the high computational complexity inherent in graph matching algorithms. In this paper we focus on the difficult problem of subgraph isomorphism, which is known to have an exponential computational complexity (with regards to the number of nodes) in the worst case [5]. Practical applications include bioinformatics, data mining, computer vision, and VLSI design.

In the naive case, subgraph isomorphism can be solved using a brute-force tree search. For small problem instances this may be sufficient, but as the graphs grow, the number of permutations to search becomes intractable. For example, finding subgraph isomorphisms of a 20 node graph in a 30 node graph has 7.3×10^{25} permutations, whereas finding a 200 node graph in a 300 node graph takes 3.2×10^{456} permutations.

To mitigate this computational complexity, various methods have been proposed to prune the search tree. An early and influential algorithm for subgraph isomorphism is Ullmann's algorithm [8], which prunes the search tree by removing branches where past node matches would prevent an isomorphism.

A more recent algorithm for subgraph isomorphism is the VF2 algorithm [1]. As with Ullmann's, VF2 determines which branches cannot lead to an isomorphism. However VF2 does this by searching all nodes neighbouring the current (partial) matching for nodes which are inconsistent with the current matching. In the general case, VF2 is widely accepted as the current state of the art for exact subgraph isomorphism.

For many graph problems, Ullmann's or VF2 will be sufficient, however there are still many cases where these will fail. If we take the SIVALab graph database [4] as an example, we can see that the ground truths are not given for random graphs of 600 nodes. This is because in some of the 600 node cases given, using VF2 to find a single subgraph isomorphism took over a month without success on a 3.20Ghz workstation with superfluous RAM. If we consider that the 1000 node cases have 2.0×10^{759} times more permutations than the 600 node cases, computational time becomes infeasible.

A recent paper by Riesen et al. [6] proposed an alternative approach to graph isomorphism. Firstly, a set of topological features is calculated for each node. The graph nodes are then matched using the Munkres' assignment algorithm without considering edges. This matching is tested for consistency with the graph edges, resulting in a solution or (occasionally) a rejected case. While a solution is not guaranteed, this method boasts a polynomial runtime. However when adapted to subgraph isomorphism with the same set of features, we found that almost all cases were rejected. A similar idea was employed by Zampelli et al. [9] to create the ILF algorithm. ILF uses topological features and constraint programming to provide a solution to subgraph isomorphism problems in a remarkably fast time.

In this paper, we present two techniques to improve subgraph matching runtimes. To be clear, we do not attempt to compete with the above matching algorithms. On the contrary, the techniques we present here are intended to enhance (almost) every exact subgraph isomorphism algorithm, both current and future. This is achieved by enhancing graph features and the associated constraint satisfaction. First, we present a number of *topological node features*, which can be used by almost any graph matching algorithm to improve pruning effectiveness. Second, we present a technique termed *iterative node elimination (INE)*, which can repetitively eliminate nodes not belonging to the matching. As INE removes nodes, the topological node features successively become more refined and more effective. These two enhancements complement each other very well and we will show how their fusion can lead to a significant reduction in matching time.

2 Topological Node Features

In this section we describe a number of topological node features. Each of these features encodes information about the local structure around a node. This information can then be used to determine compatible node matchings between graphs, as if it were part of the node label. Calculating some of these features may be computationally costly, but on large graphs, the reduced matching time justifies the extra feature calculation. Additionally, the initial (not INE refined) values of these features need only be computed once, but can be used many times. These features (or at least the concepts behind them) are not novel, but few have been used as features to determine node compatibilities in a subgraph isomorphism search.

The most basic topological feature is the *degree* of a node, which is the number of nodes adjacent to it. Due to the simplicity of this feature, it is commonly used in many graph operations, and is a built-in constraint of the VF2 algorithm. Another simple feature is a node's *distance to self*. A node's *distance to self* is the minimum path length from a node to itself, traversing each edge at most once. A more complex path feature is the number of n*-walks passing*, which is the number of paths of length n that pass through the node. This feature compiles a vector of values for each $n = 1, 2, \ldots, k$, each of which can be used as a feature. Another vectorial feature is the number of n*-cliques* (cliques of size n) which the node participates in [9]. Using this feature, we get values for each $n = 3, 4, \ldots, k$. Determining the number of cliques can be easily computed for small values of n, but must be capped as the computation can be quite expensive for higher n values. Lastly, the *clustering coefficient* is a measure of how important a node is to its immediate surrounding structure. It is defined as the number of edges between the node's immediate neighbours (not including edges to the node itself). This value defines how well the neighbours are connected, and hence whether the node is structurally important. Our definition is a local adaptation of the global clustering coefficient defined in [2]. On directed graphs, we can use enhanced versions of each of these features, taking edge direction into account.

3 Iterative Node Elimination

One of the drawbacks of using topological node features for subgraph isomorphism is that additional nodes (and their edges) that exist in the full graph but not in the subgraph increase the values of topological features. These additional nodes, which are not part of the final solution, prevent other node incompatibilities from being detected. To overcome this issue, we eliminate nodes that cannot possibly be part of the solution, along with their edges. The full procedure is shown in Algorithm 1.

Firstly, topological features (F_1, F_2) are calculated for the nodes of both graphs (G_1, G_2). From these, a compatibility matrix M is formed showing possible node assignments, where M_{ij} is true if node i from G_2 is a compatible match for node j from G_1. The algorithm then looks at each node in the full graph. If the compatibility matrix reveals that there are no compatible nodes in the subgraph (i.e. $\forall j : M_{ij} = false$), the node is eliminated. Once the nodes have all been processed, the algorithm takes one of three options. If no nodes were eliminated, the algorithm stops as nothing more can be done. If enough nodes were eliminated to make the full graph smaller than the subgraph, then it has been found that no subgraph isomorphism can exist. Otherwise, the algorithm repeats, recalculating all topo-

Algorithm 1 Iterative Node Elimination

```
1:  procedure NODEELIM(G_1, G_2)
2:      F_1 ← features(G_1)                    ▷ G_1 is subgraph
3:      F_2 ← features(G_2)                    ▷ G_2 is full graph
4:      M ← getCompatMat(F_1, F_2)
5:      repeat ← false
6:      for i ← 1, |G_2| do
7:          compat ← false
8:          j ← 1
9:          while j < |G_1| and compat = false do
10:             if M_ij = true then
11:                 compat ← true
12:             end if
13:         end while
14:         if compat = false then
15:             removeNode(i, G_2)
16:             repeat ← true
17:         end if
18:     end for
19:     if repeat = true then
20:         if |G_1| > |G_2| then
21:             output "isomorphism not possible"
22:             end procedure
23:         else
24:             goto: 2
25:         end if
26:     end if
27: end procedure
```

logical features. On each successive iteration, as more nodes are eliminated, the topological features become more refined, and more incompatibilities can be found.

In many cases, the algorithm will result in a rejection and no further processing is needed. However in all others (incl. all positive cases) a graph matching algorithm will have to follow to determine whether a subgraph isomorphism exists. With the additional incompatibilities found by our algorithm, the matching time will often be cut significantly.

4 Experimental Results

In order to gauge the performance of our enhancements, we run a number of tests on the subgraph cases from the SIVALab graph database [4]. This dataset contains three types of graphs, *bounded-valence*, *matrix*, and *random* graphs. Both *bounded-valence* and *matrix* graphs are created with strict conditions and are unlikely in practice, so we will focus on the *random* graphs. To allow results from straight VF2 to be within an acceptable timeframe, we choose the "si6_r01" parameters (subgraph has 60% of the nodes of full graph, and at least $0.1 * |V| * (|V| - 1)$ edges).

Unfortunately the algorithm used to create these random graphs leaves the full graphs *topologically flooded* when compared to their subgraphs. Topological flooding is a condition where every node in the full graph has a higher degree than (almost) every node in the subgraph, greatly limiting the effectiveness of most topological features. To overcome this, a number of randomly chosen edges are removed from the full graph until the topological flooding subsides. After this process, the ratio of Nodes:Edges is similar for G_1 and G_2.

To measure the effectiveness of our techniques we first show their pruning effectiveness. Two common measures of a search tree are the depth, and *average branching factor (ABF)*; see section 3.5 of [7]. Since the tree depth is fixed at V_1, we measure our performance against the ABF. Figure 1 shows how the ABF increases as the graph size increases. We plot ABF curves for each separate feature, for all features, for no features, and for iterative node elimination with all features. The results shown are averaged over 100 test cases at each graph size (200, 400, 600, and 800 nodes).

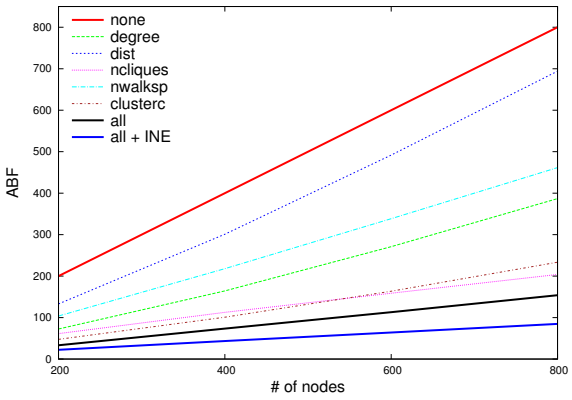

Figure 1. How graph size affects the ABF of different features

The ABF increases approximately linearly for all tests, however with quite different gradients. As expected, the maximum pruning (lowest ABF) is achieved using all features and iterative node elimination. The gradient of this test is especially noteworthy, at just 0.1, down from the gradient of 1, observed for the case where no features and no elimination was used.

This reduction in the size of the search tree is clear, but we must ensure that it is worth the extra processing required to calculate the features. To evaluate this, we run subgraph isomorphism detection with and without our enhancements using the VF2 algorithm, which is commonly used as a benchmark [6, 9]. Our enhance-

ments are implemented in C++ using the igraph library [3], which also provides the VF2 algorithm. All experiments were performed on a 3.2Ghz workstation with superfluous RAM. Figure 2 shows the processing time required to find a single subgraph isomorphism with the various features. In order to show when our enhancements become effective, we use different graph sizes (100, 200, and 400 nodes), and again average our results over 100 cases[1].

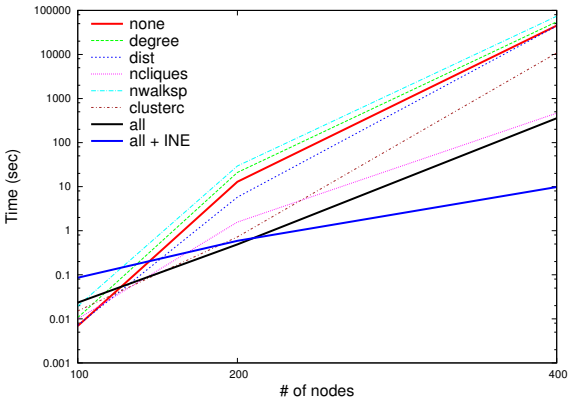

Figure 2. How graph size affects isomorphism detection time with different features

As we see here, for simple graph matching problems ($\leq$ 100 nodes), the calculation of these features takes more time than is saved during the actual matching. However as the difficulty of the problem increases, the feature calculation time is dwarfed by the matching time, and matching with topological features becomes exponentially faster. Iterative node elimination takes even longer to compute, as it may recalculate all the features many times over. This makes it more suited for very difficult matching cases, where the additional pruning can greatly speed up the matching algorithm, making the extra calculation insignificant. This is especially true in the 400 node cases where standard VF2 took an average of 45,212 seconds per graph, whereas INE with all features took under 10 seconds per graph.

5 Conclusions and Future Work

In this paper we have presented a number of enhancements that can be used to greatly speed up the process of subgraph isomorphism. The topological node features shown here can encode information about local graph structure into values which can be used to prune invalid branches from the search tree. To further prune the search tree, we presented our iterative node elimination technique. While node elimination itself is not a new concept, our use of it with topological features allows us to apply it repetitively. With each node that is eliminated, the topological features can be updated to remove any noise which may have been present due to the invalid node. We have shown here that these enhancements can reduce runtimes by a factor of over 4,500 on graphs from a well-known dataset. Aside from graphs that are topologically flooded, these enhancements greatly increase the scope of tractable subgraph isomorphism problems.

In our future work, we will show how topological node features themselves can be strengthened prior to iterative node elimination. In addition, we will expand the list of topological node features, and determine the optimal order to apply them. An interesting open research problem is the application of topological features to maximum common subgraph isomorphism problems.

[1] Due to the long runtime of unenhanced VF2 on 400 node cases, we averaged the results over the first 10 cases.

References

[1] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[2] L. d. F. Costa, F. A. Rodrigues, G. Travieso, and P. R. Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in Physics*, 56(1):167–242, 2007.

[3] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal Complex Systems*, (1695):1–9, 2006.

[4] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In *Proceedings of the 3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 176–187, 2001.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[6] K. Riesen, S. Fankhauser, H. Bunke, and P. Dickinson. Efficient suboptimal graph isomorphism. In A. Torsello, F. Escolano, and L. Brun, editors, *Graph Based Representations in Pattern Recognition*, volume 5534 of *Lecture Nodes in Computer Science*, pages 124–133. Springer Berlin / Heidelberg, 2009.

[7] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 1995.

[8] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[9] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010.