Graph Isomorphism Using a Tree-mapping Approach

Nancy Zaramian

July 10, 2021

Abstract

The graph isomorphism problem asks whether two graphs are essentially the same. Researchers have been looking for a polynomial-time solution for it for many decades. This paper provides an algorithm for the problem using a tree-mapping approach. It converts the two graphs into trees and matches them to provide a mapping. The algorithm also ensures that the conversion and the mapping run in polynomial-time by pruning the tree appropriately as it is built. Experiments are done on the ARG Database [9][7] and the results show that all isomorphisms for graphs with bounded degree were found in a reasonable time. As expected, the results also show that as the average degree of vertices in the graph increases so does the running time. An analysis of the algorithm indicates that the conversion and mapping both run in polynomial-time regardless of the average vertex degree of the graph.

1 Introduction

Graphs are structures that represent relations between data. These structures are implemented in computer programs as data types along with operations to manipulate them.

A graph can represent many real-life structures such as transportation, computer and social networks, atomic structures and molecules. In fact, we can find graph-like structures in almost any field and a natural question is whether two graphs are fundamentally the same. Since the number of vertices and edges can get very large, it is not an easy question to answer. This is the graph isomorphism problem and until recently, the only accepted algorithm was trying all permutations of vertices which requires n! steps. This quickly becomes unmanageable and researchers have been trying to determine whether a more efficient algorithm can be found.

The problem has been addressed on both the practical and theoretical fronts. Starting with the practical approaches, Ullmann [15] first provided a tree-search based algorithm that uses backtracking when looking for an isomorphism. Schmidt and Druffel [14] also provided a backtracking algorithm but this time using a distance matrix which encodes the shortest distances between two vertices. The VF and VF2 algorithms provided by Cordella,

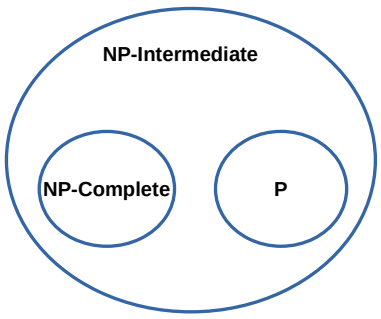
Foggia, Sansone and Vento [5][6] uses a depth-first-search strategy along with a set rules for pruning the search tree. Fundamentally, these are distance-based methods for finding isomorphisms.

Other methods were being developed in parallel using group theory. McKay did the initial work on canonical labeling of graphs which is the main tool for algorithms developed by McKay [12] and McKay and Piperno [13] called Nauty and Traces. The objective of canonical labeling is to provide a standard way of expressing a graph. If two graphs are expressed in a canonized form then they can be compared for isomorphism. This algorithm is considered to have the best performance in terms of practical applications.

A paper by Foggia, Sansone and Vento [8] gives a performance comparison of these methods. The paper shows the strengths and weaknesses of each one by categorizing the results according to graph types. These algorithms are efficient in solving most but not all cases of the problem.

With the results we have so far, graph isomorphism is solved for most practical uses. There is still the unanswered question as to whether a polynomial time algorithm exists for it. In the polynomial hierarchy it is currently suspected to be NP-Intermediate and there is no known polynomial time algorithm that reduces it to an NP-Complete problem. Finding a polynomial time algorithm for it would give some insight into the structure of the polynomial hierarchy and could open the door to solving other related problems such as subgraph isomorphism which is proven to be NP-Complete. Figure 1 provides the structure of the NP class.

Figure 1: NP class hierarchy



On the theoretical front, Luks [10] first started using a group-theoretic

approach when looking at bounded valence graphs which he proved to be solvable in polynomial time. Then, in 1993 Babai, Kantor and Luks [2] provided a theoretical algorithm for general graphs based on the classification of finite simple groups that runs in $2^{O(\sqrt{n \log n})}$ time. This is the best currently accepted running time of an algorithm. Recently Babai [3] put forth an algorithm that runs in quasipolynomial time which is currently under review.

This paper proposes a method that does not rely on finding features in the graphs. Rather, it transforms the graphs into trees and matches those trees to determine whether or not two graphs are isomorphic. It then provides a mapping by matching the nodes in the two trees.

The following sections provide a description of the algorithm along with the implementation, results and conclusions. Note that this paper focuses on simple connected graphs.

2 Algorithm description

This section describes the algorithm for the graph isomorphism problem: given two graphs G1 and G2 is there a mapping between them such that two vertices are neighbors in G1 if and only if the vertices they are mapped to are neighbors in G2?

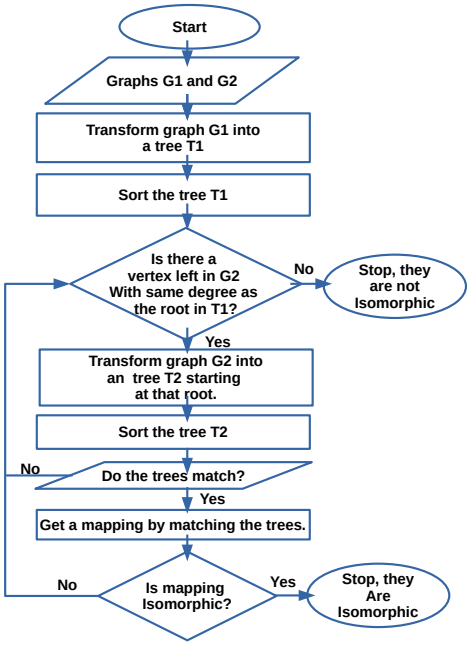
The method is based on converting the graphs to trees, sorting the trees and then finding a mapping by matching the trees. In [1] Aho, Hopcroft, and Ullman provided an algorithm for finding isomorphisms between trees. The method starts by finding roots, creates strings to describe the vertices, sorts the tree in lexicographic order of the strings and then creates a string at the root of the tree describing it completely. A recent paper by Bloyet, Marteau, Frenod[4] describes a "rooted-tree" method for graph canonization. It transforms the graph into a tree which is considered to be a canonical form of the graph. Both of these methods are closely related to the method described here and will be used in combination to provide a graph isomorphism algorithm.

Figure 2 shows the main steps of the algorithm. It receives two graphs G1 and G2. The first step is to transform graph G1 into a rooted tree T1 that represents G1. The following step is to sort tree T1 and create a string S1 at the root of the tree representing it. It then loops through the vertices in G2. If a vertex is found in G2 with the same degree as the root of T1, it transforms graph G2 into a tree T2i with that vertex as the root. It then sorts tree T2i and creates the tree string S2i at its root. If the strings S1

and S2i match, the following step is to map vertices by matching the trees and checking that they are in fact isomorphic. If the strings do not match or the mapping is not isomorphic, it continues the same process for the other vertices in G2 with the same degree as the root of T1. If no matching strings or no isomorphic mapping is found then the graphs are not isomorphic.

In the following sections, the three main sub-algorithms are explained: transforming the graph into a tree, sorting the tree and then finding a mapping of the vertices.

Figure 2: Tree-mapping algorithm for graph isomorphism

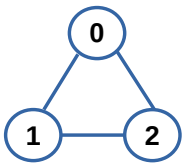


2.1 Transforming the graph into a tree

In this section the graph in figure 3 will be used as an example to show the iterations of the algorithm.

The algorithm starts by choosing any vertex in graph G1. It then adds the first node to the tree T1 which is the root of the tree. The ID of that node is set to the same ID as the chosen vertex. Starting from that vertex in

Figure 3: K_3 graph G1



the graph, it visits its neighbors and adds a child node in the tree for each one of them.

Once all the neighbors of the root are visited, we find ourselves on level 1 of the tree (the root is level 0). On this iteration, we look at the vertices associated to every node on that level of the tree and add the neighbors of those vertices as children to those nodes in the tree except for their parent. Each iteration of the algorithm advances by one step for all vertices associated to the nodes in the current level of the tree. Figure 4 shows what the tree looks like at each iteration.

Note that so far, this looks like the breadth first search algorithm. The difference here is that the vertices are not tagged as being explored at any time. If we meet a vertex more than once, we still add it to the tree with the same ID so there are multiple nodes with the same ID in the tree. Also, when visiting neighbors of a vertex, we do not go back to its direct parent.

While adding nodes to the tree we store information in a string within those nodes. This information encodes all distances between vertices including cycle lengths. If we see a vertex at level 1 of the tree, the string "(1)" is added to that node. If we see that same vertex again at level 4, the string will be "(1,4)" for all nodes with that same ID. This means the vertex is at a distance 1 and 4 from the root and is part of or connected to a cycle. If it is part of a cycle, it encodes the actual cycle length.

After having added the root to the tree, the process continues for $|V|$ iterations which is the number of vertices in graph G1 where V is the set of vertices. This is because the longest cycle in a graph is of length |V| and the tree must describe all possible distances between any two vertices. Note that this algorithm has exponential growth according to the height of the tree and this problem will be explained and addressed later on.

Figure 4: Iterations for creating the tree

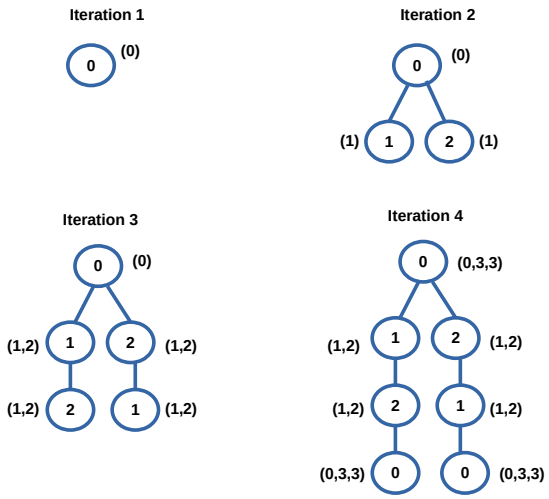
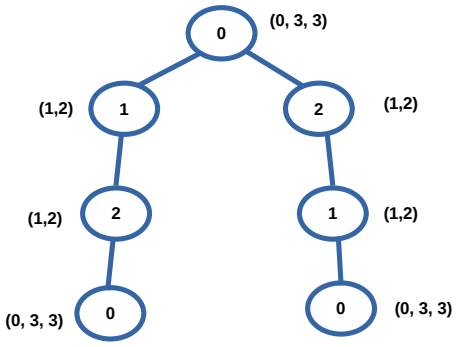


Figure 5: Tree T1 with cycle information



2.2 Sorting the tree

After creating the tree T1, the tree must be sorted. Starting from the leaves, create a string on every node of the tree. The string represents the subtree structure along with the cycle information in the node. For every leaf, add its degree to its descriptor, sort the cycle indices and add them in the node descriptor. Then add the string "()" to indicate it is a leaf. The leaf strings will be initialized in the following way:

(vertex degree, sorted cycle indices, ())

Moving up in the tree, for every parent with complete node descriptions of its children, add its degree to the node descriptor, sort its cycle indices then sort the child descriptors in ascending order of their descriptors. Then create the parent's descriptor in the following way:

(vertex degree, sorted cycle indices, (sorted child descriptors))

The parentheses will describe the subtree structure and the cycle information will be encoded in the string.

In addition to sorting the descriptors of the children, sort the order of the children in each subtree according to the sorted descriptors. This will make sure that subtrees of all children are structurally the same in both trees. This will be used when mapping the vertex IDs.

Move up the tree in this way all the way to the root. The root will have complete tree structure and cycle information of the graph. The string at the root represents the graph in the form of a tree.

Figure 6 shows how the strings are built for the graph in figure 3.

After the tree T1 in figure 5 has been sorted, the string at the root will be the one in figure 7.

Note that for the graph example in figure 3, sorting the children of any parent does not change their order since a node either has one child or the child descriptors are equal.

Figure 6: Sorted Tree T1

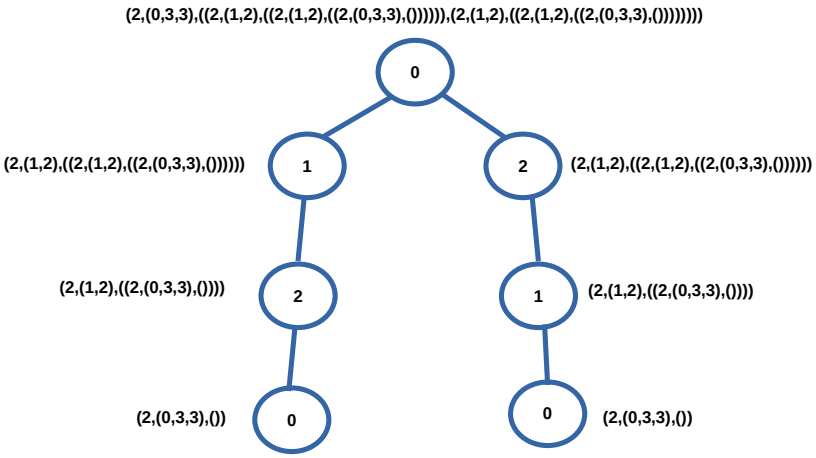


Figure 7: The string S1 representing the tree T1

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

2.3 Matching and Mapping

Once the string at the root of tree T1 is determined, the algorithm searches for a vertex in G2 with the same degree as the root of T1. It then builds tree T2i from that root and determines the string at the root of T2i. If the strings at the root match then the graphs are isomorphic. However, the mapping is not as direct as matching the nodes of the tree. Since many subtrees can have the same structure, their order might not correspond exactly in the matching trees.

The mapping starts by looking at the ordered leaves of the trees. It takes the first unmapped leaf in tree T1 and looks at its lineage all the way up to the root. It assigns the vertex IDs from that lineage to the first lineage in tree T2i. Once the first leaves are processed, the algorithm removes them from both lists.

It continues in the same way for the rest of the leaves. The search always starts at the beginning of the list of leaves since they have been ordered by sorting in the previous step. To assign any new lineage, the algorithm first checks that the existing mapping and non-mapping of the nodes is respected so it searches for a lineage match until it finds one.

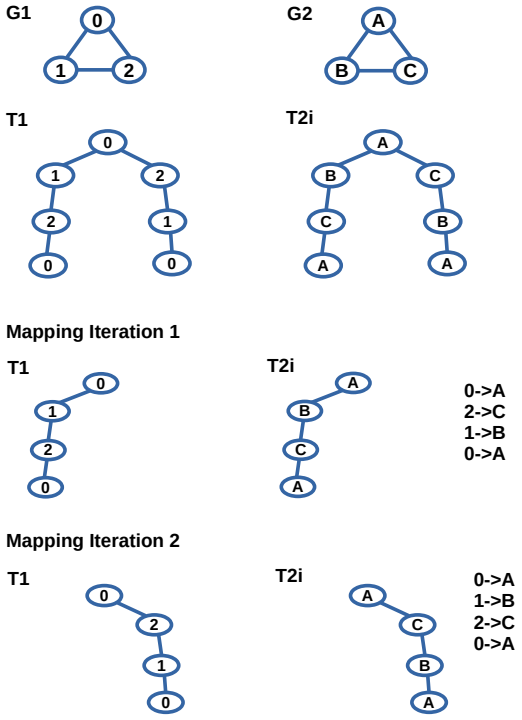
This process ends when either all leaves have been processed or the number of mapped IDs is equal to the number of vertices in the graphs.

3 Discussion

The idea behind the algorithm is to encode all distances between vertices of the graph into the tree.

The tree creation algorithm advances by one level in the tree at every iteration. This means that at every step, the algorithm moves forward by a distance of 1 for all vertices associated to the current level of the tree. Every time a vertex is encountered in the graph, a new node is added to the tree and the level of the tree is recorded within all nodes that have that same ID. The recorded levels represent the cycle lengths the vertex is part of. Since the algorithm iterates $|V|$ times then all cycles will be seen and recorded. This is because the longest distance between two nodes in a graph is $|V|$ which is the longest possible cycle length. The structure of the tree in addition to the cycle information encoded in the nodes provide all distance information necessary to determine graph isomorphism. This structure is encoded in the string at the root and equality between two trees implies isomorphism between the two graphs.

Figure 8: Graphs G1 and G2, trees T1 and T2 and their mappings



This method can be modified to find more than one isomorphism between two graphs. Instead of stopping when one isomorphism is found, the algorithm can continue looping through the vertices of graph G2 and apply the transformation and mapping until all vertices have been processed. It can also be used to find automorphisms of a graph by providing the same graph for both inputs.

The worst-case running time of the algorithm is exponential. Since the same vertex can be met more than once on a given level, that vertex will be split again in the following levels and the vertices can grow exponentially according to the height of the tree. The worst case is a complete graph. Suppose a complete graph has $|V|$ vertices. In the resulting tree, level 0 will have 1 node. Level 1 will have $|V| - 1$ nodes. All subsequent levels will have

$(|V|-1)(|V|-2)^{(l-1)}$ nodes where l is the number of levels. Since there are $|V|$ levels then there will be $(|V|-1)(|V|-2)^{(|V|-1)}$ leaves. This is obviously not a realistic growth rate.

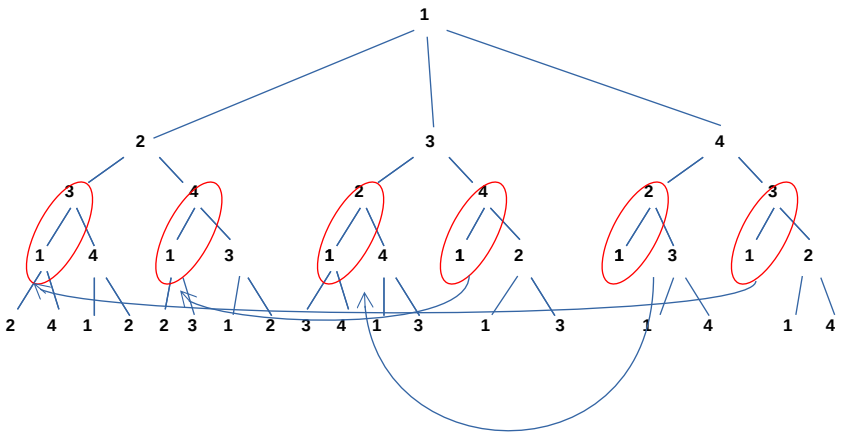
To address this problem, the following section will provide a method to reduce the algorithm to a polynomial number of nodes in the tree.

4 Improvements on the Efficiency of the Algorithm

As mentioned in the previous section, the number of nodes in the tree can grow exponentially according to the height of the tree. There is a way to reduce this to a polynomial number of nodes. The maximum number of unique parent-child pairs is $|V| \cdot |V|$ on any given level. If there are more nodes on any level then non-unique parent-child pairs exist. A subtree on a given level is uniquely defined by a parent and its child. If more than one subtree with the same parent and child is detected then the same subtree is assigned to all of them and the algorithm does not need to explore them all. This ensures that the number of nodes on any given level is not more than $|V|^3$ since there is a maximum of $|V|^2$ unique parent-child combinations.

Figure 9 shows the tree for the complete graph K_4 with the pruned subtrees pointing to the corresponding original subtree.

Figure 9: Tree with subtree pruning for the K_4 graph



When sorting the tree, the subtrees of a given level including the last level are assigned a rank according to its descriptor. This is to avoid exponentially long strings to sort. The unique subtrees of a level are sorted according to their descriptors and then assigned a rank between 0 and the number of unique subtree descriptors. The rank replaces the subtree descriptor and is moved up to its parent and then all the way up to the root in the same way as was done for the descriptor itself.

When checking if the graphs are isomorphic, the strings at the root must be the same. In addition, each subtree associated to a rank must have the same original descriptor between both trees. This ensures that the tree is structurally the same.

Before mapping, all the real subtrees are brought to the left-most side of the tree. This way, the real subtrees will be seen and mapped first. This is required because after building the tree, a sort was applied according to the node descriptors and the real subtrees might not be on the left-most side of the tree after the sort.

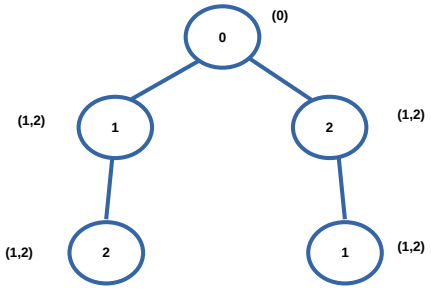
While matching each lineage, a potential subtree mapping must be kept track of for every subtree in case the lineage map fails at some point. That way, the candidate subtree is not remapped every time it is encountered. Also, once a subtree is mapped, it must be noted in the root of the subtree of T2i to ensure a real lineage match without having to check the subtree each time. Checking the parent-child combination on that level ensures the same subtree structure.

Selecting the right root for the tree can help improve the execution time of the algorithm also. One method is to sort the vertices of the graph in ascending order of the frequency of the degrees. The algorithm will start with the vertex with the least degree frequency. That way the number of times the graph G2 is converted to a tree, sorted and mapped is minimized.

Even after reducing the number of nodes in the tree to a polynomial number of $|V|$ the algorithm still takes a very long time to execute in practice. The main issue is the number of nodes added to the tree. To reduce it even more, the concept of a node that is "alive" is introduced. While creating the tree, if the algorithm encounters a node it has already seen on a previous level the node is "killed" and that lineage ends. Note that the node is considered dead for the following level, not the current level during the creation of the tree. If the algorithm encounters that same vertex on the current level of the tree it is still added to the tree until the current level of the tree is complete. Figure 10 shows the resulting tree of the graph shown in figure 3.

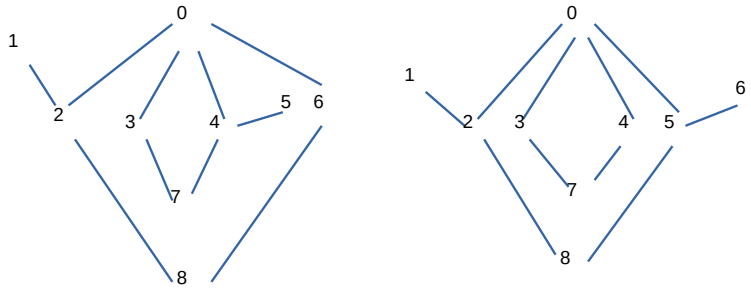
The justification for this new concept is that once a vertex is seen on a previous level all its edges have already been added to the tree so there is

Figure 10: Tree for K_3 after "kill node" algorithm



enough information to determine isomorphism in most but not all cases. The problem with this modification is it does not encode all possible distances between vertices. If the strings at the roots match, it does not mean the graphs are isomorphic. A valid mapping must be found for this to be true. In the case where there is no isomorphism, the algorithm would have to go through all vertices of the graph G2 before declaring it. Figure 11 shows two non-isomorphic graphs that result in the same tree when using this method.

Figure 11: Non-isomorphic graphs that result in the same partial tree



5 Discussion on the complexity of the algorithm

When creating the tree, each level has a maximum of $|V|^3$ nodes. This is because there is a maximum of $|V|^2$ unique parent-child combinations. Once we start having duplicates on a level the subtrees are merged and we only need to explore the unique ones.

The algorithm iterates for $|V|$ times which means that the maximum height of the tree is $|V|$. This means that the maximum number of nodes in the tree is $|V||V|^3$ which limits it to $|V|^4$. In reality, the number of nodes is much less and is dependent on the degrees of the vertices.

Sorting the tree is limited to the number of nodes in the tree multiplied by the sort time of the maximum number of children. Sorting is known to have a worst-case running time that is polynomial with respect to the input. Since the input is $|V|^4$ then sorting the tree can also be done in polynomial time with respect to the number of vertices in the graph.

Searching for a lineage match for each of the leaves is a search algorithm in a sorted list of leaves. Since a simple linear search can be applied, this is also limited to a polynomial worst-case.

Checking for isomorphism is known to have a polynomial worst-case.

6 Implementation and results

The algorithm is implemented using Python and tested on the ARG Database [9][7] with an Intel Core i5 1.19GHz with 8GB of RAM. Trials were done with both the complete tree and the partial tree methods. The results given here are with partial trees. The mappings are verified to be one-to-one and onto. Some cross-testing between different graphs was also done to verify that the algorithm does not indicate there is an isomorphism when there is none.

The following charts show the results using the number of vertices versus the average execution times. The first two graphs show results for quadri-dimensional meshes and the last two for randomly generated graphs.

The results for quadri-dimensional meshes show that execution times increase as the number of vertices increase which is the expected behaviour. During the tests all isomorphisms were found with these types of graphs. Results are comparable between rho=0.4 and rho=0.6 and are given in figures 12 and 13.

Figures 14 and 15 provide results for randomly generated graphs with increased average vertex degrees. In both cases, we see that finding isomorphisms is very quick for graphs with a small number of vertices. As the

Figure 12: Average times for quadri-dimensional meshes with rho=0.4

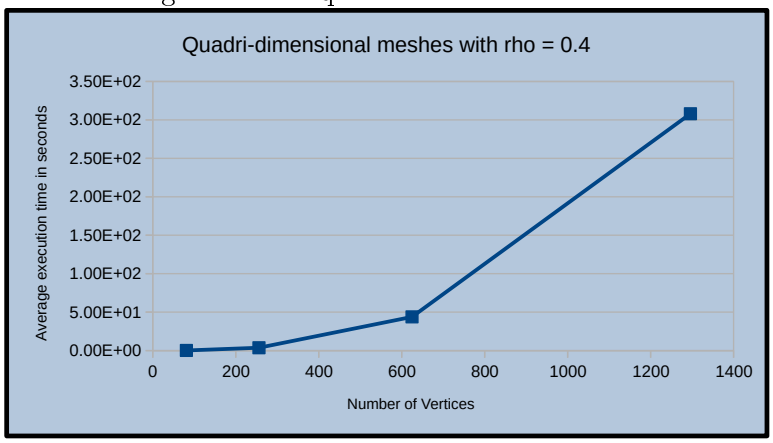
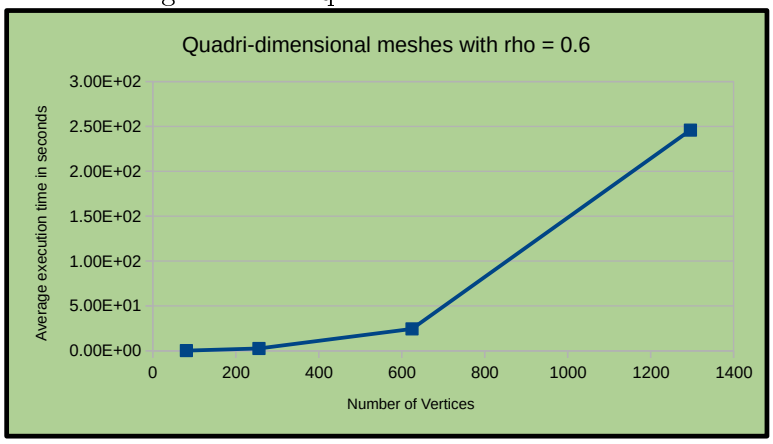


Figure 13: Average times for quadri-dimensional meshes with rho=0.6



the number of vertices grows, the execution time increases dramatically and becomes unmanageable after 400 and 200 vertices respectively. The number of nodes requires a large amount of memory which is a big bottleneck in the execution of the algorithm.

Figure 14: Average times for randomly generated graphs with eta = 0.05

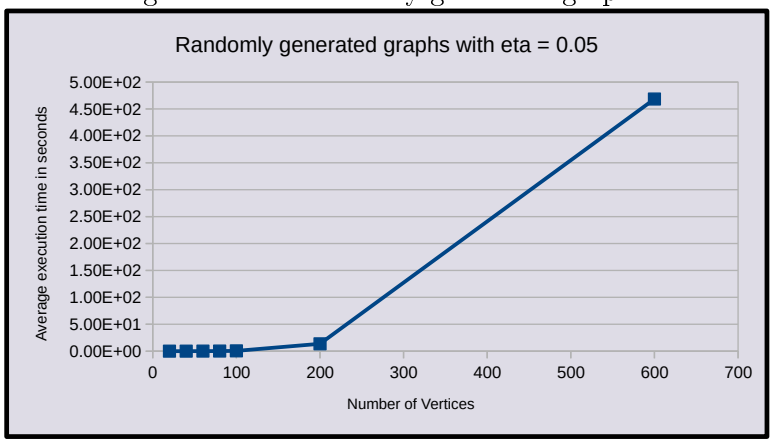
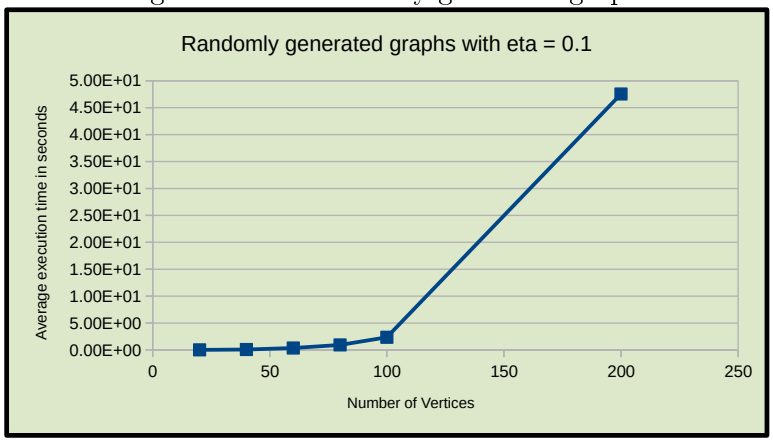


Figure 15: Average times for randomly generated graphs with eta = 0.1



7 Future work

Theoretical work is required to determine whether two graphs are isomorphic if and only if a matching tree structure exists. Another question is whether the mapping will always provide a valid one when two graphs are isomorphic. A complexity analysis must also be looked at to provide a more precise upper bound on the running time of the algorithm and also on its space complexity.

Following that, there are many optimizations that should be looked at to improve the execution times in practice. The number of nodes can grow very quickly when building the trees. Can this be reduced even more? Also, can parallelization be used for building the tree to improve its efficiency?

There is also the question of subgraph isomorphism which, as mentioned in the introduction, is an NP-Complete problem. Ullman provided a backtracking algorithm for the subgraph isomorphism problem in [15][16] which remains exponential in the worst-case. Given that subtree isomorphism has a polynomial time solution [11], the algorithm presented in this paper might be a good candidate to consider for the subgraph isomorphism problem since it represents the graph in a tree-form.

8 Conclusion

This paper provides a tree-mapping approach for graph isomorphism that decouples the identification of isomorphism from the one of mapping. The algorithm opens the door to practical and efficient implementations of a solution to the problem. In addition, it links graph isomorphism to subgraph isomorphism through the tree structure thereby providing an avenue to explore NP-Complete problems.

References

[1] A. Aho, J. Hopcroft, and J. Ullman The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Co., Reading, MA, 1974, pp. 84-85.

[2] Babai, L., Kantor, W. M., & Luks, E. M. (1983, November). Computational complexity and the classification of finite simple groups. In 24th Annual Symposium on Foundations of Computer Science (Sfcs 1983) (pp. 162-171). IEEE.

[3] Babai, L., 2016, June. Graph isomorphism in quasipolynomial time. In Proceedings of the forty-eighth annual ACM symposium on Theory of Computing (pp. 684-697).

[4] Bloyet, N., Marteau, P.F. and Frenod, E., 2019, December. Scott: A method for representing graphs as rooted trees for graph canonization. In International Conference on Complex Networks and Their Applications (pp. 578-590). Springer, Cham.

[5] Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (1999, September). Performance evaluation of the VF graph matching algorithm. In Proceedings 10th International Conference on Image Analysis and Processing (pp. 1172-1177). IEEE.

[6] Cordella, L. P., Foggia, P., Sansone, C., & Vento, M. (2001, May). An improved algorithm for matching large graphs. In 3rd IAPR-TC15 workshop on graph-based representations in pattern recognition (pp. 149-159).

[7] De Santo, M., Foggia, P., Sansone, C., & Vento, M. (2003). A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recognition Letters, 24(8), 1067-1079.

[8] Foggia, P., Sansone, C. and Vento, M., 2001, May. A performance comparison of five algorithms for graph isomorphism. In Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition (pp. 188-199).

[9] Foggia, P., Sansone, C. and Vento, M., 2001, May. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In Proc. of the 3rd IAPR TC-15 International Workshop on Graph-based Representations (pp. 176-187).

[10] Luks, E. M. (1982). Isomorphism of graphs of bounded valence can be tested in polynomial time. Journal of computer and system sciences, 25(1), 42-65.

[11] Matula, D. W. (1978). Subtree isomorphism in O (n5/2). In Annals of Discrete Mathematics (Vol. 2, pp. 91-106). Elsevier.

[12] McKay, B.D., 1981. Practical graph isomorphism (pp. 45-87). Tennessee, USA: Department of Computer Science, Vanderbilt University.

[13] McKay, B. D., & Piperno, A. (2014). Practical graph isomorphism, II. Journal of Symbolic Computation, 60, 94-112.

[14] Schmidt, D. C., & Druffel, L. E. (1976). A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. Journal of the ACM (JACM), 23(3), 433-445.

[15] Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. Journal of the ACM (JACM), 23(1), 31-42.

[16] Ullmann, J. R. (2011). Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. Journal of Experimental Algorithmics (JEA), 15, 1-1.

Appendices

A Pseudocode

A.1 Converting a graph to a full tree

```
def ConvertGraphToFullTree(G, VertIdx):
   CurLevel = 0
   NodeIdx = 0
   T = Tree()

   if G.NumVertices() <= 0
      return T

   G.VertLst[VertIdx].CurLevel = CurLevel
   VertQueue = [G.VertLst[VertIdx]]
   ParntInTreeQueue = []

   Vert = VertQueue.Pop()
   Vert.Parnt = None
   VertCurLevel = Vert.CurLevel

   CurVertParnt = Vert.Parnt

   while CurLevel <= G.NumVertices() and Vert != None:
      ChldParntLst = {}
      while (VertCurLevel == CurLevel and Vert != None):
         Vert.Cycles.Append(CurLevel)

         T.AddVert(NodeIdx)
         T.VertLst[NodeIdx].SetId(Vert.Id())
         T.VertLst[NodeIdx].Cycles = Vert.Cycles

         if (ParntInTreeIdx is not None):
            T.AddEdge(ParntInTreeIdx, NodeIdx)

            ParntChldSubTr = None

            CurParntNodeId = T.VertLst[ParntInTreeIdx].Id()

            if Vert.Id() in ChldParntLst:
               if CurParntNodeId in ChldParntLst[Vert.Id()]:
                  ParntChldSubTr =
                  ChldParntLst[Vert.Id()][CurParntNodeId]

         if (ParntChldSubTr is not None):
            T.VertLst[NodeIdx].SubTr = ParntChldSubTr
         else:
            T.VertLst[NodeIdx].SubTr = T.VertLst[NodeIdx]

         NodeIdx = NodeIdx + 1

         if (ParntChldSubTr is None):
            if (ParntInTreeIdx is not None):
```

```
                CurNodeId = T.VertLst[NodeIdx-1].Id()
                CurParntId = T.VertLst[ParntInTreeIdx].Id()

                ChldParntLst[CurNodeId][CurParntId].Append(
                           T.VertLst[NodeIdx-1].SubTr)

            for Neighbr in Vert.Neighbrs:
                if (CurVertParnt == None or Neighbr.Id() !=
                            CurVertParnt.Id()):
                    Neighbr.CurLevel = CurLevel + 1
                    Neighbr.Parnt = Vert
                    VertQueue.Append(Neighbr)
                    ParntInTreeIdxQueue.Append(NodeIdx-1)

        if (VertQueue):
            Vert = VertQueue.Pop()
            CurVertParnt = Vert.Parnt
            VertCurLevel = Vert.CurLevel
            ParntInTreeIdx = ParntInTreeIdxQueue.Pop()
        else:
            Vert = None

    CurLevel += 1

return T
```

A.2 Converting a graph to a partial tree

```
def ConvertGraphToPartialTree(G, VertIdx):
    CurLevel = 0
    NodeIdx = 0
    T = Tree()

    if G.NumVertices() <= 0
        return T

    G.VertList[VertIdx].CurLevel = CurLevel
    VertQueue = [G.VertList[VertIdx]]
    ParntInTreeQueue = []

    VerticesToKill = []
    LiveVertices = True

    Vert = VertQueue.Pop()
    Vert.Parnt = None
    VertCurLevel = Vert.CurLevel

    CurVertParnt = Vert.Parnt

    while LiveVertices and Vert != None:
        LiveVertices = False
        ChldParntList = {}

        while (VertCurLevel == CurLevel and Vert != None):
            Vert.Cycles.Append(CurLevel)
```

```
28
29          T.AddVert(NodeIdx)
30          T.VertList[NodeIdx].SetId(Vert.GetId())
31          T.VertList[NodeIdx].Cycles = Vert.Cycles
32
33          if (ParntInTreeIdx is not None):
34             T.AddEdge(ParntInTreeIdx, NodeIdx)
35
36             ParntChldSubTr = None
37
38             CurParntNodeId = T.VertList[ParntInTreeIdx].GetId()
39
40             if Vert.GetId() in ChldParntList:
41                if CurParntNodeId in ChldParntList[Vert.GetId()]:
42                   ParntChldSubTr = ChldParntList[Vert.GetId()][
    CurParntNodeId]
43
44          if (ParntChldSubTr is not None):
45             T.VertList[NodeIdx].SubTr = ParntChldSubTr
46          else:
47             T.VertList[NodeIdx].SubTr = T.VertList[NodeIdx]
48
49          NodeIdx = NodeIdx + 1
50
51          if (Vert.Alive and ParntChldSubTr is None):
52             LiveVertices = True
53
54             if (ParntInTreeIdx is not None):
55                CurNodeId = T.VertList[NodeIdx-1].GetId()
56                CurParntId = T.VertList[ParntInTreeIdx].GetId()
57
58                ChldParntList[CurNodeId][CurParntId].Append(T.VertList[
    NodeIdx-1].SubTr)
59
60             for Neighbr in Vert.Neighbrs:
61                if (CurVertParnt == None or Neighbr.GetId() !=
    CurVertParnt.GetId()):
62                   Neighbr.CurLevel = CurLevel + 1
63                   Neighbr.Parnt = Vert
64                   VertQueue.Append(Neighbr)
65                   ParntInTreeIdxQueue.Append(NodeIdx-1)
66
67             VerticesToKill.Append(Vert)
68
69          if (VertQueue):
70             Vert = VertQueue.Pop()
71             CurVertParnt = Vert.Parnt
72             VertCurLevel = Vert.CurLevel
73             ParntInTreeIdx = ParntInTreeIdxQueue.Pop()
74          else:
75             Vert = None
76
77       while (VerticesToKill):
78          VertToKill = VerticesToKill.Pop()
79          VertToKill.Alive = False
80
```

```
        CurLevel += 1

    return T
```

A.3 Sorting the tree

```
def SortTree(T):

    for i in range(T.NumVertices()):
        T.VertList[i].Desc = ""
        T.VertList[i].SubTrRank = -1

    VertIdx = T.NumVertices()-1

    while VertIdx>= 0:
        CurLevelVertices = []
        ChldDescList = []
        CurLevel = T.VertListLevelOrder[VertIdx].TreeLevel

        while VertIdx>=0 and T.VertListLevelOrder[VertIdx].TreeLevel ==
     CurLevel:
            CurLevelVertices.Append(T.VertListLevelOrder[VertIdx])
            VertIdx -= 1

        for V in CurLevelVertices:
            if (V.SubTr.Desc == ""):
                ChldDescList.clear()
                Neighbrs = V.SubTr.Neighbrs

                for Neighbr in Neighbrs:
                    ChldDescList.Append(str(Neighbr.SubTr.SubTrRank))
                ChldDescList.Sort()

                V.SubTr.Cycles.Sort()
                V.SubTr.Desc += "("
                V.SubTr.Desc += str(V.SubTr.getNumNeighbrs())
                V.SubTr.Desc += ","

                for C in V.SubTr.Cycles:
                    V.SubTr.Desc += str(C)
                    V.SubTr.Desc += ","

                V.SubTr.Desc += "("

                for C in ChldDescList:
                    V.SubTr.Desc += str(C)
                    V.SubTr.Desc += ","

                V.SubTr.Desc += "))"

                T.SortChldren(V.SubTr.VertIdx)

            V.Desc = V.SubTr.Desc

        CurLevelRealSubTrs = []
```

```
49
50      for C in CurLevelVertices:
51          if C.VertIdx == C.SubTr.VertIdx:
52              CurLevelRealSubTrs.Append(C)
53
54      CurLevelRealSubTrs.SortByChldDesc()
55
56      CurRank = 0
57
58      if (CurLevelRealSubTrs.Size() > 0):
59          CurLevelRealSubTrs[0].SubTrRank = CurRank
60
61      for i in range(1, CurLevelRealSubTrs.Size()):
62          if CurLevelRealSubTrs[i].Desc != CurLevelRealSubTrs[i-1].Desc
    :
63              CurRank += 1
64          CurLevelRealSubTrs[i].SubTrRank = CurRank
65
66      for i in range(0, CurLevelRealSubTrs.Size()):
67          T.CompleteSubTrListLevelOrder.Insert(0, CurLevelRealSubTrs[i
    ])
```

A.4 Determining the mapping

```
1  def GetMapping(G1, G2, T1, T2):
2      GraphMapping = {}
3
4      T1Leaves = T1.BringSubTrsToLeft()
5      T2Leaves = T2.BringSubTrsToLeft()
6
7      T1Leaves = T1.GetLeaves()
8      T2Leaves = T2.GetLeaves()
9
10     for V1 in T1Leaves:
11         V2Idx = 0
12
13         for V2 in T2Leaves:
14             if LineageMatch(G1, G2, V1, V2, T1, T2, GraphMapping):
15                 AssignLineage(V1, V2, T1, T2, GraphMapping)
16                 T2Leaves.pop(V2Idx)
17                 break
18
19             V2Idx += 1
20
21             if len(GraphMapping) == G1.NumVertices():
22                 break
23     return GraphMapping
24
25
26 %#########################################################
27 def LineageMatch(G1, G2, StartNode1, StartNode2, T1, T2, Mapping):
28     CurNode1 = StartNode1
29     CurNode2 = StartNode2
30     TempMapping = {}
31
```

```
   Isleaf1_RealLeaf = (StartNode1.SubTr.NumNeighbrs() == 0)
   Isleaf2_RealLeaf = (StartNode2.SubTr.NumNeighbrs() == 0)

   if not (Isleaf1_RealLeaf and Isleaf2_RealLeaf):
      while CurNode1.TreeLevel < CurNode2.TreeLevel:
         CurNode2 = T1.VertList[CurNode2.ParntIdx]

      while (CurNode2.TreeLevel < CurNode1.TreeLevel):
         curNode1 = T1.VertList[curNode1.Parnt[0]]

   if not LeavesComparable(StartNode1, StartNode2, CurNode1, CurNode2)
    :
      return False

   IsMatch = True

   while True:
      if CurNode1.GetId() in Mapping:
         if (mapping[curNode1.GetId()] != CurNode2.GetId()):
            IsMatch = False
            return IsMatch
      else:
         for V in Mapping:
           if Mapping[V] == CurNode2.GetId():
              IsMatch = False
              return IsMatch

      if (CurNode1.GetId() in TempMapping) and (TempMapping[CurNode1.
    GetId()] != CurNode2.GetId()):
         IsMatch = False
         return IsMatch
      else:
         if (CurNode1.GetId() not in TempMapping):
            for V in TempMapping:
               if TempMapping[V] == CurNode2.GetId():
                  IsMatch = False
                  return IsMatch

      if G1.VertList[CurNode1.GetId()].NumNeighbrs() != G2.VertList[
    CurNode2.GetId()].NumNeighbrs():
         IsMatch = False
         return IsMatch

      TempMapping[CurNode1.GetId()] = CurNode2.GetId()

      if not CurNode2.SubTr.MySubTrIsACandidate and CurNode2.SubTr.
    NumNeighbrs()!= 0):
         if (CurNode1.SubTr.SubTrUsageCount > 0) and (CurNode1.SubTr.
    SubTrUsageCount == CurNode2.SubTr.SubTrUsageCount):
            CurNode2.SubTr.MySubTrIsACandidate = True
            CurNode2.SubTr.SubTrLineagePotentialMapping[CurNode1.GetId
    ()] = CurNode2.GetId()

      IsCurNode1Root = CurNode1.Parnt == None
      IsCurNode2Root = CurNode2.Parnt == None

```

```
      if (IsCurNode1Root) != (IsCurNode2Root):
         IsMatch = False
         return IsMatch
         break

      if IsCurNode1Root and IsCurNode2Root:
         return True
         break

      T2.VertList[CurNode2.Parnt].SubTr.SubTrLineagePotentialMapping =
     CurNode2.SubTr.SubTrLineagePotentialMapping

      CurNode1 = T1.VertList[CurNode1.Parnt]
      CurNode2 = T2.VertList[CurNode2.Parnt]

   return isMatch

%#########################################################
def AssignLineage(StartNode1, StartNode2, T1, T2, Mapping):
   CurNode1 = StartNode1
   CurNode2 = StartNode2

   while CurNode1.TreeLevel < CurNode2.TreeLevel:
      CurNode2 = T1.VertList[CurNode2.Parnt]

   while CurNode2.TreeLevel < CurNode1.TreeLevel:
      CurNode1 = T1.VertList[CurNode1.Parnt]

   if CurNode2.SubTr.MySubTrIsACandidate:
      for id in CurNode2.SubTr.SubTrLineagePotentialMapping:
         if id not in Mapping:
            Mapping[id] = CurNode2.SubTr.SubTrLineagePotentialMapping[
    id]

   while True:
      if (CurNode1.id not in Mapping):
         Mapping[CurNode1.id] = CurNode2.id

      CurNode2.SubTr.SubTrLineageMappingIsComplete = True
      if CurNode1 != T1.VertList[0] and CurNode2 != T2.VertList[0]:
         CurNode1 = T1.VertList[CurNode1.Parnt]
         CurNode2 = T2.VertList[CurNode2.Parnt]
      else:
         break

   for i in range(T2.NumVertices()):
      T2.VertList[i].MySubTrIsACandidate = False
      T2.VertList[i].SubTrLineagePotentialMapping = {}

%#########################################################
def ComparableLeaves(Leaf1, Leaf2, Leaf1_AtLevel, Leaf2_AtLevel):

   Isleaf1_RealLeaf = (Leaf1.SubTr.NumChldren() == 0)
   Isleaf2_RealLeaf = (Leaf2.SubTr.NumChldren() == 0)

   if IsLeaf1_RealLeaf and IsLeaf2_RealLeaf:
```

```
      if Leaf1.SubTr.TreeLevel == Leaf2.SubTr.TreeLevel:
         return True
      else:
         return False

   if not IsLeaf1_RealLeaf:
      return Leaf2_Atlevel.SubTr.SubTrLineageMappingIsComplete
   else:
      return Leaf2.SubTr.MySubTrIsACandidate
```

A.5 The main matching function

```
def Match(G1, G2, VertIdx):
   T1 = ConvertGraphToPartialTree(G1, VertIdx)
   #T1 = ConvertGraphToFullTree(G1, VertIdx)

   SortTree(T1)

   T2 = None
   NumIso = 0

   for i in range G2.NumVertices()):
      if (G2.VertList[i].NumNeighbrs() == G1.VertList[VertIdx].
    NumNeighbrs()):
         T2 = ConvertGraphToPartialTree(G2, i)
         #T2 = ConvertGraphToFullTree(G2, i)

         SortTree(T2)

         if (TreesMatch(T1, T2)):
            Mapping = GetMapping(G1, G2, T1, T2)

            if CheckIsomorphic(G1, G2, Mapping):
               NumIso += 1

               break
            T1.ResetMappingInfo()

   return numIso, Mapping

%#########################################################
def TreesMatch(T1, T2):
  IsMatch = True

  T1_SubTrLen = len(T1.SubTrList())
  T2_SubTrLen = len(T2.SubTrList())

  IsMatch = (T1_SubTrLen == T2_SubTrLen)

  for i in range(T1_SubTrLen):
     IsMatch = IsMatch and (T1.SubTrList[i].Desc == T2.SubTrList[i].
    Desc)

     if not IsMatch:
        break
```

```
42
43  return IsMatch
```