International Journal of Computer Applications (0975 - 8887)
Volume 176 - No.12, April 2020

Graph Isomorphism Algorithm using Pieces Patching Puzzle Technique ($ppp - Technique$)

Mohammad Alhashmi
Department of Computer and Information Systems
College of Business
PAAET Kuwait

Abdulaziz Alroomi
Department of Computer and Information Systems
College of Business
PAAET Kuwait

ABSTRACT

This paper proposes a novel technique for a polynomial time algorithm to detect an existence of isomorphism between two unlabeled graphs that is fast and accurate for the mass majority of large random graphs. This technique, namely $ppp - Technique$, is based on cutting the first graph into large sub-graphs to mimic the pieces (or *patches*) of a puzzle that have to be patched to their correct places (or *matches*) on the second graph. It is a difficult process to draw the borders of each patch for best results. In other words, deciding the sizes of the patches is an optimization problem. Clearly, the larger these patches are, the faster the algorithm can decide whether an isomorphism exists. The greedy concept has been applied in the process of creating patches which led to high efficiency. The time complexity of the proposed algorithm is $O(n^3 \log n)$. Examples that clarify the process of constructing the patches from one graph and matching them to the places in the other one are shown. The algorithm is tested with many graphs with different sizes of nodes and different densities of connecting edges which gave complete accurate results.

General Terms

Computer Science, Algorithms, Graphs

Keywords

Unlabeled Graphs, Random Graphs, Graph Matching, Isomorphism

1. INTRODUCTION

Graph Isomorphism (GI), a problem of deciding whether two graphs are similar under certain conditions, is among the most important challenges of graph processing [1] and is a vital problem in the field of Computer Science. Because of its significant role in a large variety of applications today such as image matching, biochemistry, and information retrieval [2], scientists and researchers have been trying to find an easy solution to this problem for more than 40 years and no one succeeded fully yet [3] [4]. Therefor, it remains one of many unresolved computational problems with unknown complexity status such as those from Garey and Johnsons list dating back to 1979 [5].

Problems are categorized into two groups according the complexity theory, namely P and NP-complete [4], based on the time needed to solve these problems when their sizes grow large. Simply, when the size of a given problem increases and the time of the solving algorithm grows only polynomially, then it is considered an easy problem and belongs to the category P denoting deterministic polynomial time solution. On the other hand, and with the increase of the size of the problem, when the time an algorithm needs to find a solution increases exponentially then this problem is considered a hard and difficult one. For some of these hard problems with exponential time complexity, such as the boolean satisfiability problem (SAT), knapsack problem, and hamiltonian path problem, scientists have been able to provide non-deterministic polynomial time algorithms. These types of problems are said to be NP-complete. For the GI problem, neither scientists could prove them to be NP-complete, nor anyone have been able to provide a solution in polynomial time [6]. That is why researchers are still trying to come up with new algorithms in the hope that one day they find a polynomial time solution.

The proposed solution mimics the process of solving puzzles by patching the different pieces in their correct places on the image. The difficulty in this process arises when a piece independently may correctly be patched to many places on the image. This difficulty increases as the number of such pieces increase.

1.1 Graph Definitions

Solving the GI problem is by finding a one to one mapping (a bijection) between the nodes of the two graphs such that the relations (adjacencies) are preserved [7].

Objects in real world can be represented with graphs, where a graph consists of $vertices$ (or $nodes$) that represent the different parts of the object and $edges$ that connect the nodes which represent the relations between these different parts of the object.

Due to this representation of objects, graph isomorphism or graph matching algorithms in general were applied on a variety of fields such as Computational Biology, Chemistry, Information Retrieval and Medicine [8]. Using graph representation made it easy for very large systems to benefit from graph isomorphism algorithms such as the analysis of protein to protein interaction in complex biological systems [9].

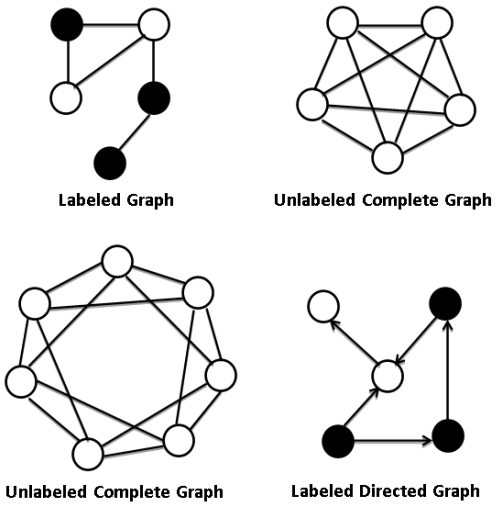

Fig. 1. Sample Graphs Representations

When a node is connected to another one, they are called neighbors or one is a child of the other. A $degree$ of a node is equal to the number of neighbors or children connected to it. This is the simplest representation of objects and the type of these graphs is called $unlabeled\ graphs$.

There are many other categories and types of graphs based on a set of given values that are assigned to nodes and/or edges. Some of these types are: $labeled\ graphs$, $directed\ graphs$, and $undirected\ graphs$. For another categorization of graphs based on the topology of the graph connectivity, i.e. the way the nodes are connected via edges, we have many different types of graphs. Examples of that are $regular\ graphs$ where every node in the graph has the same degree and $complete\ graphs$ where every node is connected to every other node in the graph. When the whole graph is one piece, it is called a $connected\ graph$. Figure 1 shows examples of different categories and types of graphs.

A graph $G = (V, E)$ is represented by a set V of vertices (or nodes) and a set E of edges. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, then the graph isomorphism (GI) problem is to find a bijection function $f : V_1 \rightarrow V_2$ such that $(u, v) \in E_1$ iff $(f(u), f(v)) \in E_2$. The technique that tries to find whether the function f is available is explained next.

1.2 Pieces Patching Puzzle Technique (*ppp-Technique*)

The graph isomorphism detection process is performed mimicking the puzzle solving process. Instead of pursuing random trial and error, an educated trial and error is followed in trying to match the different patches against the original image. This approach of the algorithm is named *Pieces Patching Puzzle Technique (ppp-Technique)*.

Given two unlabeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, G_2 is considered the original image and patches (subgraphs) are create from G_1. The process then starts by trying to patch these subgraphs correctly onto G_2. The next section shows the process of creating the patches followed by a section explaining the technique of patching them against the original graph.

2. THE PPP-TECHNIQUE

2.1 Definitions

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, first G_1 is cut into patches (subgraphs of G_1) where each Patch (subgraph) is a *star graph*. Star graphs are those with a central vertex where every other vertex (a child) in the graph is connected only to this central vertex [10] as shown in Figure 2(A). A variation of a star graph is proposed here by relaxing the definition of the star graph and is called *pseudo-star* graph. A pseudo-star graph has a central vertex that is connected to all children vertices but also allowing the children to have additional connections among themselves such as the one shown in Figure 2(B).

Since the algorithm works on unlabeled graphs, the only data available about the graphs is the degree of the vertices and the connecting edges. The degree of a vertex v denoted by $deg(v)$, is the number of vertices (children) it is connected to via edges. Each patch p (or *pseudo-star* graph) created is assigned an integer number as a *patch label* (pl). Let $pl(p_i) = m$ where m is an integer denoting the patch label for p_i. The integer m is constructed by accumulating the degrees of the children of the central vertex of patch p_i or simply of the children of patch p_i in a special way that will be explained. Each patch is also given another label which is the *central vertex* (cv) in it. Let $cv(p_i) = c$, then c denotes the number of the central vertex in patch p_i. For example, $pl(p_1) = 332$ and $cv(p_1) = 6$ where p_1 is one possible patch that is created from the unlabeled graph of Figure 3. Notice that p_1 is a *pseudo-star* graph with the central vertex being vertex 6, i.e, $cv(p_1) = 6$. The integer 332 represents the degrees of the children of vertex 6 (vertices 4, 7, and 5) from left to right in descending order. Notice that $deg(v_4) = 3, deg(v_5) = 2,\ and\ deg(v_7) = 3$. Likewise, $pl(p_2) = 21$ and $cv(p_2) = 2$.

The patches may consist of only a central vertex and only one child or even they may consist of only the central vertex alone. Figure 3 shows a third patch with $pl = 2$ and $cv = 10$. After creating all possible patches, there may be single vertices that are considered tiny patches and are given $pl = 0$, i.e. the central vertex has zero children as far as patches are concerned.

There are 3 edges crossing or outgoing from p_1. One connecting v_4 with v_3, another connecting v_7 with v_8, and the last connecting v_7 with v_9. These edges are considered as special ones and each of them is denoted by *patch crossing edge* (pce) and is given a label. A label of a special edge connecting v_i with v_j is denoted by $pcel(v_i, v_j) = qr$ where $q = deg(v_i)$ and $r = deg(v_j)$. Hence, $pcel(v_4, v_3) = 32$, $pcel(v_7, v_8) = 32$, and $pcel(v_7, v_9) = 31$.

For the implementation of the algorithm, which is done in Java, two important issues are to be mentioned here about patch labels $pl's$ and patch crossing edge labels $pcel's$ such as $pl(p_1) = 332$ and $pcel(v_4, v_3) = 32$ in the example above. First, the integers 332 and 32 may get very huge in large graphs with degrees of some nodes being in hundreds or thousands or more. In this case, pl may be with hundreds or thousands of digits where integers are not supported with this magnitude on most computer systems. Therefore, the integers 332 and 32 are dealt with as strings ”332” and ”32” to have enough width. Furthermore, normal sorting for strings considers for example the string ”23456” less than ”53”. Hence, a class $Patch$ is implemented to perform a special sorting order for the patch labels so that ”23456” is considered larger than ”53” to be able to apply the greedy concept that is explained in the next Section.

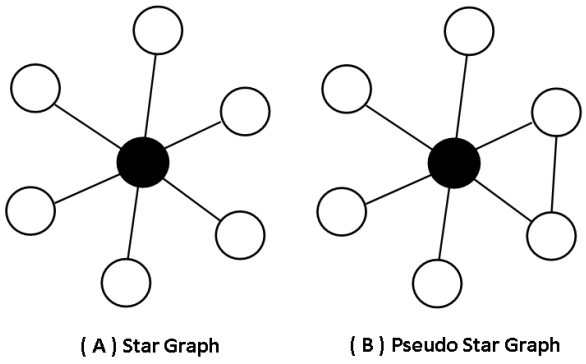

Fig. 2. Star and Pseudo-Star Graphs

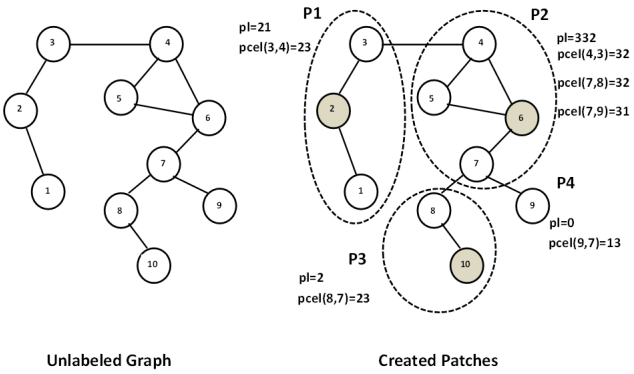

Fig. 3. Creating Patches (pseudo-star graphs)

2.2 Creating Patches and Patch Crossing Edges

Given an unlabeled graph $G = (V, E)$ such as the one shown in Figure 3, we create a set of patches $P = \{p_1, p_2, ..., p_n\}$ in the following way.

The greedy heuristic or concept that has been used in many algorithms to solve a variety of problems [11] [12] [13] [14] [15] led to solutions close to optimal in acceptable time. In this paper, the greedy concept is applied in creating the patches so that later on in the process of finding matches for these patches they are tried in the order of their creation. The first patch created is the first encountered with maximum pl. Then the second patch is created and so on and so forth.

In the example of Figure 3, the first patch p_1 is one with a central vertex of the highest degree. That patch would be either the one with central vertex v_6 and its children $v_4, v_5, \ and \ v_7$ or the one (not shown) with central vertex v_7 and its children $v_6, v_8, \ and \ v_9$. So in this case p_1 is created with $pl(p_1) = 332$ and $cv(p_1) = 6$ along with the 3 crossing edges with $pcel(v_4, v_3) = 32$, $pcel(v_7, v_8) = 32$, and $pcel(v_7, v_9) = 31$. The next patch will be created finding a central vertex with highest degree among the remaining vertices that are not already included in any patch as well as its children. Notice that v_4 and v_7 are the next highest degree vertices but they have already been included in p_1. Also notice that vertices $v_3 \ and \ v_8$ can not be next even though their degrees are as that of v_2 because some of their children are already included in previously created patches. So interference is not allowed among the patches. Thus, v_2 is the next choice and p_2 is created with $pl(p_2) = 21$ and $cv(p_2) = 2$ along with one crossing edge with $pcel(v_3, v_4) = 23$. Next, p_3 is one with $pl(p_3) = 2$ and $cv(p_3) = 10$ along with one crossing edge with $pcel(v_8, v_7) = 23$. The remaining single vertices are considered patches as well and are labeled with $pl = 0$. So in Figure 3, v_9 alone is considered p_4 and hence $pl(p_4) = 0$ and $cv(p_4) = 9$ along with one crossing edge where $pcel(v_9, v_7) = 13$.

2.3 Finding Matches for the Patches

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, patches are created from G_1 as explained in Section 2.2. These patches are subgraphs in G_1 that have to be found in G_2. This is a difficult process since subgraph isomorphism is NP-complete [16] [17].

Suppose that the set $P = \{p_1, p_2, p_3, p_4\}$ of patches is created for the graph G_1 as shown in Figure 4. The algorithm tries now to find

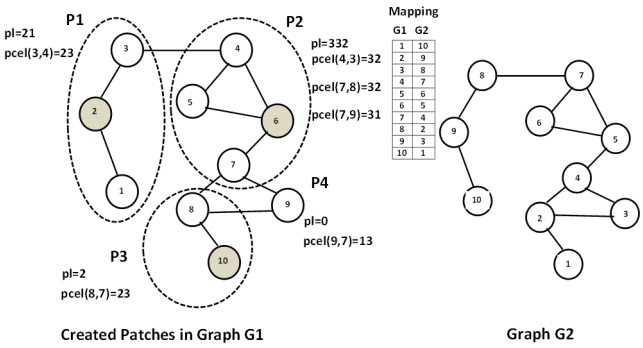

Fig. 4. Finding Matches in Graph G_2 for the Patches of Graph G_1

matches for each member in the set of patches P on the second graph G_2. The patches will be tested against graph G_2 in descending order of their $pl's$. So the first one will be p_1 with $pl(p_1) = 332$. The mapping or patching process is explained on Figure 4.

Notice that the vertices of graph G_2 are renumbered in different order. Recall that $cv(p_1) = 6$, i.e. the central vertex for p_1 is v_6 and $pl(p_1) = 332$. First, v_1 in G_2 is verified whether it is a candidate place to patch p_1. Clearly it is not a candidate. Next, v_2 in G_2 is verified and so on until v_4 in G_2 is found to be a candidate place for p_1. But after testing the labels of patch crossing edges of p_1, inconsistency is found because p_1 has 3 outgoing edges with $pcel's$ 32, 33, and 23 that do not have matching $pcel's$ in G_2 centered in v_4. Therefore, the algorithm continues to find a candidate match for p_1 until one is found and proceeds to the next patch or it exhaust all possibilities for p_1 and declare absence of isomorphism. The next candidate is v_5 in G_2 and it is a full match.

All vertices of p_1 in G_1 are mapped to their corresponding ones of G_2. The mapping from G_1 to G_2 is as follows: $6 \rightarrow 5$, $4 \rightarrow 7$, $5 \rightarrow 6$, and $7 \rightarrow 4$. So p_1 is patched to a tentative correct place on G_2.

Next, a try is made to find a match on G_2 for the patch p_2 of G_1. Since the algorithm is recursive, if all possibilities are exhausted here, it retracts and tries to find the next possible match for the previous patch and so on.

Table 1. Graphs Categories (100 randomly generated graphs for each category

No. of Nodes	Density	No. of Edges
6	40	6
6	70	10
7	30	6
7	60	12
10	25	11
10	50	22
10	75	33
25	25	74
25	50	150
25	75	225
50	25	306
50	50	612
50	75	918
100	25	1237
100	50	2475
100	75	3712
200	25	4975
200	50	9950
200	75	14925
500	25	31187
500	50	62375
500	75	93562
1000	25	124875
1000	50	249750
1000	75	374625
2500	25	780937
2500	50	1561875
2500	75	2342812
5000	25	3124375
5000	50	6248750
5000	75	9373125

Only the vertices in G_2 that are not mapped yet are tested, i.e. excluding the vertices 4, 5, 6, and 7. A match in G_2 with central vertex v_9 is found and the mapping is done as shown in Figure 4.

The algorithm continues to find the rest of matches. If all possibilities are exhausted in the first level without finding a proper match, the absence of an isomorphism is declared. Otherwise, the recursive algorithm continues to find all possible matches that may lead to the existence of an isomorphism.

3. TESTING RESULTS

The testing is done on Windows 7 64-bit OS on a Laptop with the processor Intel(R) Core(TM) i7 cpu with 4GB of RAM. Hundred graphs of small, medium, and large graphs of different categories-sizes and densities have been randomly created. Unfortunately, due to the constraint of the 4GB RAM, a maximum size of 5000 node-graphs were created. Table 1 shows the categories of these created graphs. Notice that loading time of graph details are excluded from the execution times shown in the tables.

To test the algorithm for isomorphic graphs, each graph in all categories is compared with a copy of itself after scrambling the node numbering in the copied graph. So 100 tests were done for each category and full accurate results were obtained. Tables 2 and 3 show the timing of these results.

Table 2. Average no. of patches created and average time in microseconds for comparing isomorphic graphs

No. of Nodes	Density	Avg Time	Avg Patches
6	40	164.7	2.4
6	70	146.1	1.5
7	30	179.8	3.1
7	60	133.4	1.9
10	25	172.2	4.0
10	50	205.3	2.7
10	75	209.8	1.6
25	25	418.5	6.2
25	50	531.7	3.7
25	75	593.8	2.2
50	25	850.6	8.2
50	50	1139.3	4.7
50	75	1277.1	2.8
100	25	1838.5	10.3
100	50	2469.7	5.5
100	75	3456.6	3.3

Table 3. Average no. of patches created and average time in Milliseconds/Seconds for comparing isomorphic graphs

No. of Nodes	Density	Avg Time	Avg Patches
200	25	3.8 m. sec	12.7
200	50	9.6 m. sec	6.6
200	75	19.3 m. sec	3.7
500	25	38.5 m. sec	15.7
500	50	110.8 m. sec	7.9
500	75	221.9 m. sec	4.5
1000	25	231.2 m. sec	18.7
1000	50	761.2 m. sec	9.1
1000	75	1578.2 m. sec	5.1
2500	25	3.7 sec	22.4
2500	50	17.2 sec	10.7
2500	75	38.7 sec	5.9
5000	25	28.1 sec	24.9
5000	50	105.1 sec	11.7
5000	75	246.6 sec	6.5

For comparing different graphs, expectedly non-isomorphic ones, 198 tests were done for each category, 99 times comparing the first graph to each of the other 99 graphs in the category and 99 times comparing the fiftieth graph to each of the other 99 graphs in the category. Tables 4 and 5 show the timing of these results.

In some categories, comparing supposedly different graphs, the algorithm declared them to be isomorphic. These graphs have been manually tested and found to be in fact isomorphic. Hence, they have been excluded from the average timing of Table 4. Table 6 shows the categories and the number of randomly generated isomorphic graphs that were expected to be non-isomorphic.

Notice from Figure 5 that the denser the graphs are the less number of patches are created by the algorithm. The reason is that a lot of nodes in dense graphs generated by the random graph generator with size n will have degrees close to $n-1$ or equal to it. Due to the greedy concept, these nodes will be used as the centers of the first patches created. Recall from Section 2.2 that when a patch is created, then all children of that patch are excluded from any further patches created which leads to less remaining patches.

Table 4. Average no. of patches created and average time in microseconds for comparing non-isomorphic graphs

No. of Nodes	Density	Avg Time	Avg Patches
6	40	124.4	3.0
6	70	103.3	1.0
7	30	93.9	3.0
7	60	107.5	2.0
10	25	109.9	4.0
10	50	132.5	2.0
10	75	144.2	2.0
25	25	234.1	7.0
25	50	300.5	3.5
25	75	346.1	2.5
50	25	405.0	7.0
50	50	602.6	4.5
50	75	875.5	3.0
100	25	1175.7	10.0
100	50	1888.6	5.5
100	75	3125.0	3.0

Table 5. Average no. of patches created and average time in Milliseconds/Seconds for comparing non-isomorphic graphs

No. of Nodes	Density	Avg Time	Avg Patches
200	25	3.0 m. sec	13.5
200	50	8.4 m. sec	7.5
200	75	18.1 m. sec	4.5
500	25	34.9 m. sec	16.5
500	50	108.5 m. sec	8.0
500	75	219.3 m. sec	4.0
1000	25	224.1 m. sec	19.5
1000	50	750.5 m. sec	10.0
1000	75	1563.6 m. sec	5.0
2500	25	3.72 sec	22.5
2500	50	17.3 sec	10.5
2500	75	38.7 sec	6.0
5000	25	26.5 sec	24.5
5000	50	105.0 sec	12.5
5000	75	244.0 sec	6.5

Table 6. Number of randomly generated graphs that happened to be isomorphic in some categories

Nodes	Density	Edges	No. of Graphs
6	40%	6	18
6	70%	10	26
7	30%	6	34
7	60%	12	1
10	75%	33	5

Notice that execution time shown in Figure 6 and Figure 7 is little more than that of Figure 8 and Figure 9 respectively. That is because in the case of comparing non isomorphic graphs, at a certain level of finding a match, all possibilities are exhausted before examining all nodes and the algorithm stops to declare the result. On the other hand, in the case of comparing isomorphic graphs, many retractions occur but all nodes are examined and a match is found at the end along with the mapping that is done to all nodes from the first graph to the other.

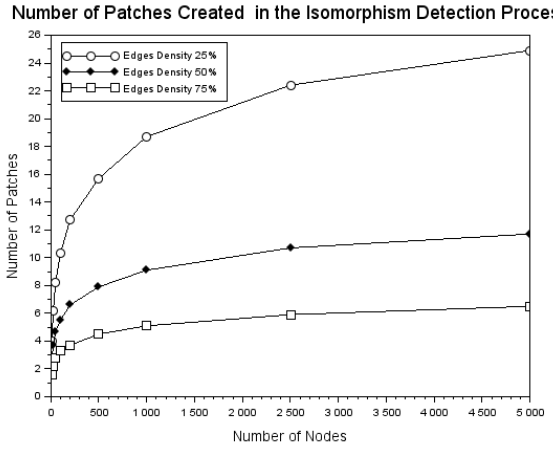

Fig. 5. Number of Created Patches

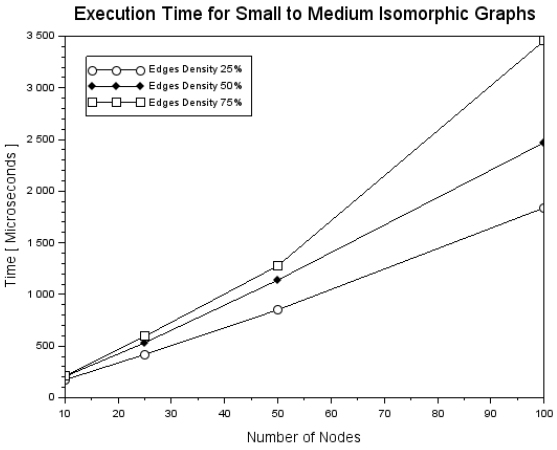

Fig. 6. Timing for Small to Medium Size Isomorphic Graphs

In any of these mentioned figures and for a certain graph size, the execution time shown increases as the density increases. This may lead to a claim of contradiction to what is mentioned in the abstract that ”the larger the created patches are, the faster the algorithm is” since larger patches means less number of these patches in a graph and hence should lead to a faster execution. That is true when graphs under testing have fixed densities. But more dense graphs lead to more number of checks and comparisons to nodes and edges.

4. THE ALGORITHM

Given two graphs G and H which are represented by two dimensional matrices, Algorithm 1 decides whether G and H are isomorphic. The algorithm assumes that the given graphs are of same number of vertices.

Algorithm 1 PPP Isomorphism Detection Algorithm

procedure TESTFORISOMORPHISM(G , H)
  $availableGNodes$ = all $v \in G$ // global variable
  $availableHNodes$ = all $v \in H$ // global variable
  $patchesSet$ = createPatchesInGraphG
  if findMatchesInHForPatchesInG(p_1, $p_{patchesSetSize}$) = true **then**
    G and H are isomorphic
  else
    G and H are NOT isomorphic
  end if
end procedure

procedure CREATEPATCHESINGRAPHG
  Compute $deg(v_i) \; \forall \; v_i \in G$
  Compute $patchLabel \; \forall \; v \in G$
  Sort all $patchLabel's$ in G in descending order
  $pSet$ = empty
  while $availableGNodes$ *not* empty **do**
    Create a patch p_i centered with a node of highest $patchLabel$ available
    Remove this node along with all nodes in this p_i from $AvailableGNodes$
    Add the created Patch p_i to $pSet$
  end while
  return $pSet$
end procedure

procedure FINDMATCHESINHFORPATCHESING(p_{start} , p_{end})
  if $start = end$ **then**
    if findMatch(p_{start}) = true **then**
      return true
    else
      Add all $v \in p$ of the last found match back to $availableHNodes$
      return false
    end if
  else
    if findMatch(p_{start}) = true **then**
      if findMatchesInHForPatchesInG($p_{start+1}$, p_{end}) **then**
        return true
      else
        return false
      end if
    else
      return false
    end if
  end if
end procedure

procedure FINDMATCH(p_i)
  find in H a match for patch p_i from $availableHNodes$
  if matchFound and $pcel's$ of p_i are preserved **then**
    remove all v in the found match from $availableHNodes$
    return true
  else
    return false
  end if
end procedure

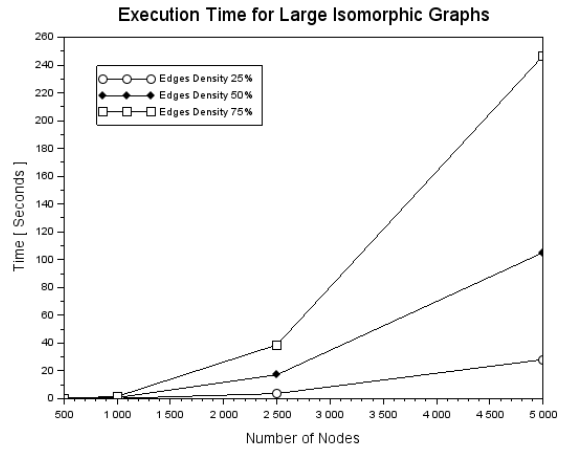

Fig. 7. Timing for Large Size Isomorphic Graphs

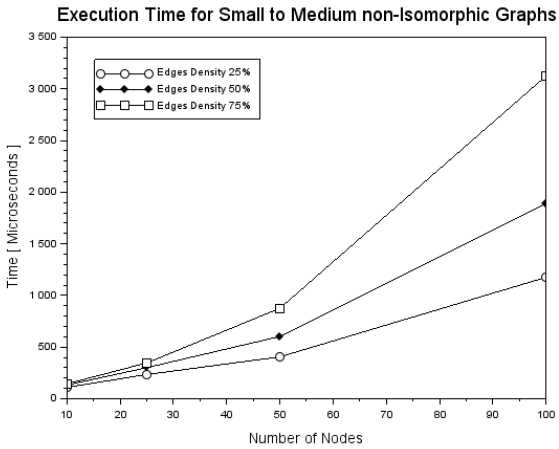

Fig. 8. Timing for Small to Medium Size non-Isomorphic Graphs

5. CONCLUSION

An algorithm with a new approach namely, $ppp - Technique$, that is fast and efficient to solve the isomorphism problem for random unlabeled graphs that are found in many applications today was presented. Unlabeled graphs were targeted here because these types of graphs have the least amount of heuristics that can aid the process of isomorphism detection. We believe that researchers will find this technique as a new window to be further experimented and improved in the hope of finding a solution to the troublesome problem- Graph Isomprphism.

6. REFERENCES

[1] M. Zaslavskiy, F. Bach, and J.-P. Vert, "A path following algorithm for the graph matching problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 12, pp. 2227–2242, 2008.

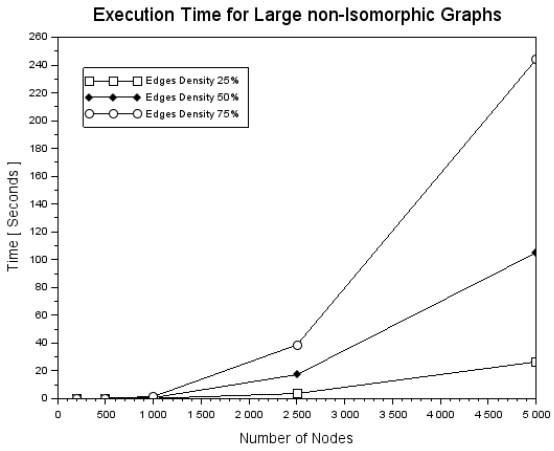

Fig. 9. Timing for Large Size non-Isomorphic Graphs

[2] K. Liu, Y. Zhang, K. Lu, X. Wang, X. Wang, and G. Tian, "Mapeff: An effective graph isomorphism agorithm based on the discrete-time quantum walk," *Entropy*, vol. 21, no. 6, p. 569, 2019.

[3] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International journal of pattern recognition and artificial intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[4] A. Dawar and K. Khan, "Constructing hard examples for graph isomorphism," *arXiv preprint arXiv:1809.08154*, 2018.

[5] L. Babai, A. Dawar, P. Schweitzer, and J. Torán, "The graph isomorphism problem (dagstuhl seminar 15511)," in *Dagstuhl Reports*, vol. 5, no. 12. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[6] G. L. Miller, "Graph isomorphism, general remarks," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 128–142, 1979.

[7] B. D. McKay and A. Piperno, "Practical graph isomorphism, ii," *Journal of Symbolic Computation*, vol. 60, pp. 94–112, 2014.

[8] G. Nikolentzos, P. Meladianos, and M. Vazirgiannis, "Matching node embeddings for graph similarity," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[9] V. Bonnici, R. Giugno, A. Pulvirenti, D. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BMC bioinformatics*, vol. 14, no. S7, p. S13, 2013.

[10] S. B. Akers, "The star graph: An attractive alternative to the n-cube," in *Proc. Int'l Conf. Parallel Processing., 1987*, 1987.

[11] M. Verma and S. Sharma, "A greedy approach for coverage hole detection and restoration in wireless sensor networks," *Wireless Personal Communications*, vol. 101, no. 1, pp. 75–86, 2018.

[12] H. Joudrier and F. Thiard, "A greedy approach for a rolling stock management problem using multi-interval constraint propagation," *Annals of Operations Research*, vol. 271, no. 2, pp. 1165–1183, 2018.

[13] Y. Duan, J. Wu, and H. Zheng, "A greedy approach for carpool scheduling optimisation in smart cities," *International Journal of Parallel, Emergent and Distributed Systems*, pp. 1–15, 2018.

[14] E. Oh and C. Woo, "Performance analysis of dynamic channel allocation based on the greedy approach for orthogonal frequency-division multiple access downlink systems," *International Journal of Communication Systems*, vol. 25, no. 7, pp. 953–961, 2012.

[15] N. Meghanathan, "A greedy algorithm for neighborhood overlap-based community detection," *Algorithms*, vol. 9, no. 1, p. 8, 2016.

[16] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, 2001, pp. 149–159.

[17] C. McCreesh, P. Prosser, C. Solnon, and J. Trimble, "When subgraph isomorphism is really hard, and why this matters for graph databases," *Journal of Artificial Intelligence Research*, vol. 61, pp. 723–759, 2018.